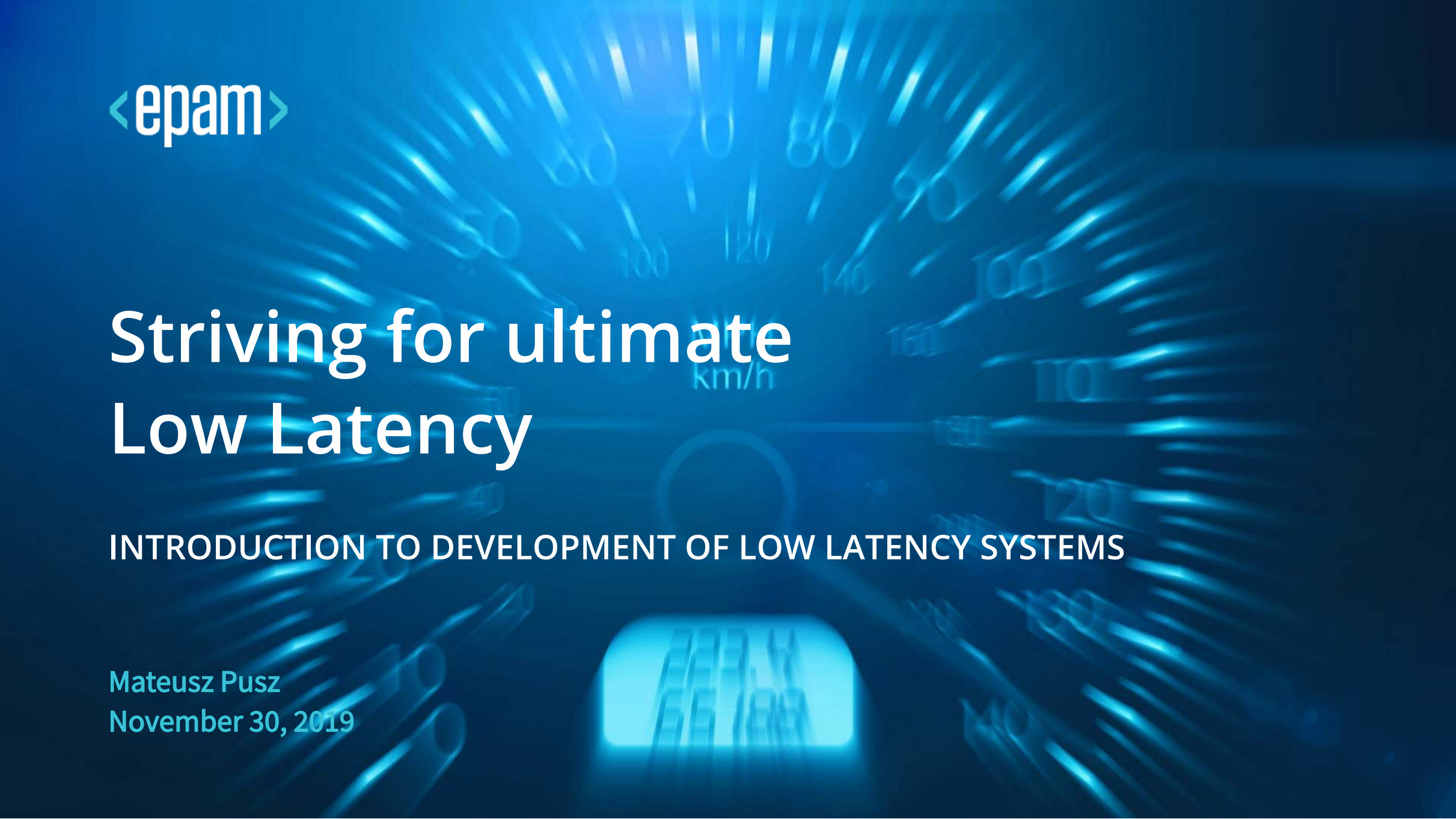


Striving for ultimate Low Latency

INTRODUCTION TO DEVELOPMENT OF LOW LATENCY SYSTEMS

Mateusz Pusz
November 30, 2019

ISO C++ Committee (WG21) Study Group 14 (SG14)

Special thanks to Carl Cook and Chris Kohlhoff

Latency vs Throughput

Latency vs Throughput

Latency is the time required to perform some action or to produce some result. Measured in units of time like hours, minutes, seconds, nanoseconds, or clock periods.

Latency vs Throughput

Latency is the time required to perform some action or to produce some result. Measured in units of time like hours, minutes, seconds, nanoseconds, or clock periods.

Throughput is the number of such actions executed or results produced per unit of time. Measured in units of whatever is being produced per unit of time.

What do we mean by Low Latency?

What do we mean by Low Latency?

Low Latency allows human-unnoticeable delays between an input being processed and the corresponding output providing real time characteristics

What do we mean by Low Latency?

Low Latency allows human-unnoticeable delays between an input being processed and the corresponding output providing real time characteristics

Especially important for internet connections utilizing services such as **trading**, **online gaming**, and **VoIP**

Why do we strive for Low latency?

Why do we strive for Low latency?

- In VoIP substantial delays between input from conversation participants may impair their communication

Why do we strive for Low latency?

- In VoIP substantial delays between input from conversation participants may impair their communication
- In online gaming a player with a high latency internet connection may show slow responses in spite of superior tactics or appropriate reaction time

Why do we strive for Low latency?

- In VoIP substantial delays between input from conversation participants may impair their communication
- In online gaming a player with a high latency internet connection may show slow responses in spite of superior tactics or appropriate reaction time
- Within capital markets the proliferation of algorithmic trading requires firms to react to market events faster than the competition to increase profitability of trades

High-Frequency Trading (HFT)

A program trading platform that uses powerful computers to transact a large number of orders at very fast speeds

-- Investopedia

High-Frequency Trading (HFT)

> A program trading platform that uses powerful computers to transact a large number of orders at very fast speeds
>
> *-- Investopedia*

- Using *complex algorithms* to analyze multiple markets and execute orders based on market conditions

High-Frequency Trading (HFT)

A program trading platform that uses powerful computers to transact a large number of orders at very fast speeds

-- Investopedia

- Using *complex algorithms* to analyze multiple markets and execute orders based on market conditions
- Buying and selling of securities *many times over a period of time* (often hundreds of times an hour)

High-Frequency Trading (HFT)

> A program trading platform that uses powerful computers to transact a large number of orders at very fast speeds
>
> *-- Investopedia*

- Using *complex algorithms* to analyze multiple markets and execute orders based on market conditions
- Buying and selling of securities *many times over a period of time* (often hundreds of times an hour)
- Done to *profit from time-sensitive opportunities* that arise during trading hours

High-Frequency Trading (HFT)

> A program trading platform that uses powerful computers to transact a large number of orders at very fast speeds
>
> *-- Investopedia*

- Using *complex algorithms* to analyze multiple markets and execute orders based on market conditions
- Buying and selling of securities *many times over a period of time* (often hundreds of times an hour)
- Done to *profit from time-sensitive opportunities* that arise during trading hours
- Implies *high turnover of capital* (i.e. one's entire capital or more in a single day)

High-Frequency Trading (HFT)

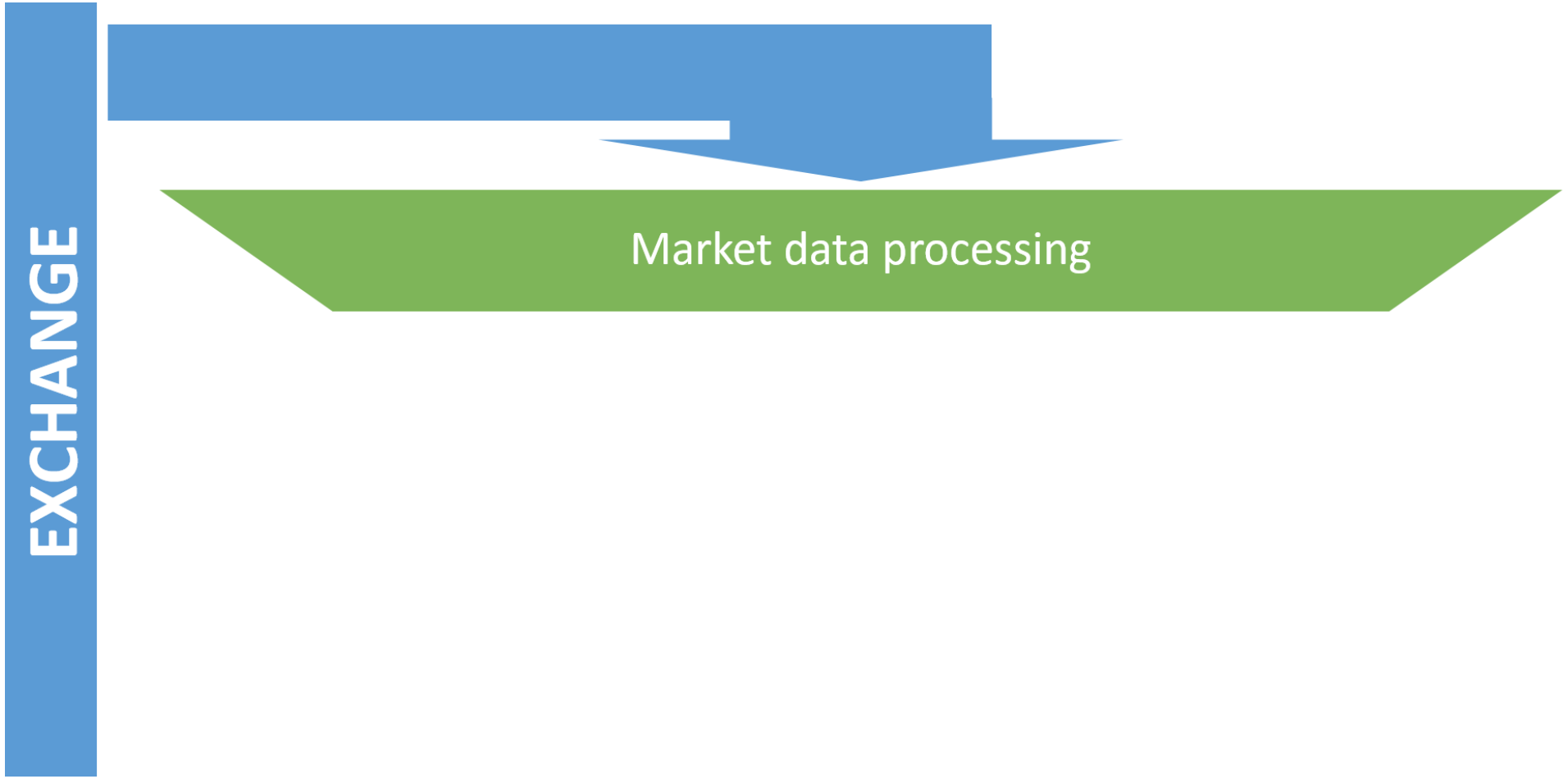
> A program trading platform that uses powerful computers to transact a large number of orders at very fast speeds
>
> *-- Investopedia*

- Using *complex algorithms* to analyze multiple markets and execute orders based on market conditions
- Buying and selling of securities *many times over a period of time* (often hundreds of times an hour)
- Done to *profit from time-sensitive opportunities* that arise during trading hours
- Implies *high turnover of capital* (i.e. one's entire capital or more in a single day)
- Typically, the traders with **the fastest execution speeds** are more profitable

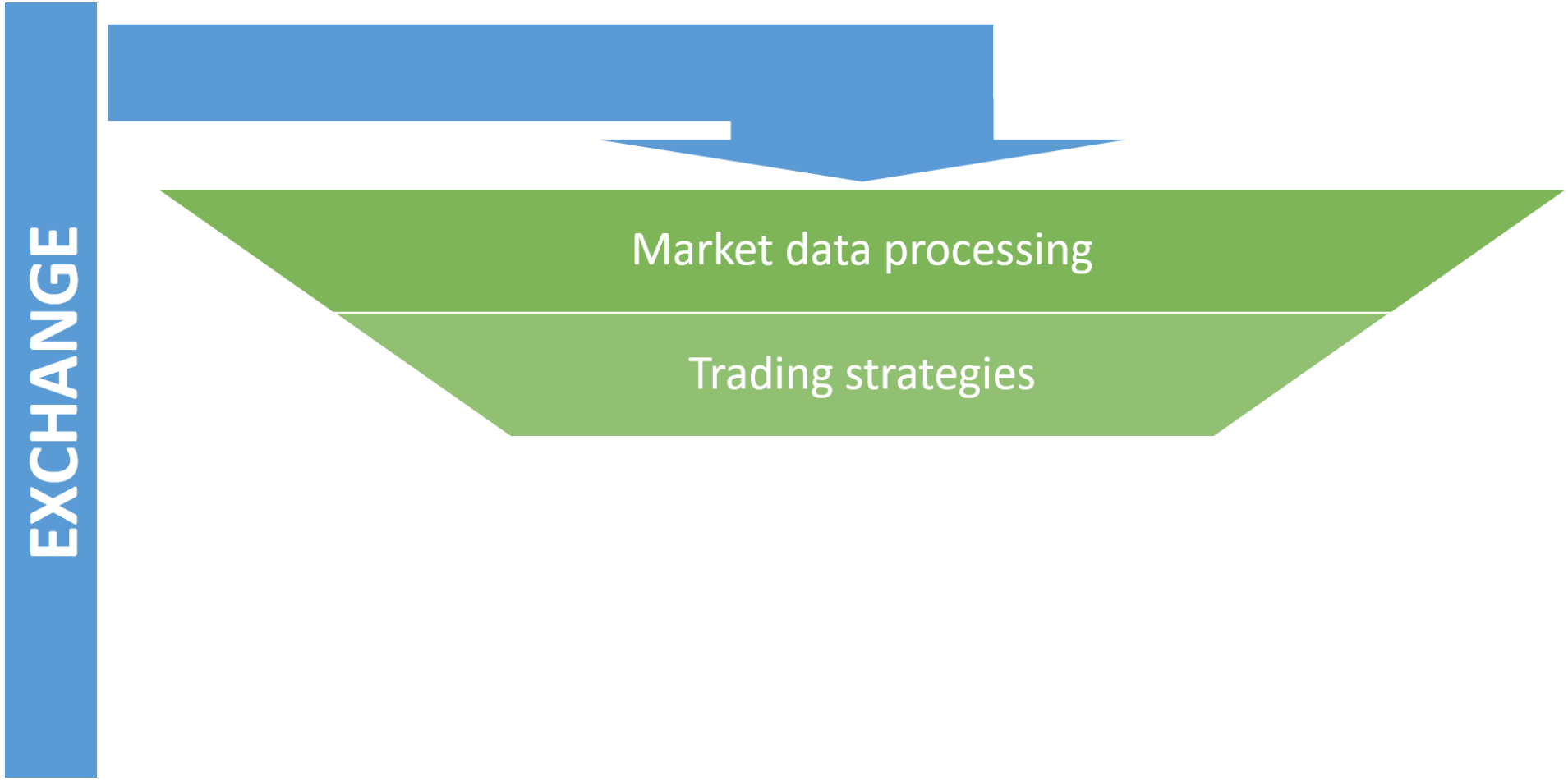
Market data processing

Market data processing

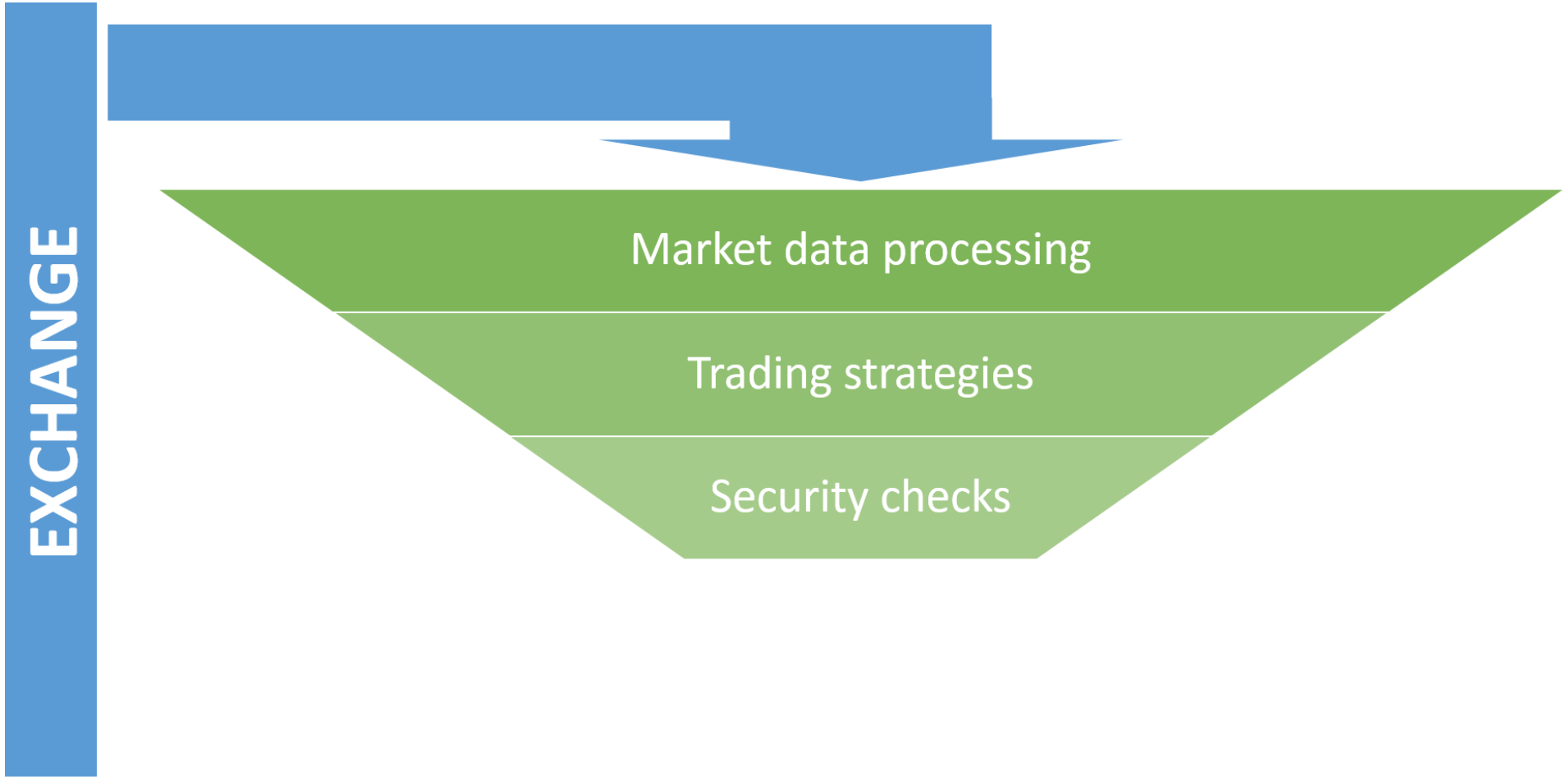
Market data processing



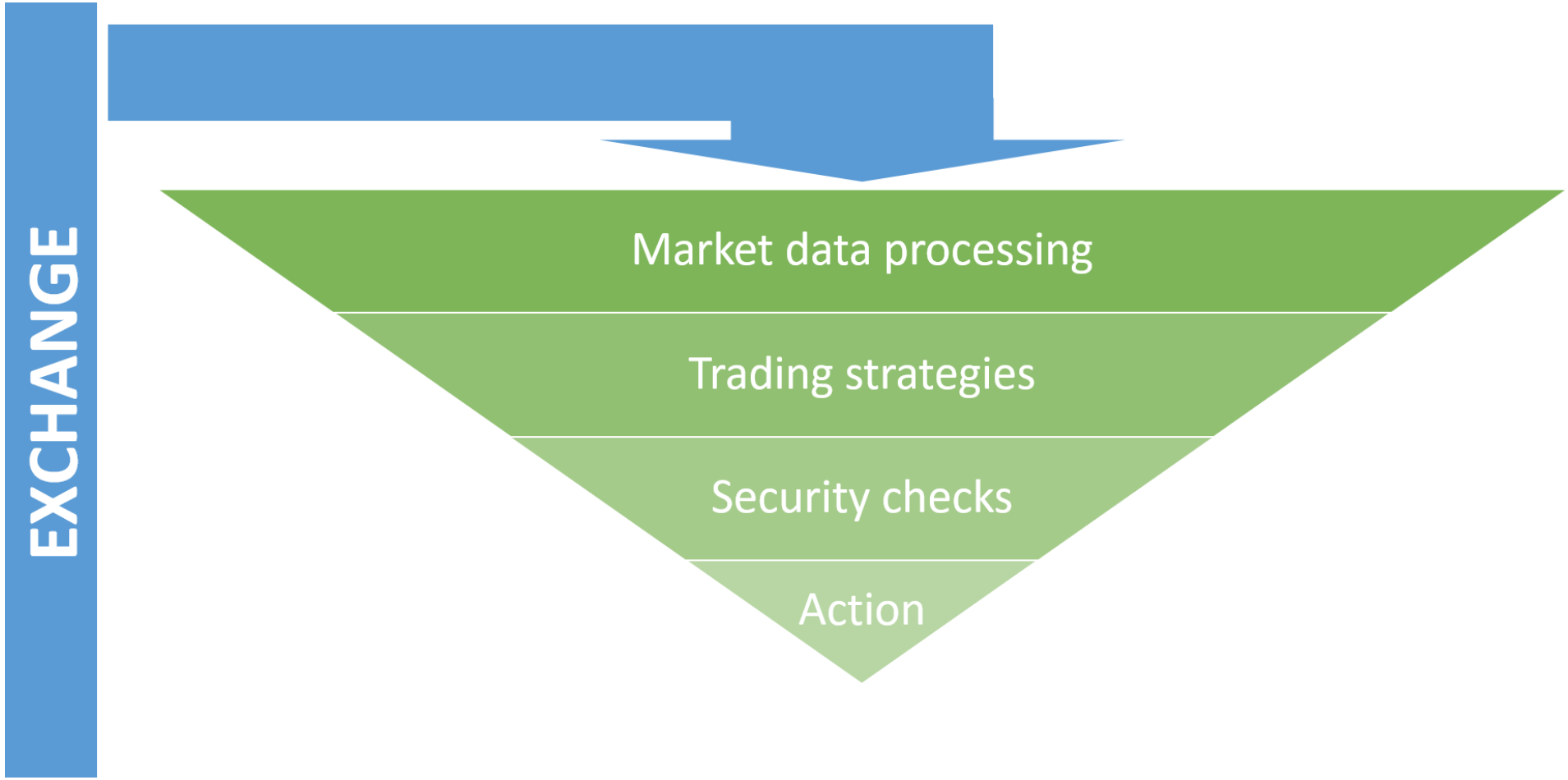
Market data processing



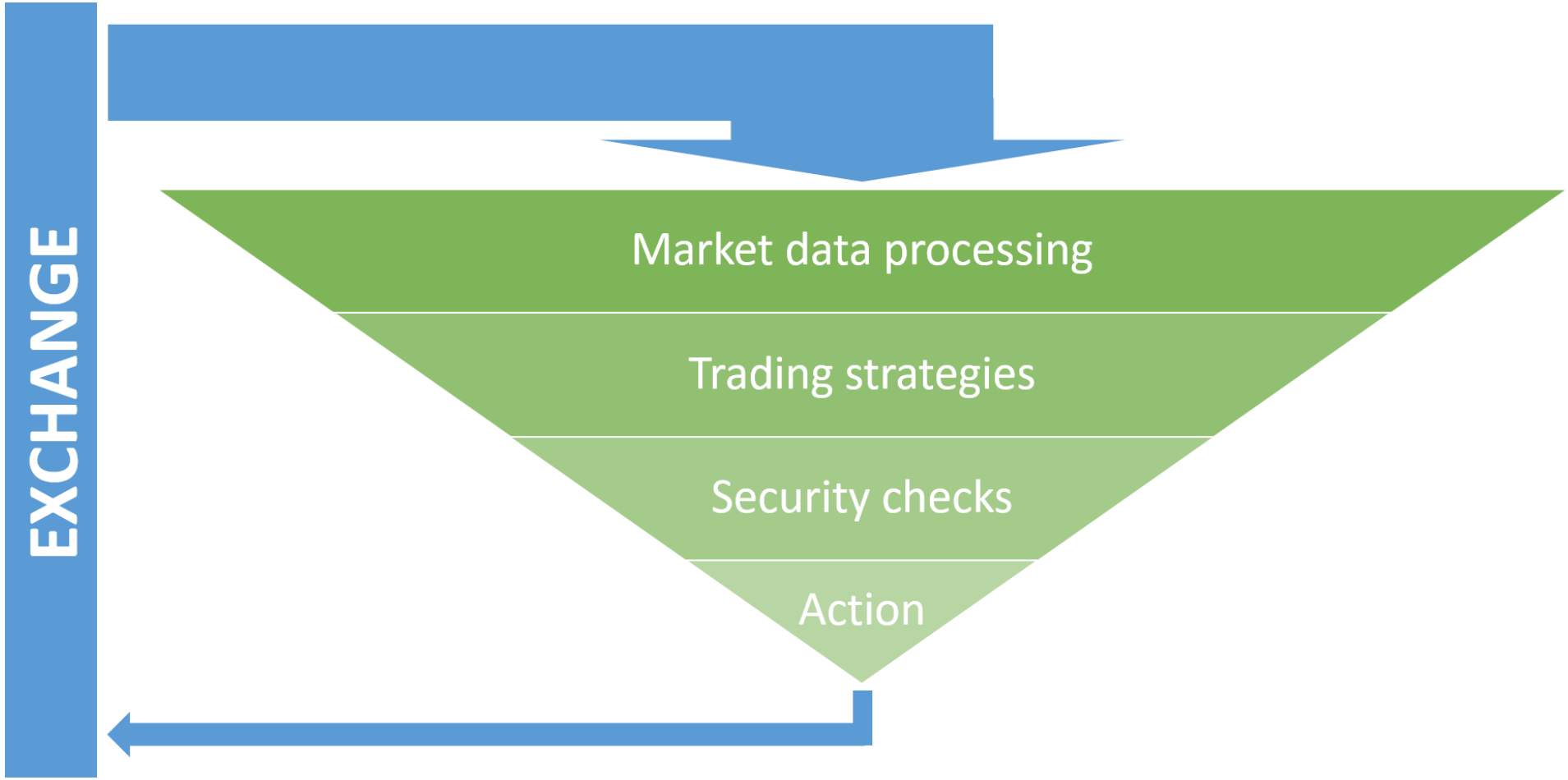
Market data processing



Market data processing



Market data processing



How fast do we do?

ALL SOFTWARE APPROACH

1-10us

ALL HARDWARE APPROACH

100-1000ns

How fast do we do?

ALL SOFTWARE APPROACH

1-10us

ALL HARDWARE APPROACH

100-1000ns

How fast do we do?

ALL SOFTWARE APPROACH

1-10us

ALL HARDWARE APPROACH

100-1000ns

- Average human eye blink takes 350 000us (1/3s)
- Millions of orders can be traded that time

What if something goes wrong?

What if something goes wrong?

KNIGHT CAPITAL

- In 2012 was the largest trader in U.S. equities
- Market share
  - 17.3% on NYSE
  - 16.9% on NASDAQ
- Had approximately *$365 million* in cash and equivalents
- Average **daily** trading volume
  - 3.3 billion trades
  - trading over *21 billion* dollars

What if something goes wrong?

KNIGHT CAPITAL

-- LinkedIn

- In 2012 was the largest trader in U.S. equities
- Market share
  - 17.3% on NYSE
  - 16.9% on NASDAQ
- Had approximately *$365 million* in cash and equivalents
- Average **daily** trading volume
  - 3.3 billion trades
  - trading over *21 billion* dollars
- **pre-tax loss of $440 million in 45 minutes**

How a software bug made Knight Capital lose $500M in a day & almost go bankrupt

C++ often not the most important part of the system

C++ often not the most important part of the system

1. Low Latency network

C++ often not the most important part of the system

1. Low Latency network
2. Modern hardware

C++ often not the most important part of the system

1. Low Latency network
2. Modern hardware
3. BIOS profiling

C++ often not the most important part of the system

1. Low Latency network
2. Modern hardware
3. BIOS profiling
4. Kernel profiling

C++ often not the most important part of the system

1. Low Latency network
2. Modern hardware
3. BIOS profiling
4. Kernel profiling
5. OS profiling

Spin, pin, and drop-in

Spin, pin, and drop-in

SPIN

- Don't sleep
- Don't context switch
- Prefer single-threaded scheduling
- Disable locking and thread support
- Disable power management
- Disable C-states
- Disable interrupt coalescing

Spin, pin, and drop-in

SPIN

- Don't sleep
- Don't context switch
- Prefer single-threaded scheduling
- Disable locking and thread support
- Disable power management
- Disable C-states
- Disable interrupt coalescing

PIN

- Assign CPU affinity
- Assign interrupt affinity
- Assign memory to NUMA nodes
- Consider the physical location of NICs
- Isolate cores from general OS use
- Use a system with a single physical CPU

Spin, pin, and drop-in

SPIN

- Don't sleep
- Don't context switch
- Prefer single-threaded scheduling
- Disable locking and thread support
- Disable power management
- Disable C-states
- Disable interrupt coalescing

PIN

- Assign CPU affinity
- Assign interrupt affinity
- Assign memory to NUMA nodes
- Consider the physical location of NICs
- Isolate cores from general OS use
- Use a system with a single physical CPU

DROP-IN

- Choose NIC vendors based on performance and availability of drop-in kernel bypass libraries
- Use the kernel bypass library


Let's scope on the software

Characteristics of Low Latency software

- Typically only a **small part of code is really important** (a fast/hot path)

Characteristics of Low Latency software

- Typically only a **small part of code is really important** (a fast/hot path)
- This code is not executed often
- When it is executed it has to
  - **start and finish as soon as possible**
  - have **predictable and reproducible** performance (low jitter)

Characteristics of Low Latency software

- Typically only a **small part of code is really important** (a fast/hot path)
- This code is not executed often
- When it is executed it has to
  - **start and finish as soon as possible**
  - have **predictable and reproducible** performance (low jitter)
- Multithreading increases latency
  - we need low latency and **not throughput**
  - concurrency (even on different cores) trashes CPU caches above L1, share memory bus, shares IO, shares network

Characteristics of Low Latency software

- Typically only a **small part of code is really important** (a fast/hot path)
- This code is not executed often
- When it is executed it has to
  - **start and finish as soon as possible**
  - have **predictable and reproducible** performance (low jitter)
- Multithreading increases latency
  - we need low latency and **not throughput**
  - concurrency (even on different cores) trashes CPU caches above L1, share memory bus, shares IO, shares network
- **Mistakes are really expensive**
  - good error checking and recovery is mandatory
  - one second is 4 billion CPU instructions (a lot can happen that time)

How to develop software that have predictable performance?

How to develop software that have predictable performance?

It turns out that the more important question here is...

How NOT to develop software that have predictable performance?

How NOT to develop software that have predictable performance?

- In Low Latency system we care a lot about WCET (Worst Case Execution Time)
- In order to limit WCET we should limit the usage of specific C++ language features

How NOT to develop software that have predictable performance?

- In Low Latency system we care a lot about WCET (Worst Case Execution Time)
- In order to limit WCET we should limit the usage of specific C++ language features
- A task not only for developers but also for code architects

The list of things to avoid on fast path

The list of things to avoid on fast path

1. C++ tools that trade performance for usability (e.g. **`std::shared_ptr<T>`**, **`std::function<>`**)

The list of things to avoid on fast path

1. C++ tools that trade performance for usability (e.g. **`std::shared_ptr<T>`**, **`std::function<>`**)
2. Throwing exceptions on a likely code path

The list of things to avoid on fast path

1. C++ tools that trade performance for usability (e.g. **`std::shared_ptr<T>`**, **`std::function<>`**)
2. Throwing exceptions on a likely code path
3. Dynamic polymorphism

The list of things to avoid on fast path

1. C++ tools that trade performance for usability (e.g. **`std::shared_ptr<T>`**, **`std::function<>`**)
2. Throwing exceptions on a likely code path
3. Dynamic polymorphism
4. Multiple inheritance

The list of things to avoid on fast path

1. C++ tools that trade performance for usability (e.g. **`std::shared_ptr<T>`**, **`std::function<>`**)
2. Throwing exceptions on a likely code path
3. Dynamic polymorphism
4. Multiple inheritance
5. RTTI

The list of things to avoid on fast path

1. C++ tools that trade performance for usability (e.g. `std::shared_ptr<T>`, `std::function<>`)
2. Throwing exceptions on a likely code path
3. Dynamic polymorphism
4. Multiple inheritance
5. RTTI
6. Dynamic memory allocations

std::shared_ptr<T>

```
template<class T>
class shared_ptr;
```

- Smart pointer that retains shared ownership of an object through a pointer
- Several **shared_ptr** objects may own the same object
- The shared object is destroyed and its memory deallocated when the last remaining **shared_ptr** owning that object is either destroyed or assigned another pointer via **operator=** or **reset()**
- Supports user provided deleter

std::shared_ptr<T>

```
template<class T>
class shared_ptr;
```

- Smart pointer that retains **shared ownership** of an object through a pointer
- Several **shared_ptr** objects **may own the same object**
- The shared object is destroyed and its memory deallocated when the last remaining **shared_ptr** owning that object is either destroyed or assigned another pointer via **operator=** or **reset()**
- Supports user provided **deleter**

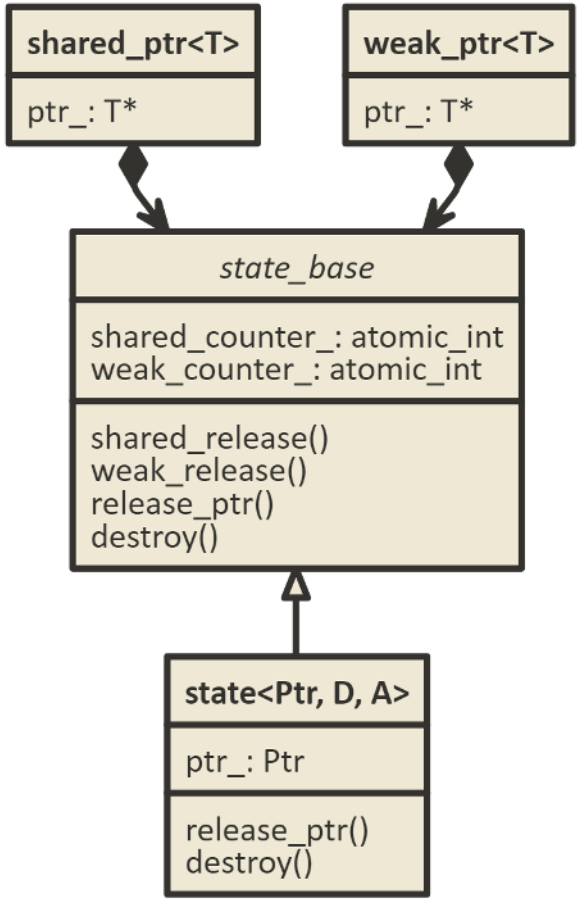
Too often overused by C++ programmers

Question: What is the difference here?

```cpp
void foo()
{
  std::unique_ptr<int> ptr{new int{1}};
  // some code using 'ptr'
}
```

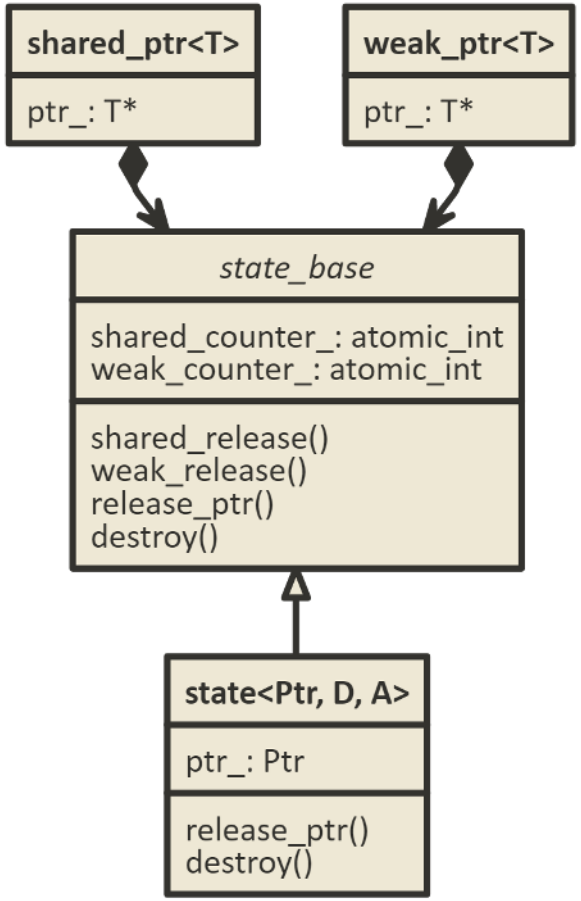
```cpp
void foo()
{
  std::shared_ptr<int> ptr{new int{1}};
  // some code using 'ptr'
}
```

Key `std::shared_ptr<T>` issues

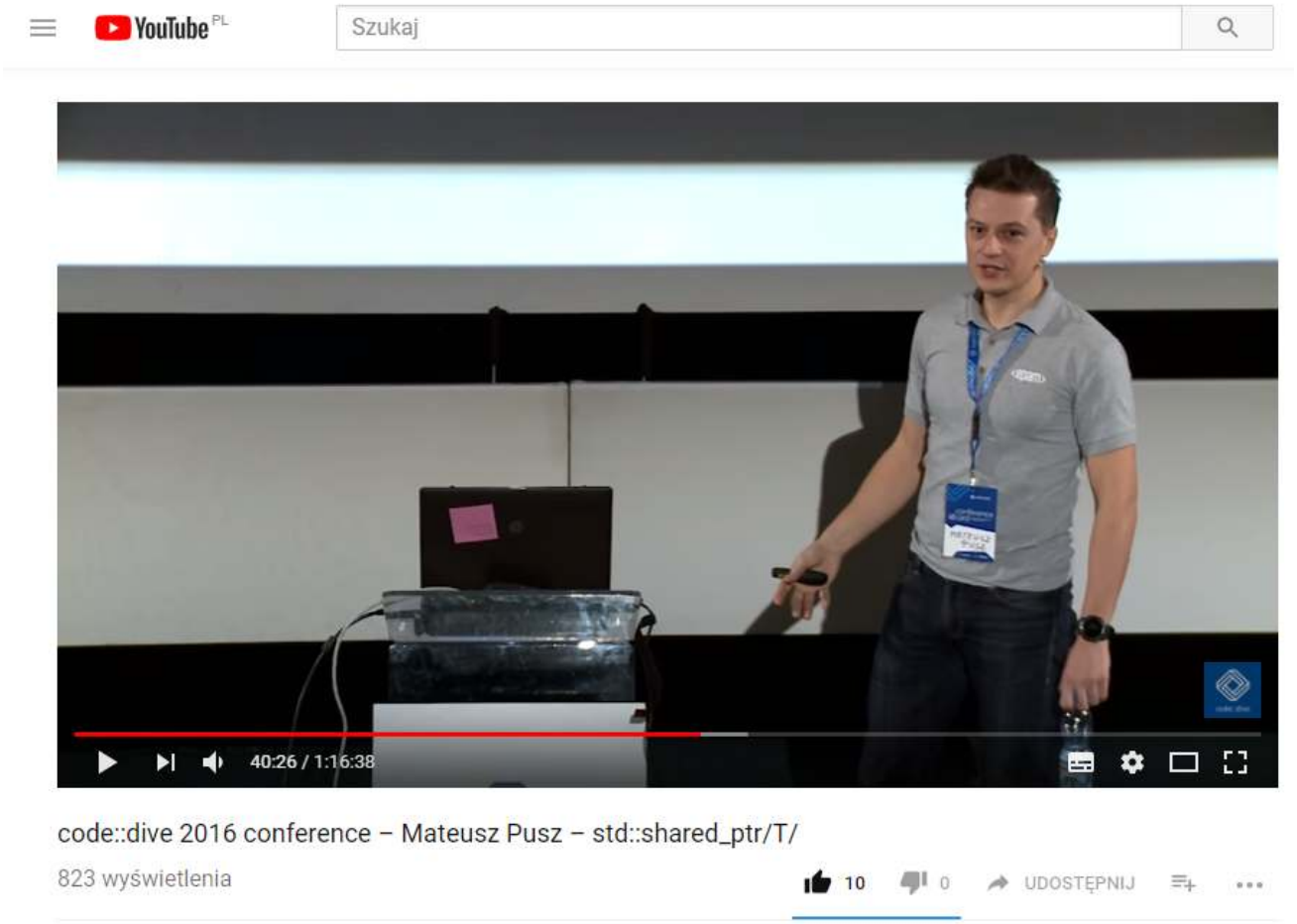


Key `std::shared_ptr<T>` issues

- Shared state
  - performance + memory footprint
- Mandatory synchronization
  - performance
- Type Erasure
  - performance
- `std::weak_ptr<T>` support
  - memory footprint
- Aliasing constructor
  - memory footprint



More info on code::dive 2016



C++ exceptions

C++ exceptions

- Code generated by nearly all C++ compilers *does not introduce significant runtime overhead* for C++ exceptions

C++ exceptions

- Code generated by nearly all C++ compilers *does not introduce significant runtime overhead* for C++ exceptions
- ... **if they are not thrown**

C++ exceptions

- Code generated by nearly all C++ compilers *does not introduce significant runtime overhead* for C++ exceptions
- ... **if they are not thrown**
- Throwing an exception can take *significant and not deterministic time*

C++ exceptions

- Code generated by nearly all C++ compilers *does not introduce significant runtime overhead* for C++ exceptions
- ... **if they are not thrown**
- Throwing an exception can take *significant and not deterministic time*
- *Advantages* of C++ exceptions usage
  - cannot be ignored!
  - simplify interfaces
  - (if not thrown) actually can improve application performance
  - make source code of likely path easier to reason about

C++ exceptions

- Code generated by nearly all C++ compilers *does not introduce significant runtime overhead* for C++ exceptions
- ... **if they are not thrown**
- Throwing an exception can take *significant and not deterministic time*
- *Advantages* of C++ exceptions usage
  - cannot be ignored!
  - simplify interfaces
  - (if not thrown) actually can improve application performance
  - make source code of likely path easier to reason about

Not using C++ exceptions is not an excuse to write not exception-safe code!

Polymorphism

Polymorphism

DYNAMIC

```
class base : noncopyable {
  virtual void setup() = 0;
  virtual void run() = 0;
  virtual void cleanup() = 0;
public:
  virtual ~base() = default;
  void process()
  {
    setup();
    run();
    cleanup();
  }
};

class derived : public base {
  void setup() override   { /* ... */ }
  void run() override     { /* ... */ }
  void cleanup() override { /* ... */ }
};
```

Polymorphism

DYNAMIC

```
class base : noncopyable {
  virtual void setup() = 0;
  virtual void run() = 0;
  virtual void cleanup() = 0;
public:
  virtual ~base() = default;
  void process()
  {
    setup();
    run();
    cleanup();
  }
};

class derived : public base {
  void setup() override   { /* ... */ }
  void run() override     { /* ... */ }
  void cleanup() override { /* ... */ }
};
```

- Additional pointer stored in an object
- Extra indirection (pointer dereference)
- Often not possible to devirtualize
- Not inlined
- Instruction cache miss

Polymorphism

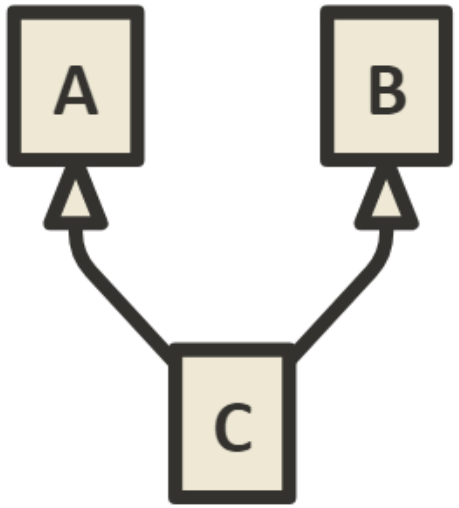
DYNAMIC

```
class base : noncopyable {
  virtual void setup() = 0;
  virtual void run() = 0;
  virtual void cleanup() = 0;
public:
  virtual ~base() = default;
  void process()
  {
    setup();
    run();
    cleanup();
  }
};

class derived : public base {
  void setup() override   { /* ... */ }
  void run() override     { /* ... */ }
  void cleanup() override { /* ... */ }
};
```

STATIC

```
template<class Derived>
class base {
public:
  void process()
  {
    static_cast<Derived*>(this)->setup();
    static_cast<Derived*>(this)->run();
    static_cast<Derived*>(this)->cleanup();
  }
};

class derived : public base<derived> {
  friend class base<derived>;
  void setup()   { /* ... */ }
  void run()     { /* ... */ }
  void cleanup() { /* ... */ }
};
```

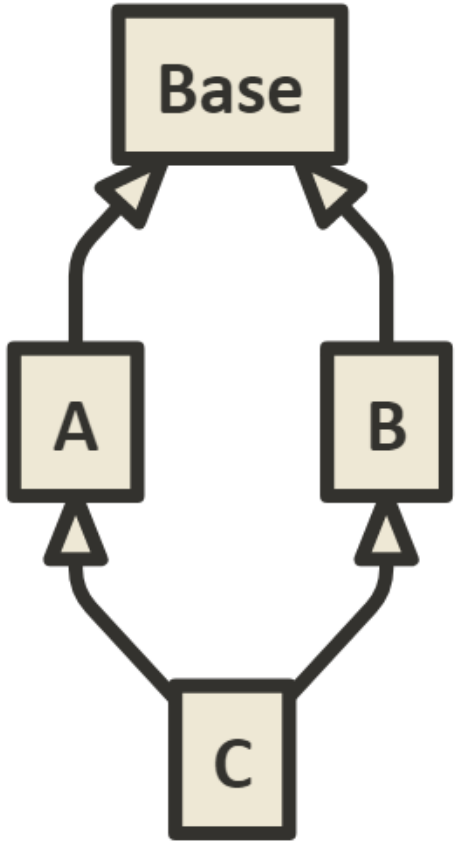
Multiple inheritance



MULTIPLE INHERITANCE

- **`this`** pointer adjustments needed to call member function (for not empty base classes)

Multiple inheritance



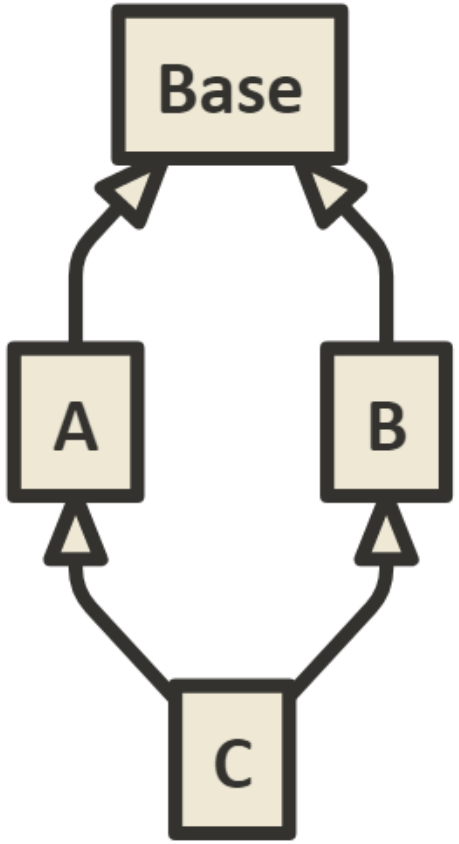
MULTIPLE INHERITANCE

- `this` pointer adjustments needed to call member function (for not empty base classes)

DIAMOND OF DREAD

- Virtual inheritance as an answer
- virtual in C++ means "determined at runtime"
- Extra indirection to access data members

Multiple inheritance



MULTIPLE INHERITANCE

- `this` pointer adjustments needed to call member function (for not empty base classes)

DIAMOND OF DREAD

- Virtual inheritance as an answer
- virtual in C++ means "determined at runtime"
- Extra indirection to access data members

Always prefer composition before inheritance!

RunTime Type Identification (RTTI)

```
class base : noncopyable {
public:
  virtual ~base() = default;
  virtual void foo() = 0;
};
```

```
class derived : public base {
public:
  void foo() override;
  void boo();
};
```

RunTime Type Identification (RTTI)

```
class base : noncopyable {
public:
  virtual ~base() = default;
  virtual void foo() = 0;
};
```

```
class derived : public base {
public:
  void foo() override;
  void boo();
};
```

```
void foo(base& b)
{
  derived* d = dynamic_cast<derived*>(&b);
  if(d) {
    d->boo();
  }
}
```

RunTime Type Identification (RTTI)

```
class base : noncopyable {
public:
  virtual ~base() = default;
  virtual void foo() = 0;
};
```

```
class derived : public base {
public:
  void foo() override;
  void boo();
};
```

```
void foo(base& b)
{
  derived* d = dynamic_cast<derived*>(&b);
  if(d) {
    d->boo();
  }
}
```

Often the sign of a *smelly* design

RunTime Type Identification (RTTI)

```
class base : noncopyable {
public:
  virtual ~base() = default;
  virtual void foo() = 0;
};
```

```
class derived : public base {
public:
  void foo() override;
  void boo();
};
```

```
void foo(base& b)
{
  derived* d = dynamic_cast<derived*>(&b);
  if(d) {
    d->boo();
  }
}
```

- Traversing an inheritance tree
- Comparisons

RunTime Type Identification (RTTI)

```
class base : noncopyable {
public:
  virtual ~base() = default;
  virtual void foo() = 0;
};
```

```
class derived : public base {
public:
  void foo() override;
  void boo();
};
```

```
void foo(base& b)
{
  derived* d = dynamic_cast<derived*>(&b);
  if(d) {
    d->boo();
  }
}
```

- Traversing an inheritance tree
- Comparisons

```
void foo(base& b)
{
  if(typeid(b) == typeid(derived)) {
    derived* d = static_cast<derived*>(&b);
    d->boo();
  }
}
```

- Only one comparison of **`std::type_info`**
- Often only one runtime pointer compare

Dynamic memory allocations

- *General purpose* operation
- *Nondeterministic* execution performance
- Causes memory *fragmentation*
- May *fail* (error handling is needed)
- *Memory leaks* possible if not properly handled

Custom allocators to the rescue

- Address *specific needs* (functionality and hardware constrains)
- Typically *low number of* dynamic memory *allocations*
- *Data structures* needed to manage big chunks of memory

Custom allocators to the rescue

- Address *specific needs* (functionality and hardware constrains)
- Typically *low number of* dynamic memory *allocations*
- *Data structures* needed to manage big chunks of memory

```
template<typename T> struct pool_allocator {
  T* allocate(std::size_t n);
  void deallocate(T* p, std::size_t n);
};
```

```
using pool_string = std::basic_string<char, std::char_traits<char>, pool_allocator>;
```

Custom allocators to the rescue

- Address *specific needs* (functionality and hardware constrains)
- Typically *low number of* dynamic memory *allocations*
- *Data structures* needed to manage big chunks of memory

```
template<typename T> struct pool_allocator {
  T* allocate(std::size_t n);
  void deallocate(T* p, std::size_t n);
};
```

```
using pool_string = std::basic_string<char, std::char_traits<char>, pool_allocator>;
```

Preallocation makes the allocator *jitter more stable*, helps in keeping *related data together* and avoiding long term *fragmentation*.

Small Object Optimization (SOO / SSO / SBO)

- *Prevent dynamic memory allocation* for the (common) case of dealing with small objects

Small Object Optimization (SOO / SSO / SBO)

- *Prevent dynamic memory allocation* for the (common) case of dealing with small objects

```
class sso_string {
  char* data_ = u_.sso_;
  size_t size_ = 0;
  union {
    char sso_[16] = "";
    size_t capacity_;
  } u_;
public:
  [[nodiscard]] size_t capacity() const
  {
    return data_ == u_.sso_ ? sizeof(u_.sso_) - 1 : u_.capacity_;
  }

  // ...
};
```

No dynamic allocation

```
template<std::size_t MaxSize>
class inplace_string {
  std::array<value_type, MaxSize + 1> chars_;
public:
  // string-like interface
};
```

No dynamic allocation

```
template<std::size_t MaxSize>
class inplace_string {
  std::array<value_type, MaxSize + 1> chars_;
public:
  // string-like interface
};
```

```
struct db_contact {
  inplace_string<7> symbol;
  inplace_string<15> name;
  inplace_string<15> surname;
  inplace_string<23> company;
};
```

No dynamic allocation

```cpp
template<std::size_t MaxSize>
class inplace_string {
  std::array<value_type, MaxSize + 1> chars_;
public:
  // string-like interface
};
```

```cpp
struct db_contact {
  inplace_string<7> symbol;
  inplace_string<15> name;
  inplace_string<15> surname;
  inplace_string<23> company;
};
```

No dynamic memory allocations or pointer indirections guaranteed with the cost of possibly bigger memory usage

The list of things to do on the fast path

The list of things to do on the fast path

1. Use tools that **improve efficiency without sacrificing performance**

The list of things to do on the fast path

1. Use tools that **improve efficiency without sacrificing performance**
2. Use **compile time** wherever possible

The list of things to do on the fast path

1. Use tools that **improve efficiency without sacrificing performance**
2. Use **compile time** wherever possible
3. **Know your hardware**

The list of things to do on the fast path

1. Use tools that **improve efficiency without sacrificing performance**
2. Use **compile time** wherever possible
3. **Know your hardware**
4. Clearly **isolate cold code from the fast path**

Examples of safe to use C++ features

- **`static_assert()`**
- Automatic type deduction
- Type aliases
- Move semantics
- **`noexcept`**
- **`constexpr`**
- Lambda expressions
- **`type_traits`**
- **`std::string_view`**
- Variadic templates
- and many more...

Do you agree?

The fastest programs are those that do nothing

Compile-time calculations (C++98)

```
int factorial(int n)
{
  int result = n;
  while(n > 1)
    result *= --n;
  return result;
}
```

```
foo(factorial(n));  // not compile-time
foo(factorial(4));  // not compile-time
```

Compile-time calculations (C++98)

```
int factorial(int n)
{
  int result = n;
  while(n > 1)
    result *= --n;
  return result;
}
```

```
foo(factorial(n));  // not compile-time
foo(factorial(4));  // not compile-time
```

```
template<int N>
struct Factorial {
  static const int value =
      N * Factorial<N - 1>::value;
};

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

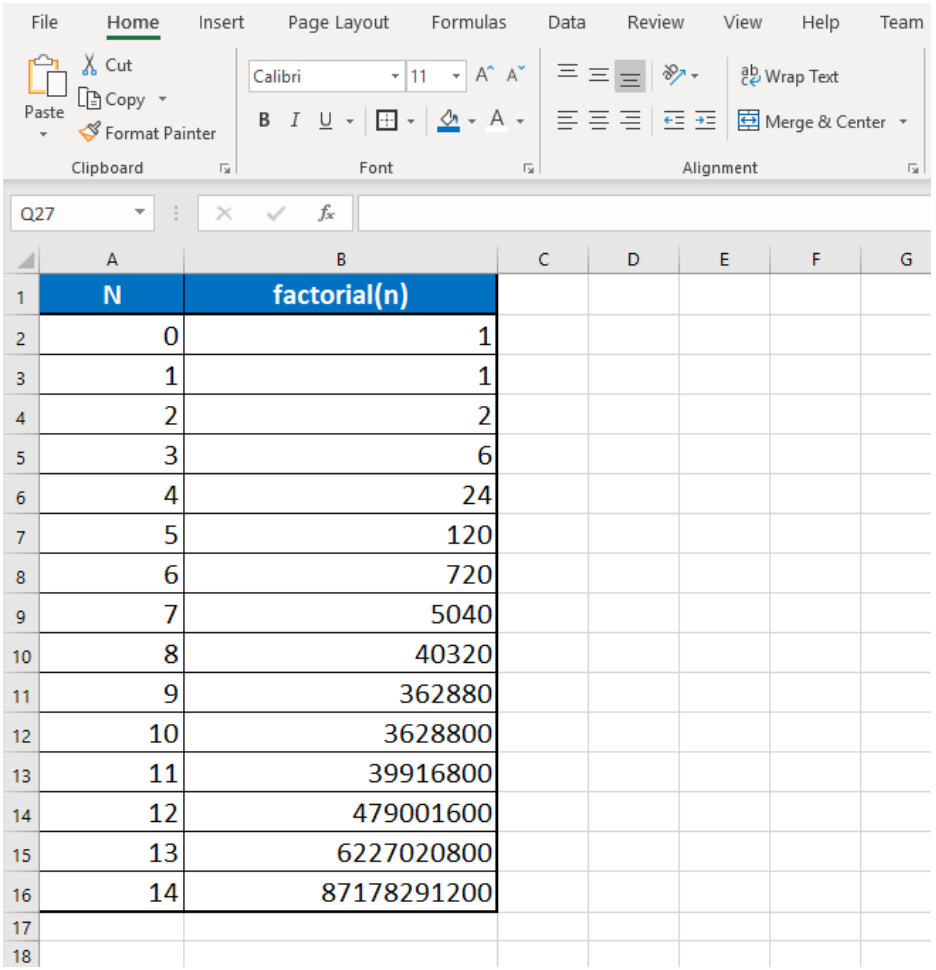
```
int array[Factorial<4>::value];
```

```
const int precalc_values[] = {
  Factorial<0>::value,
  Factorial<1>::value,
  Factorial<2>::value
};
```

```
foo(factorial<4>::value);  // compile-time
// foo(factorial<n>::value);  // compile-time error
```

Just too hard!

- 2 separate implementations
  - hard to keep in sync
- Template *metaprogramming* is hard!
- Easier to precalculate in Excel and hardcode results in the code ;-)

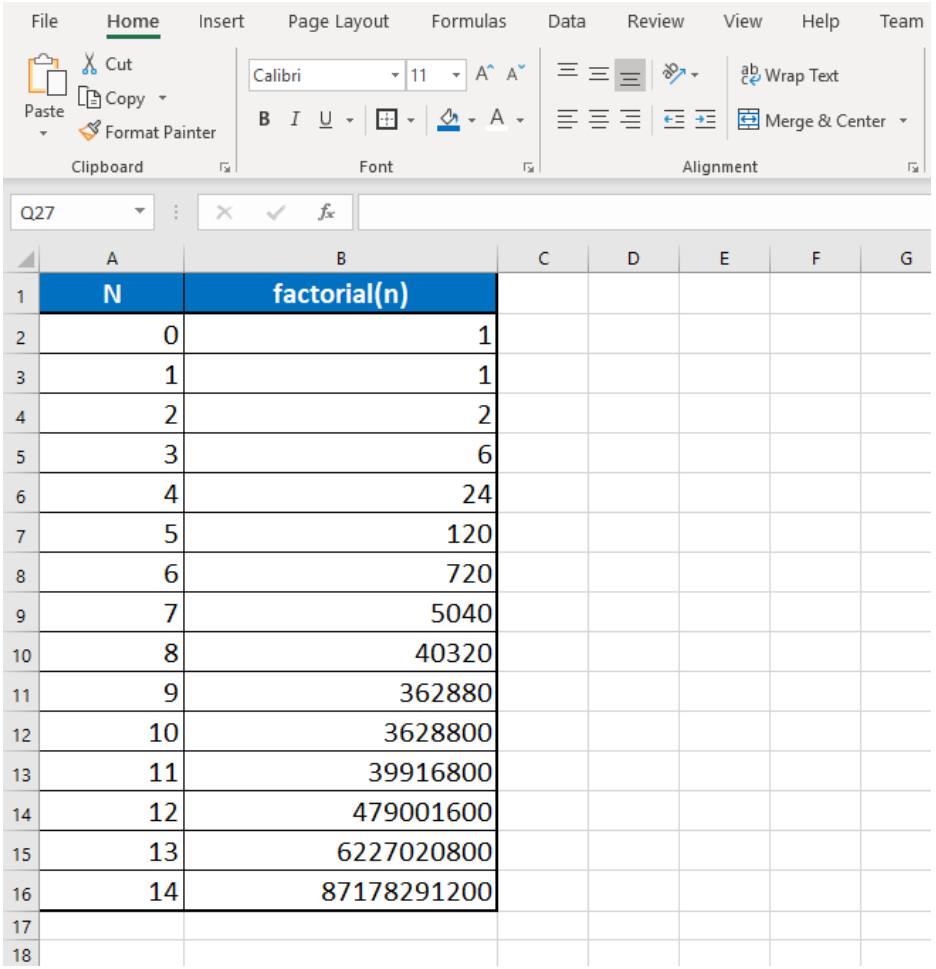


	A	B	C	D	E	F	G
1	N	factorial(n)					
2	0	1					
3	1	1					
4	2	2					
5	3	6					
6	4	24					
7	5	120					
8	6	720					
9	7	5040					
10	8	40320					
11	9	362880					
12	10	3628800					
13	11	39916800					
14	12	479001600					
15	13	6227020800					
16	14	87178291200					
17							
18							

Just too hard!

- 2 separate implementations
  - hard to keep in sync
- Template *metaprogramming* is hard!
- Easier to precalculate in Excel and hardcode results in the code ;-)

```
const int precalc_values[] = {
  1,
  1,
  2,
  6,
  24,
  120,
  720,
  // ...
};
```



	A	B	C	D	E	F	G
1	N	factorial(n)					
2	0	1					
3	1	1					
4	2	2					
5	3	6					
6	4	24					
7	5	120					
8	6	720					
9	7	5040					
10	8	40320					
11	9	362880					
12	10	3628800					
13	11	39916800					
14	12	479001600					
15	13	6227020800					
16	14	87178291200					
17							
18							

`constexpr` specifier

Declares that it is **possible to evaluate** the value of the function or variable **at compile time**

`constexpr` specifier

Declares that it is **possible to evaluate** the value of the function or variable **at compile time**

- **`constexpr`** specifier used in an *object declaration implies* ***`const`***
- **`constexpr`** specifier used in a *function* or *static member variable declaration implies* ***`inline`***

`constexpr` specifier

Declares that it is **possible to evaluate** the value of the function or variable **at compile time**

- **`constexpr`** specifier used in an *object declaration implies* ***`const`***
- **`constexpr`** specifier used in a *function* or *static member variable declaration implies* ***`inline`***
- Such variables and functions *can be used in compile time constant expressions* (provided that appropriate function arguments are given)

```
constexpr int factorial(int n);
```

```
std::array<int, factorial(4)> array;
```

Compile-time calculations (C++11)

- A **constexpr** function must not be virtual
- The function body may contain only
  - **static_assert** declarations
  - alias declarations
  - using declarations and directives
  - exactly one return statement with an expression that does not mutate any of the objects

```
constexpr int factorial(int n)
{
  return n <= 1 ? 1 : (n * factorial(n - 1));
}
```

```
std::array<int, factorial(4)> array;
```

```
constexpr std::array<int, 3> precalc_values = {
  factorial(0),
  factorial(1),
  factorial(2)
};
```

```
static_assert(factorial(4) == 24, "Error");
foo(factorial(4));  // maybe compile-time
foo(factorial(n));  // not compile-time
```

Compile-time calculations (C++14)

- A **constexpr** function must not be virtual
- The function body must not contain
  - a **goto** statement and labels other than **case** and **default**
  - a **try** block
  - an **asm** declaration
  - variable of *non-literal type*
  - variable of *static* or *thread storage duration*
  - *uninitialized* variable
  - evaluation of a *lambda expression*
  - **reinterpret_cast** and **dynamic_cast**
  - *changing an active member* of an union
  - a **new** or **delete** expression

```
constexpr int factorial(int n)
{
  int result = n;
  while(n > 1)
    result *= --n;
  return result;
}
```

```
std::array<int, factorial(4)> array;
```

```
constexpr std::array<int, 3> precalc_values = {
  factorial(0),
  factorial(1),
  factorial(2)
};
```

```
static_assert(factorial(4) == 24, "Error");
foo(factorial(4));  // maybe compile-time
foo(factorial(n));  // not compile-time
```

Compile-time calculations (C++17)

- A **constexpr** function must not be virtual
- The function body must not contain
  - a **goto** statement and labels other than **case** and **default**
  - a **try** block
  - an **asm** declaration
  - variable of *non-literal type*
  - variable of *static* or *thread storage duration*
  - *uninitialized* variable
  - **reinterpret_cast** and **dynamic_cast**
  - *changing an active member* of an union
  - a **new** or **delete** expression

```
constexpr int factorial(int n)
{
  int result = n;
  while(n > 1)
    result *= --n;
  return result;
}
```

```
std::array<int, factorial(4)> array;
```

```
constexpr std::array precalc_values = {
  factorial(0),
  factorial(1),
  factorial(2)
};
```

```
static_assert(factorial(4) == 24);
foo(factorial(4));  // maybe compile-time
foo(factorial(n));  // not compile-time
```

Compile-time calculations (C++20)

- The function body **must not contain**
  - a **goto** statement and labels other than **case** and **default**
  - variable of *non-literal type*
  - variable of *static* or *thread storage duration*
  - *uninitialized* variable
  - **reinterpret_cast**

```
constexpr int factorial(int n)
{
  int result = n;
  while(n > 1)
    result *= --n;
  return result;
}
```

```
std::array<int, factorial(4)> array;
```

```
constexpr std::array precalc_values = {
  factorial(0),
  factorial(1),
  factorial(2)
};
```

```
static_assert(factorial(4) == 24);
foo(factorial(4));  // maybe compile-time
foo(factorial(n));  // not compile-time
```

Compile-time calculations (C++20)

- The **`consteval`** specifier declares a function to be an immediate function
- Every call must produce a compile time constant expression
- An immediate function *is a* ***`constexpr`*** *function*

```
consteval int factorial(int n)
{
  int result = n;
  while(n > 1)
    result *= --n;
  return result;
}
```

```
std::array<int, factorial(4)> array;
```

```
constexpr std::array precalc_values = {
  factorial(0),
  factorial(1),
  factorial(2)
};
```

```
static_assert(factorial(4) == 24);
foo(factorial(4));  // compile-time
// foo(factorial(n));  // compile-time error
```

Compile-time dispatch (C++14)

```cpp
template<typename T>
struct is_array : std::false_type {};

template<typename T>
struct is_array<T[]> : std::true_type {};

template<typename T>
constexpr bool is_array_v = is_array<T>::value;

static_assert(is_array_v<int> == false);
static_assert(is_array_v<int[]>);
```

Compile-time dispatch (C++14)

```cpp
template<typename T>
struct is_array : std::false_type {};

template<typename T>
struct is_array<T[]> : std::true_type {};

template<typename T>
constexpr bool is_array_v = is_array<T>::value;

static_assert(is_array_v<int> == false);
static_assert(is_array_v<int[]>);
```

```cpp
template<typename T>
class smart_ptr {
  void destroy(std::true_type) noexcept
  { delete[] ptr_; }

  void destroy(std::false_type) noexcept
  { delete ptr_; }

  void destroy() noexcept
  { destroy(is_array<T>()); }

  // ...
};
```

Tag dispatch provides the possibility to select the proper function overload in compile-time based on properties of a type

Compile-time dispatch (C++17)

```
template<typename T>
inline constexpr bool is_array = false;

template<typename T>
inline constexpr bool is_array<T[]> = true;

static_assert(is_array<int> == false);
static_assert(is_array<int[]>);
```

Compile-time dispatch (C++17)

```cpp
template<typename T>
inline constexpr bool is_array = false;

template<typename T>
inline constexpr bool is_array<T[]> = true;

static_assert(is_array<int> == false);
static_assert(is_array<int[]>);
```

```cpp
template<typename T>
class smart_ptr {
  void destroy() noexcept
  {
    if constexpr(is_array<T>)
      delete[] ptr_;
    else
      delete ptr_;
  }

  // ...
};
```

Usage of variable templates and **`if constexpr`** decrease compilation time

Type traits: A negative overhead abstraction

```
struct X {
  int a, b, c;
  int id;
};
```

```
void foo(const std::vector<X>& a1, std::vector<X>& a2)
{
  memcpy(a2.data(), a1.data(), a1.size() * sizeof(X));
}
```

Type traits: A negative overhead abstraction

```
struct X {
  int a, b, c;
  int id;
};
```

```
void foo(const std::vector<X>& a1, std::vector<X>& a2)
{
  memcpy(a2.data(), a1.data(), a1.size() * sizeof(X));
}
```

Type traits: A negative overhead abstraction

```
struct X {
  int a, b, c;
  int id;
};
```

```
struct X {
  int a, b, c;
  std::string id;
};
```

```
void foo(const std::vector<X>& a1, std::vector<X>& a2)
{
  memcpy(a2.data(), a1.data(), a1.size() * sizeof(X));
}
```

Type traits: A negative overhead abstraction

```
struct X {
  int a, b, c;
  int id;
};
```

```
struct X {
  int a, b, c;
  std::string id;
};
```

```
void foo(const std::vector<X>& a1, std::vector<X>& a2)
{
  memcpy(a2.data(), a1.data(), a1.size() * sizeof(X));
}
```

Ooops!!!

Type traits: A negative overhead abstraction

```
struct X {
  int a, b, c;
  int id;
};
```

```
struct X {
  int a, b, c;
  std::string id;
};
```

```
void foo(const std::vector<X>& a1, std::vector<X>& a2)
{
  std::copy(begin(a1), end(a1), begin(a2));
}
```

Type traits: A negative overhead abstraction

```
struct X {
  int a, b, c;
  int id;
};
```

```
struct X {
  int a, b, c;
  std::string id;
};
```

```
void foo(const std::vector<X>& a1, std::vector<X>& a2)
{
  std::copy(begin(a1), end(a1), begin(a2));
}
```

```
movq    %rsi, %rax
movq    8(%rdi), %rdx
movq    (%rdi), %rsi
cmpq    %rsi, %rdx
je      .L1
movq    (%rax), %rdi
subq    %rsi, %rdx
jmp     memmove
```

```
// 100 lines of assembly code
```

Type traits: Compile-time contract

```
struct X {
  int a, b, c;
  int id;
};
```

```
struct X {
  int a, b, c;
  std::string id;
};
```

Type traits: Compile-time contract

```
struct X {
  int a, b, c;
  int id;
};
```

```
static_assert(std::is_trivially_copyable_v<X>);
```

```
struct X {
  int a, b, c;
  std::string id;
};
```

```
static_assert(std::is_trivially_copyable_v<X>);
```

Type traits: Compile-time contract

```
struct X {
  int a, b, c;
  int id;
};
```

```
static_assert(std::is_trivially_copyable_v<X>);
```

```
Compiler returned: 0
```

```
struct X {
  int a, b, c;
  std::string id;
};
```

```
static_assert(std::is_trivially_copyable_v<X>);
```

```
<source>:9:20: error: static assertion failed
    9 | static_assert(std::is_trivially_copyable_v<X>);
      |               ~~~~~^~~~~~~~~~~~~~~~~~~~~~~~~~
Compiler returned: 1
```

Type traits: Compile-time contract

```cpp
struct X {
  int a, b, c;
  int id;
};
```

```cpp
static_assert(std::is_trivially_copyable_v<X>);
```

```
Compiler returned: 0
```

```cpp
struct X {
  int a, b, c;
  std::string id;
};
```

```cpp
static_assert(std::is_trivially_copyable_v<X>);
```

```
<source>:9:20: error: static assertion failed
    9 | static_assert(std::is_trivially_copyable_v<X>);
      |               ~~~~~^~~~~~~~~~~~~~~~~~~~~~~~~~
Compiler returned: 1
```

Enforce compile-time contracts and class invariants with contracts or **`static_asserts`**

Type traits: Compile-time branching

```
enum class side { BID, ASK };

class order_book {
  template<side S>
  class book_side {

    std::vector<price> levels_;
  public:
    void insert(order o)
    {




    }
    bool match(price p) const
    {

    }
    // ...
  };

  book_side<side::BID> bids_;
  book_side<side::ASK> asks_;
  // ...
};
```

Type traits: Compile-time branching

```
enum class side { BID, ASK };

class order_book {
  template<side S>
  class book_side {
    using compare = std::conditional_t<S == side::BID, std::greater<>, std::less<>>;
    std::vector<price> levels_;
  public:
    void insert(order o)
    {




    }
    bool match(price p) const
    {

    }
    // ...
  };

  book_side<side::BID> bids_;
  book_side<side::ASK> asks_;
  // ...
};
```

Type traits: Compile-time branching

```
enum class side { BID, ASK };

class order_book {
  template<side S>
  class book_side {
    using compare = std::conditional_t<S == side::BID, std::greater<>, std::less<>>;
    std::vector<price> levels_;
  public:
    void insert(order o)
    {
      const auto it = lower_bound(begin(levels_), end(levels_), o.price, compare{});
      if(it != end(levels_) && *it != o.price)
        levels_.insert(it, o.price);
      // ...
    }
    bool match(price p) const
    {

    }
    // ...
  };

  book_side<side::BID> bids_;
  book_side<side::ASK> asks_;
  // ...
};
```

Type traits: Compile-time branching

```
enum class side { BID, ASK };

class order_book {
  template<side S>
  class book_side {
    using compare = std::conditional_t<S == side::BID, std::greater<>, std::less<>>;
    std::vector<price> levels_;
  public:
    void insert(order o)
    {
      const auto it = lower_bound(begin(levels_), end(levels_), o.price, compare{});
      if(it != end(levels_) && *it != o.price)
        levels_.insert(it, o.price);
      // ...
    }
    bool match(price p) const
    {
      return compare{}(levels_.back(), p);
    }
    // ...
  };

  book_side<side::BID> bids_;
  book_side<side::ASK> asks_;
  // ...
};
```

What is wrong here?

```
constexpr int array_size = 10'000;
int array[array_size][array_size];

for(auto i = 0L; i < array_size; ++i) {
  for(auto j = 0L; j < array_size; ++j) {
    array[j][i] = i + j;
  }
}
```

What is wrong here?

```
constexpr int array_size = 10'000;
int array[array_size][array_size];

for(auto i = 0L; i < array_size; ++i) {
  for(auto j = 0L; j < array_size; ++j) {
    array[j][i] = i + j;
  }
}
```

Reckless cache usage can cost you a lot of performance!

Quiz: How much slower is the bad case?

```
Architecture:                    x86_64
CPU op-mode(s):                  32-bit, 64-bit
Byte Order:                      Little Endian
Address sizes:                   39 bits physical, 48 bits virtual
CPU(s):                          8
On-line CPU(s) list:             0-7
Thread(s) per core:              2
Core(s) per socket:              4
Socket(s):                       1
NUMA node(s):                    1
Vendor ID:                       GenuineIntel
CPU family:                      6
Model:                           142
Model name:                      Intel(R) Core(TM) i5-8350U CPU @ 1.70GHz
Stepping:                        10
CPU MHz:                         1700.035
CPU max MHz:                     1700,0000
CPU min MHz:                     400,0000
BogoMIPS:                        3799.90
Virtualization:                  VT-x
L1d cache:                       128 KiB
L1i cache:                       128 KiB
L2 cache:                        1 MiB
L3 cache:                        6 MiB
NUMA node0 CPU(s):               0-7
```

Quiz: How much slower is the bad case?

Please everybody stand up :-)

Quiz: How much slower is the bad case?

Please everybody stand up :-)

- Similar

Quiz: How much slower is the bad case?

Please everybody stand up :-)

- Similar
- **< 2x**

Quiz: How much slower is the bad case?

Please everybody stand up :-)

- Similar
- **< 2x**
- **< 5x**

Quiz: How much slower is the bad case?

Please everybody stand up :-)

- Similar
- **< 2x**
- **< 5x**
- **< 10x**

Quiz: How much slower is the bad case?

Please everybody stand up :-)

- Similar
- **< 2x**
- **< 5x**
- **< 10x**
- **< 20x**

Quiz: How much slower is the bad case?

Please everybody stand up :-)

- Similar
- **< 2x**
- **< 5x**
- **< 10x**
- **< 20x**
- **< 50x**

Quiz: How much slower is the bad case?

Please everybody stand up :-)

- Similar
- **< 2x**
- **< 5x**
- **< 10x**
- **< 20x**
- **< 50x**
- **< 100x**

Quiz: How much slower is the bad case?

Please everybody stand up :-)

- Similar
- **< 2x**
- **< 5x**
- **< 10x**
- **< 20x**
- **< 50x**
- **< 100x**
- **>= 100x**

Quiz: How much slower is the bad case?

Please everybody stand up :-)

- Similar
- **< 2x**
- **< 5x**
- **< 10x**
- **< 20x**
- **< 50x**
- **< 100x**
- **>= 100x**

GCC 7.4.0

- column-major: 2490 ms
- row-major: 51 ms
- ratio: 47x

Quiz: How much slower is the bad case?

Please everybody stand up :-)

- Similar
- **< 2x**
- **< 5x**
- **< 10x**
- **< 20x**
- **< 50x**
- **< 100x**
- **>= 100x**

GCC 7.4.0

- column-major: **2490 ms**
- row-major: **51 ms**
- **ratio: 47x**

GCC 9.2.1

- column-major: **51 ms**
- row-major: **50 ms**
- **ratio: Similar :-)**

CPU data cache

```
     3 274,28 msec task-clock                #    0,999 CPUs utilized
 5 536 529 085      cycles                    #    1,691 GHz                      (66,53%)
 1 246 414 976      instructions              #    0,23  insn per cycle           (83,27%)
    92 872 234      L1-dcache-loads           #   28,364 M/sec                    (83,29%)
   410 636 090      L1-dcache-load-misses     #  442,15% of all L1-dcache hits    (83,39%)
    87 191 333      LLC-loads                 #   26,629 M/sec                    (83,39%)
       527 256      LLC-load-misses           #    0,60% of all LL-cache hits    (83,41%)
        97 781      page-faults               #    0,030 M/sec
```

```
      360,48 msec task-clock                #    0,999 CPUs utilized
   609 105 177      cycles                    #    1,690 GHz                      (66,71%)
   891 120 837      instructions              #    1,46  insn per cycle           (83,36%)
    90 219 862      L1-dcache-loads           #  250,279 M/sec                    (83,35%)
    14 481 217      L1-dcache-load-misses     #   16,05% of all L1-dcache hits    (83,35%)
       167 850      LLC-loads                 #    0,466 M/sec                    (83,36%)
        79 492      LLC-load-misses           #   47,36% of all LL-cache hits    (83,22%)
        97 783      page-faults               #    0,271 M/sec
```

CPU data cache

```
     3 274,28 msec task-clock                #    0,999 CPUs utilized
5 536 529 085      cycles                    #    1,691 GHz                      (66,53%)
1 246 414 976      instructions              #    0,23  insn per cycle           (83,27%)
   92 872 234      L1-dcache-loads           #   28,364 M/sec                    (83,29%)
  410 636 090      L1-dcache-load-misses     #  442,15% of all L1-dcache hits    (83,39%)
   87 191 333      LLC-loads                 #   26,629 M/sec                    (83,39%)
      527 256      LLC-load-misses           #    0,60% of all LL-cache hits    (83,41%)
       97 781      page-faults               #    0,030 M/sec
```

```
      360,48 msec task-clock                #    0,999 CPUs utilized
  609 105 177      cycles                    #    1,690 GHz                      (66,71%)
  891 120 837      instructions              #    1,46  insn per cycle           (83,36%)
   90 219 862      L1-dcache-loads           #  250,279 M/sec                    (83,35%)
   14 481 217      L1-dcache-load-misses     #   16,05% of all L1-dcache hits    (83,35%)
      167 850      LLC-loads                 #    0,466 M/sec                    (83,36%)
       79 492      LLC-load-misses           #   47,36% of all LL-cache hits    (83,22%)
       97 783      page-faults               #    0,271 M/sec
```

CPU data cache

```
     3 274,28 msec task-clock                #    0,999 CPUs utilized
 5 536 529 085      cycles                    #    1,691 GHz                      (66,53%)
 1 246 414 976      instructions              #    0,23  insn per cycle           (83,27%)
    92 872 234      L1-dcache-loads           #   28,364 M/sec                    (83,29%)
   410 636 090      L1-dcache-load-misses     #  442,15% of all L1-dcache hits    (83,39%)
    87 191 333      LLC-loads                 #   26,629 M/sec                    (83,39%)
       527 256      LLC-load-misses           #    0,60% of all LL-cache hits    (83,41%)
        97 781      page-faults               #    0,030 M/sec
```

```
      360,48 msec task-clock                #    0,999 CPUs utilized
   609 105 177      cycles                    #    1,690 GHz                      (66,71%)
   891 120 837      instructions              #    1,46  insn per cycle           (83,36%)
    90 219 862      L1-dcache-loads           #  250,279 M/sec                    (83,35%)
    14 481 217      L1-dcache-load-misses     #   16,05% of all L1-dcache hits    (83,35%)
       167 850      LLC-loads                 #    0,466 M/sec                    (83,36%)
        79 492      LLC-load-misses           #   47,36% of all LL-cache hits    (83,22%)
        97 783      page-faults               #    0,271 M/sec
```

CPU data cache

```
     3 274,28 msec task-clock                #    0,999 CPUs utilized
5 536 529 085      cycles                    #    1,691 GHz                      (66,53%)
1 246 414 976      instructions              #    0,23  insn per cycle           (83,27%)
   92 872 234      L1-dcache-loads           #   28,364 M/sec                    (83,29%)
  410 636 090      L1-dcache-load-misses     #  442,15% of all L1-dcache hits    (83,39%)
   87 191 333      LLC-loads                 #   26,629 M/sec                    (83,39%)
      527 256      LLC-load-misses           #    0,60% of all LL-cache hits    (83,41%)
       97 781      page-faults               #    0,030 M/sec
```

```
      360,48 msec task-clock                #    0,999 CPUs utilized
  609 105 177      cycles                    #    1,690 GHz                      (66,71%)
  891 120 837      instructions              #    1,46  insn per cycle           (83,36%)
   90 219 862      L1-dcache-loads           #  250,279 M/sec                    (83,35%)
   14 481 217      L1-dcache-load-misses     #   16,05% of all L1-dcache hits    (83,35%)
      167 850      LLC-loads                 #    0,466 M/sec                    (83,36%)
       79 492      LLC-load-misses           #   47,36% of all LL-cache hits    (83,22%)
       97 783      page-faults               #    0,271 M/sec
```

CPU data cache

```
     3 274,28 msec task-clock                #    0,999 CPUs utilized
 5 536 529 085      cycles                    #    1,691 GHz                      (66,53%)
 1 246 414 976      instructions              #    0,23  insn per cycle           (83,27%)
    92 872 234      L1-dcache-loads           #   28,364 M/sec                    (83,29%)
   410 636 090      L1-dcache-load-misses     #  442,15% of all L1-dcache hits    (83,39%)
    87 191 333      LLC-loads                 #   26,629 M/sec                    (83,39%)
       527 256      LLC-load-misses           #    0,60% of all LL-cache hits    (83,41%)
        97 781      page-faults               #    0,030 M/sec
```

```
      360,48 msec task-clock                #    0,999 CPUs utilized
   609 105 177      cycles                    #    1,690 GHz                      (66,71%)
   891 120 837      instructions              #    1,46  insn per cycle           (83,36%)
    90 219 862      L1-dcache-loads           #  250,279 M/sec                    (83,35%)
    14 481 217      L1-dcache-load-misses     #   16,05% of all L1-dcache hits    (83,35%)
       167 850      LLC-loads                 #    0,466 M/sec                    (83,36%)
        79 492      LLC-load-misses           #   47,36% of all LL-cache hits    (83,22%)
        97 783      page-faults               #    0,271 M/sec
```

Another example

```
struct coordinates { int x, y; };

void draw(const coordinates& coord);
void verify(int threshold);

constexpr int OBJECT_COUNT = 1'000'000;
class objectMgr;

void process(const objectMgr& mgr)
{
  const auto size = mgr.size();
  for(auto i = 0UL; i < size; ++i) { draw(mgr.position(i)); }
  for(auto i = 0UL; i < size; ++i) { verify(mgr.threshold(i)); }
}
```

Naive objectMgr implementation

```
class objectMgr {
  struct object {
    coordinates coord;
    std::string errorTxt_1;
    std::string errorTxt_2;
    std::string errorTxt_3;
    int threshold;
    std::array<char, 100> otherData;
  };
  std::vector<object> data_;

public:
  explicit objectMgr(std::size_t size) : data_{size} {}
  std::size_t size() const                              { return data_.size(); }
  const coordinates& position(std::size_t idx) const { return data_[idx].coord; }
  int threshold(std::size_t idx) const                  { return data_[idx].threshold; }
};
```

Naive objectMgr implementation

```
class objectMgr {
  struct object {
    coordinates coord;
    std::string errorTxt_1;
    std::string errorTxt_2;
    std::string errorTxt_3;
    int threshold;
    std::array<char, 100> otherData;
  };
  std::vector<object> data_;

public:
  explicit objectMgr(std::size_t size) : data_{size} {}
  std::size_t size() const                              { return data_.size(); }
  const coordinates& position(std::size_t idx) const { return data_[idx].coord; }
  int threshold(std::size_t idx) const                  { return data_[idx].threshold; }
};
```

DOD (Data-Oriented Design)

- Program optimization approach motivated by cache coherency
- Focuses on data layout
- Results in objects decomposition

DOD (Data-Oriented Design)

- Program optimization approach motivated by cache coherency
- **Focuses on data layout**
- Results in objects decomposition

Keep data used together close to each other

Object decomposition example

```
class objectMgr {
  std::vector<coordinates> positions_;
  std::vector<int> thresholds_;

  struct otherData {
    struct errorData { std::string errorTxt_1, errorTxt_2, errorTxt_3; };
    errorData error;
    std::array<char, 100> data;
  };
  std::vector<otherData> coldData_;

public:
  explicit objectMgr(std::size_t size) :
    positions_{size}, thresholds_(size), coldData_{size} {}
  std::size_t size() const                                { return positions_.size(); }
  const coordinates& position(std::size_t idx) const      { return positions_[idx]; }
  int threshold(std::size_t idx) const                    { return thresholds_[idx]; }
};
```

Members order might be important

```
struct A {
  char c;
  short s;
  int i;
  double d;
  static double dd;
};

static_assert( sizeof(A) == ??);
static_assert(alignof(A) == ??);
```

```
struct B {
  char c;
  double d;
  short s;
  int i;
  static double dd;
};

static_assert( sizeof(B) == ??);
static_assert(alignof(B) == ??);
```

Members order might be important

```
struct A {
  char c;     // size=1, alignment=1, padding=1
  short s;    // size=2, alignment=2, padding=0
  int i;      // size=4, alignment=4, padding=0
  double d;   // size=8, alignment=8, padding=0
  static double dd;
};

static_assert( sizeof(A) == 16);
static_assert(alignof(A) == 8);
```

```
struct B {
  char c;     // size=1, alignment=1, padding=7
  double d;   // size=8, alignment=8, padding=0
  short s;    // size=2, alignment=2, padding=2
  int i;      // size=4, alignment=4, padding=0
  static double dd;
};

static_assert( sizeof(B) == 24);
static_assert(alignof(B) == 8);
```

Members order might be important

```
struct A {
  char c;     // size=1, alignment=1, padding=1
  short s;    // size=2, alignment=2, padding=0
  int i;      // size=4, alignment=4, padding=0
  double d;   // size=8, alignment=8, padding=0
  static double dd;
};

static_assert( sizeof(A) == 16);
static_assert(alignof(A) == 8);
```

```
struct B {
  char c;     // size=1, alignment=1, padding=7
  double d;   // size=8, alignment=8, padding=0
  short s;    // size=2, alignment=2, padding=2
  int i;      // size=4, alignment=4, padding=0
  static double dd;
};

static_assert( sizeof(B) == 24);
static_assert(alignof(B) == 8);
```

In order to satisfy alignment requirements of all non-static members of a class, padding may be inserted after some of its members

Be aware of alignment side effects

```
struct A {
  char c;     // size=1, alignment=1, padding=1
  short s;    // size=2, alignment=2, padding=0
  int i;      // size=4, alignment=4, padding=0
  double d;   // size=8, alignment=8, padding=0
  static double dd;
};

static_assert( sizeof(A) == 16);
static_assert(alignof(A) == 8);
```

Be aware of alignment side effects

```
struct A {
  char c;     // size=1, alignment=1, padding=1
  short s;    // size=2, alignment=2, padding=0
  int i;      // size=4, alignment=4, padding=0
  double d;   // size=8, alignment=8, padding=0
  static double dd;
};

static_assert( sizeof(A) == 16);
static_assert(alignof(A) == 8);
```

```
using opt = std::optional<A>;

static_assert( sizeof(opt) == 24);
static_assert(alignof(opt) == 8);
```

Be aware of alignment side effects

```
struct A {
  char c;     // size=1, alignment=1, padding=1
  short s;    // size=2, alignment=2, padding=0
  int i;      // size=4, alignment=4, padding=0
  double d;   // size=8, alignment=8, padding=0
  static double dd;
};

static_assert( sizeof(A) == 16);
static_assert(alignof(A) == 8);
```

```
using opt = std::optional<A>;

static_assert( sizeof(opt) == 24);
static_assert(alignof(opt) == 8);
```

```
using array = std::array<opt, 256>;

static_assert( sizeof(array) == 24 * 256);
static_assert(alignof(array) == 8);
```

Be aware of alignment side effects

```cpp
struct A {
  char c;     // size=1, alignment=1, padding=1
  short s;    // size=2, alignment=2, padding=0
  int i;      // size=4, alignment=4, padding=0
  double d;   // size=8, alignment=8, padding=0
  static double dd;
};

static_assert( sizeof(A) == 16);
static_assert(alignof(A) == 8);
```

```cpp
using opt = std::optional<A>;

static_assert( sizeof(opt) == 24);
static_assert(alignof(opt) == 8);
```

```cpp
using array = std::array<opt, 256>;

static_assert( sizeof(array) == 24 * 256);
static_assert(alignof(array) == 8);
```

Be aware of implementation details of the tools you use every day

Packing

```
struct A {
  char c;     // size=1, alignment=1, padding=1
  short s;    // size=2, alignment=2, padding=0
  int i;      // size=4, alignment=4, padding=0
  double d;   // size=8, alignment=8, padding=0
  static double dd;
};

static_assert( sizeof(A) == 16);
static_assert(alignof(A) == 8);
```

```
struct C {
  char c;
  double d;
  short s;
  static double dd;
  int i;
} __attribute__((packed));

static_assert( sizeof(C) == 15);
static_assert(alignof(C) == 1);
```

Packing

```
struct A {
  char c;     // size=1, alignment=1, padding=1
  short s;    // size=2, alignment=2, padding=0
  int i;      // size=4, alignment=4, padding=0
  double d;   // size=8, alignment=8, padding=0
  static double dd;
};

static_assert( sizeof(A) == 16);
static_assert(alignof(A) == 8);
```

```
struct C {
  char c;
  double d;
  short s;
  static double dd;
  int i;
} __attribute__((packed));

static_assert( sizeof(C) == 15);
static_assert(alignof(C) == 1);
```

On modern hardware may be faster than aligned structure

Packing

```
struct A {
  char c;     // size=1, alignment=1, padding=1
  short s;    // size=2, alignment=2, padding=0
  int i;      // size=4, alignment=4, padding=0
  double d;   // size=8, alignment=8, padding=0
  static double dd;
};

static_assert( sizeof(A) == 16);
static_assert(alignof(A) == 8);
```

```
struct C {
  char c;
  double d;
  short s;
  static double dd;
  int i;
} __attribute__((packed));

static_assert( sizeof(C) == 15);
static_assert(alignof(C) == 1);
```

On modern hardware may be faster than aligned structure

Not portable! May be slower or even crash

Latency Numbers Every Programmer Should Know

L1 cache reference	0.5	ns		
Branch misprediction	5	ns		
L2 cache reference	7	ns		14x L1 cache
Mutex lock/unlock	25	ns		
Main memory reference	100	ns		20x L2 cache, 200x L1 cache
Compress 1K bytes with Zippy	3,000	ns		
Send 1K bytes over 1 Gbps network	10,000	ns	0.01 ms	
Read 4K randomly from SSD	150,000	ns	0.15 ms	~1GB/sec SSD
Read 1 MB sequentially from memory	250,000	ns	0.25 ms	
Round trip within same datacenter	500,000	ns	0.5 ms	
Read 1 MB sequentially from SSD	1,000,000	ns	1 ms	~1GB/sec SSD, 4X memory
Disk seek	10,000,000	ns	10 ms	20x datacenter roundtrip
Read 1 MB sequentially from disk	20,000,000	ns	20 ms	80x memory, 20X SSD
Send packet CA->Netherlands->CA	150,000,000	ns	150 ms	

https://gist.github.com/jboner/2841832

What is wrong here?

```
class vector_downward {
  uint8_t *make_space(size_t len) {
    if(len > static_cast<size_t>(cur_ - buf_)) {
      auto old_size = size();
      auto largest_align = AlignOf<largest_scalar_t>();
      reserved_ += (std::max)(len, growth_policy(reserved_));
      // Round up to avoid undefined behavior from unaligned loads and stores.
      reserved_ = (reserved_ + (largest_align - 1)) & ~(largest_align - 1);
      auto new_buf = allocator_.allocate(reserved_);
      auto new_cur = new_buf + reserved_ - old_size;
      memcpy(new_cur, cur_, old_size);
      cur_ = new_cur;
      allocator_.deallocate(buf_);
      buf_ = new_buf;
    }
    cur_ -= len;
    // Beyond this, signed offsets may not have enough range:
    // (FlatBuffers > 2GB not supported).
    assert(size() < FLATBUFFERS_MAX_BUFFER_SIZE);
    return cur_;
  }
  // ...
};
```

What is wrong here?

```
class vector_downward {
  uint8_t *make_space(size_t len) {
    if(len > static_cast<size_t>(cur_ - buf_)) {
      auto old_size = size();
      auto largest_align = AlignOf<largest_scalar_t>();
      reserved_ += (std::max)(len, growth_policy(reserved_));
      // Round up to avoid undefined behavior from unaligned loads and stores.
      reserved_ = (reserved_ + (largest_align - 1)) & ~(largest_align - 1);
      auto new_buf = allocator_.allocate(reserved_);
      auto new_cur = new_buf + reserved_ - old_size;
      memcpy(new_cur, cur_, old_size);
      cur_ = new_cur;
      allocator_.deallocate(buf_);
      buf_ = new_buf;
    }
    cur_ -= len;
    // Beyond this, signed offsets may not have enough range:
    // (FlatBuffers > 2GB not supported).
    assert(size() < FLATBUFFERS_MAX_BUFFER_SIZE);
    return cur_;
  }
  // ...
};
```

Separate the code to hot and cold paths

```
class vector_downward {
  uint8_t *make_space(size_t len) {
    if(len > static_cast<size_t>(cur_ - buf_))
      reallocate(len);
    cur_ -= len;
    // Beyond this, signed offsets may not have enough range:
    // (FlatBuffers > 2GB not supported).
    assert(size() < FLATBUFFERS_MAX_BUFFER_SIZE);
    return cur_;
  }

  void reallocate(size_t len) {
    auto old_size = size();
    auto largest_align = AlignOf<largest_scalar_t>();
    reserved_ += (std::max)(len, growth_policy(reserved_));
    // Round up to avoid undefined behavior from unaligned loads and stores.
    reserved_ = (reserved_ + (largest_align - 1)) & ~(largest_align - 1);
    auto new_buf = allocator_.allocate(reserved_);
    auto new_cur = new_buf + reserved_ - old_size;
    memcpy(new_cur, cur_, old_size);
    cur_ = new_cur;
    allocator_.deallocate(buf_);
    buf_ = new_buf;
  }
  // ...
};
```

Separate the code to hot and cold paths

```
class vector_downward {
  uint8_t *make_space(size_t len) {
    if(len > static_cast<size_t>(cur_ - buf_))
      reallocate(len);
    cur_ -= len;
    // Beyond this, signed offsets may not have enough range:
    // (FlatBuffers > 2GB not supported).
    assert(size() < FLATBUFFERS_MAX_BUFFER_SIZE);
    return cur_;
  }
  // ...
};
```

Code is data too!

Separate the code to hot and cold paths

```
class vector_downward {
  uint8_t *make_space(size_t len) {
    if(len > static_cast<size_t>(cur_ - buf_))
      reallocate(len);
    cur_ -= len;
    // Beyond this, signed offsets may not have enough range:
    // (FlatBuffers > 2GB not supported).
    assert(size() < FLATBUFFERS_MAX_BUFFER_SIZE);
    return cur_;
  }
  // ...
};
```

Code is data too!

Performance improvement of 20% thanks to a better inlining

Typical validation and error handling

```
std::optional<Error> validate(const request& r)
{
  switch(r.type) {
  case request::type_1:
    if(/* simple check */)
      return std::nullopt;
    return /* complex error msg generation */;
  case request::type_2:
    if(/* simple check */)
      return std::nullopt;
    return /* complex error msg generation */;
  case request::type_3:
    if(/* simple check */)
      return std::nullopt;
    return /* complex error msg generation */;
  // ...
  }
  throw std::logic_error("");
}
```

Isolating cold path

```
std::optional<Error> validate(const request& r)
{
  if(is_valid(r))
    return std::nullopt;
  return make_error(r)
}
```

Isolating cold path

```
std::optional<Error> validate(const request& r)
{
  if(is_valid(r))
    return std::nullopt;
  return make_error(r)
}
```

```
bool is_valid(const request& r)
{
  switch(r.type) {
  case request::type_1:
    return /* simple check */;
  case request::type_2:
    return /* simple check */;
  case request::type_3:
    return /* simple check */;
  // ...
  }
  throw std::logic_error("");
}
```

Isolating cold path

```
std::optional<Error> validate(const request& r)
{
  if(is_valid(r))
    return std::nullopt;
  return make_error(r)
}
```

```
bool is_valid(const request& r)
{
  switch(r.type) {
  case request::type_1:
    return /* simple check */;
  case request::type_2:
    return /* simple check */;
  case request::type_3:
    return /* simple check */;
  // ...
  }
  throw std::logic_error("");
}
```

```
Error make_error(const request& r)
{
  switch(r.type) {
  case request::type_1:
    return /* complex error msg generation */;
  case request::type_2:
    return /* complex error msg generation */;
  case request::type_3:
    return /* complex error msg generation */;
  // ...
  }
  throw std::logic_error("");
}
```

Expression short-circuiting

WRONG

```
if(expensiveCheck() && fastCheck()) { /* ... */ }
```

GOOD

```
if(fastCheck() && expensiveCheck()) { /* ... */ }
```

Expression short-circuiting

WRONG

```
if(expensiveCheck() && fastCheck()) { /* ... */ }
```

GOOD

```
if(fastCheck() && expensiveCheck()) { /* ... */ }
```

Bail out as early as possible and continue fast path

Integer arithmetic

```
int foo(int i)
{
  int k = 0;
  for(int j = i; j < i + 10; ++j)
    ++k;
  return k;
}
```

```
int foo(unsigned i)
{
  int k = 0;
  for(unsigned j = i; j < i + 10; ++j)
    ++k;
  return k;
}
```

Integer arithmetic

```
int foo(int i)
{
  int k = 0;
  for(int j = i; j < i + 10; ++j)
    ++k;
  return k;
}
```

```
foo(int):
        mov     eax, 10
        ret
```

```
int foo(unsigned i)
{
  int k = 0;
  for(unsigned j = i; j < i + 10; ++j)
    ++k;
  return k;
}
```

```
foo(unsigned int):
        cmp     edi, -10
        sbb     eax, eax
        and     eax, 10
        ret
```

Integer arithmetic

```
int foo(int i)
{
  int k = 0;
  for(int j = i; j < i + 10; ++j)
    ++k;
  return k;
}
```

```
foo(int):
        mov     eax, 10
        ret
```

```
int foo(unsigned i)
{
  int k = 0;
  for(unsigned j = i; j < i + 10; ++j)
    ++k;
  return k;
}
```

```
foo(unsigned int):
        cmp     edi, -10
        sbb     eax, eax
        and     eax, 10
        ret
```

Integer arithmetic differs for the signed and unsigned integral types

How to develop system with low-latency constraints

- Avoid using **`std::list`**, **`std::map`**, etc

How to develop system with low-latency constraints

- Avoid using **`std::list`**, **`std::map`**, etc
- **`std::unordered_map`** is also broken

How to develop system with low-latency constraints

- Avoid using **`std::list`**, **`std::map`**, etc
- **`std::unordered_map`** is also broken
- Prefer **`std::vector`**
  - also as the underlying storage for hash tables or flat maps
  - consider using **`tsl::hopscotch_map`**, Swiss Tables, F14 hash map, or similar

How to develop system with low-latency constraints

- Avoid using **`std::list`**, **`std::map`**, etc
- **`std::unordered_map`** is also broken
- Prefer **`std::vector`**
  - also as the underlying storage for hash tables or flat maps
  - consider using **`tsl::hopscotch_map`**, Swiss Tables, F14 hash map, or similar
- Preallocate storage
  - free list
  - **`plf::colony`**

How to develop system with low-latency constraints

- Avoid using **`std::list`**, **`std::map`**, etc
- **`std::unordered_map`** is also broken
- Prefer **`std::vector`**
  - also as the underlying storage for hash tables or flat maps
  - consider using **`tsl::hopscotch_map`**, Swiss Tables, F14 hash map, or similar
- Preallocate storage
  - free list
  - **`plf::colony`**
- Consider storing only pointers to objects in the container
- Consider storing expensive hash values with the key

How to develop system with low-latency constraints

- Avoid using **`std::list`**, **`std::map`**, etc
- **`std::unordered_map`** is also broken
- Prefer **`std::vector`**
  - also as the underlying storage for hash tables or flat maps
  - consider using **`tsl::hopscotch_map`**, Swiss Tables, F14 hash map, or similar
- Preallocate storage
  - free list
  - **`plf::colony`**
- Consider storing only pointers to objects in the container
- Consider storing expensive hash values with the key
- Limit the number of *type conversions*

How to develop system with low-latency constraints

How to develop system with low-latency constraints

- Keep the **number of threads** close (less or equal) to the number of available *physical CPU cores*

How to develop system with low-latency constraints

- Keep the **number of threads** close (less or equal) to the number of available *physical CPU cores*
- **Separate** IO threads from business logic thread (unless business logic is extremely lightweight)

How to develop system with low-latency constraints

- Keep the **number of threads** close (less or equal) to the number of available *physical CPU cores*
- **Separate** IO threads from business logic thread (unless business logic is extremely lightweight)
- Use fixed size *lock free queues* / *busy spins* to pass the data between threads

How to develop system with low-latency constraints

- Keep the **number of threads** close (less or equal) to the number of available *physical CPU cores*
- **Separate** IO threads from business logic thread (unless business logic is extremely lightweight)
- Use fixed size *lock free queues* / *busy spins* to pass the data between threads
- Use optimal *algorithms*/*data structures* and *data locality* principle

How to develop system with low-latency constraints

- Keep the **number of threads** close (less or equal) to the number of available *physical CPU cores*
- **Separate** IO threads from business logic thread (unless business logic is extremely lightweight)
- Use fixed size *lock free queues* / *busy spins* to pass the data between threads
- Use optimal *algorithms*/*data structures* and *data locality* principle
- **Precompute**, use compile time instead of runtime whenever possible

How to develop system with low-latency constraints

- Keep the **number of threads** close (less or equal) to the number of available *physical CPU cores*
- **Separate** IO threads from business logic thread (unless business logic is extremely lightweight)
- Use fixed size *lock free queues* / *busy spins* to pass the data between threads
- Use optimal *algorithms*/*data structures* and *data locality* principle
- **Precompute**, use compile time instead of runtime whenever possible
- *The simpler the code, the faster it is likely to be*

How to develop system with low-latency constraints

- Keep the **number of threads** close (less or equal) to the number of available *physical CPU cores*
- **Separate** IO threads from business logic thread (unless business logic is extremely lightweight)
- Use fixed size *lock free queues* / *busy spins* to pass the data between threads
- Use optimal *algorithms*/*data structures* and *data locality* principle
- **Precompute**, use compile time instead of runtime whenever possible
- *The simpler the code, the faster it is likely to be*
- Do not try to be smarter than the compiler

How to develop system with low-latency constraints

- Keep the **number of threads** close (less or equal) to the number of available *physical CPU cores*
- **Separate** IO threads from business logic thread (unless business logic is extremely lightweight)
- Use fixed size *lock free queues* / *busy spins* to pass the data between threads
- Use optimal *algorithms*/*data structures* and *data locality* principle
- **Precompute**, use compile time instead of runtime whenever possible
- *The simpler the code, the faster it is likely to be*
- Do not try to be smarter than the compiler
- Know the *language*, *tools*, and *libraries*

How to develop system with low-latency constraints

- Keep the **number of threads** close (less or equal) to the number of available *physical CPU cores*
- **Separate** IO threads from business logic thread (unless business logic is extremely lightweight)
- Use fixed size *lock free queues* / *busy spins* to pass the data between threads
- Use optimal *algorithms*/*data structures* and *data locality* principle
- **Precompute**, use compile time instead of runtime whenever possible
- *The simpler the code, the faster it is likely to be*
- Do not try to be smarter than the compiler
- Know the *language*, *tools*, and *libraries*
- Know your *hardware*!

How to develop system with low-latency constraints

- Keep the **number of threads** close (less or equal) to the number of available *physical CPU cores*
- **Separate** IO threads from business logic thread (unless business logic is extremely lightweight)
- Use fixed size *lock free queues* / *busy spins* to pass the data between threads
- Use optimal *algorithms*/*data structures* and *data locality* principle
- **Precompute**, use compile time instead of runtime whenever possible
- *The simpler the code, the faster it is likely to be*
- Do not try to be smarter than the compiler
- Know the *language*, *tools*, and *libraries*
- Know your *hardware*!
- *Bypass the kernel* (100% user space code)

How to develop system with low-latency constraints

- Keep the **number of threads** close (less or equal) to the number of available *physical CPU cores*
- **Separate** IO threads from business logic thread (unless business logic is extremely lightweight)
- Use fixed size *lock free queues* / *busy spins* to pass the data between threads
- Use optimal *algorithms*/*data structures* and *data locality* principle
- **Precompute**, use compile time instead of runtime whenever possible
- *The simpler the code, the faster it is likely to be*
- Do not try to be smarter than the compiler
- Know the *language*, *tools*, and *libraries*
- Know your *hardware*!
- *Bypass the kernel* (100% user space code)
- **Measure** performance… **ALWAYS**

The most important recommendation

The most important recommendation

Always measure your performance!

How to measure the performance of your programs

- Always measure Release version

```
cmake -DCMAKE_BUILD_TYPE=Release ..
cmake -DCMAKE_BUILD_TYPE=RelWithDebInfo ..
```

How to measure the performance of your programs

- Always measure **Release** version

```
cmake -DCMAKE_BUILD_TYPE=Release ..
cmake -DCMAKE_BUILD_TYPE=RelWithDebInfo ..
```

- Prefer *hardware based black box* performance meassurements

How to measure the performance of your programs

- Always measure **Release** version

```
cmake -DCMAKE_BUILD_TYPE=Release ..
cmake -DCMAKE_BUILD_TYPE=RelWithDebInfo ..
```

- Prefer *hardware based black box* performance meassurements
- In case that is not possible or you want to debug specific performance issue use *profiler*
- To gather meaningful stack traces *preserve frame pointer*

```
cmake -DCMAKE_BUILD_TYPE=RelWithDebInfo -DCMAKE_CXX_FLAGS="-fno-omit-frame-pointer" ..
```

How to measure the performance of your programs

- Always measure **Release** version

```
cmake -DCMAKE_BUILD_TYPE=Release ..
cmake -DCMAKE_BUILD_TYPE=RelWithDebInfo ..
```

- Prefer *hardware based black box* performance meassurements
- In case that is not possible or you want to debug specific performance issue use *profiler*
- To gather meaningful stack traces *preserve frame pointer*

```
cmake -DCMAKE_BUILD_TYPE=RelWithDebInfo -DCMAKE_CXX_FLAGS="-fno-omit-frame-pointer" ..
```

- Familiarize yourself with linux perf tools (xperf on Windows) and flame graphs
- Use tools like Intel VTune

How to measure the performance of your programs

- Always measure **Release** version

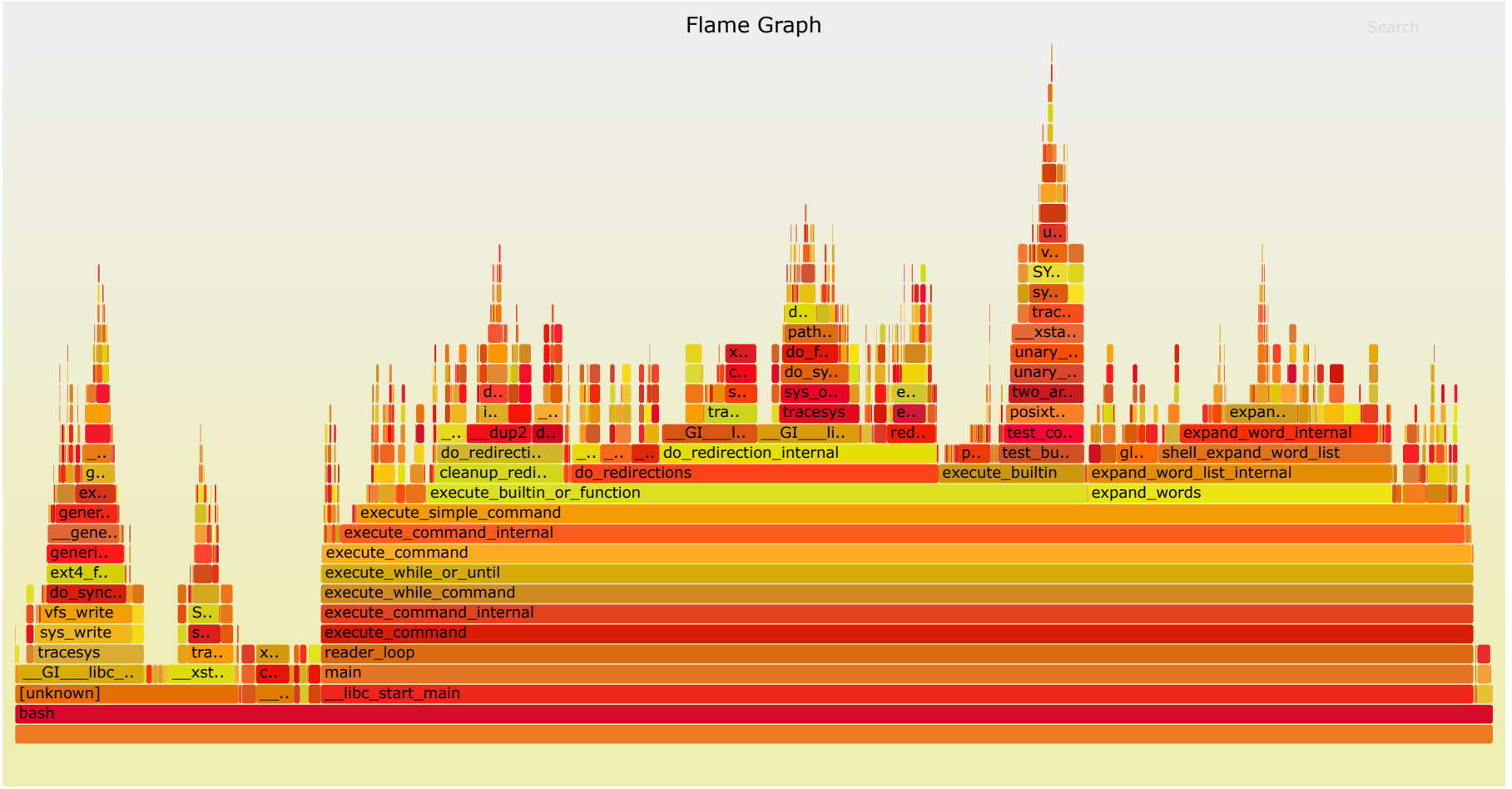
```
cmake -DCMAKE_BUILD_TYPE=Release ..
cmake -DCMAKE_BUILD_TYPE=RelWithDebInfo ..
```

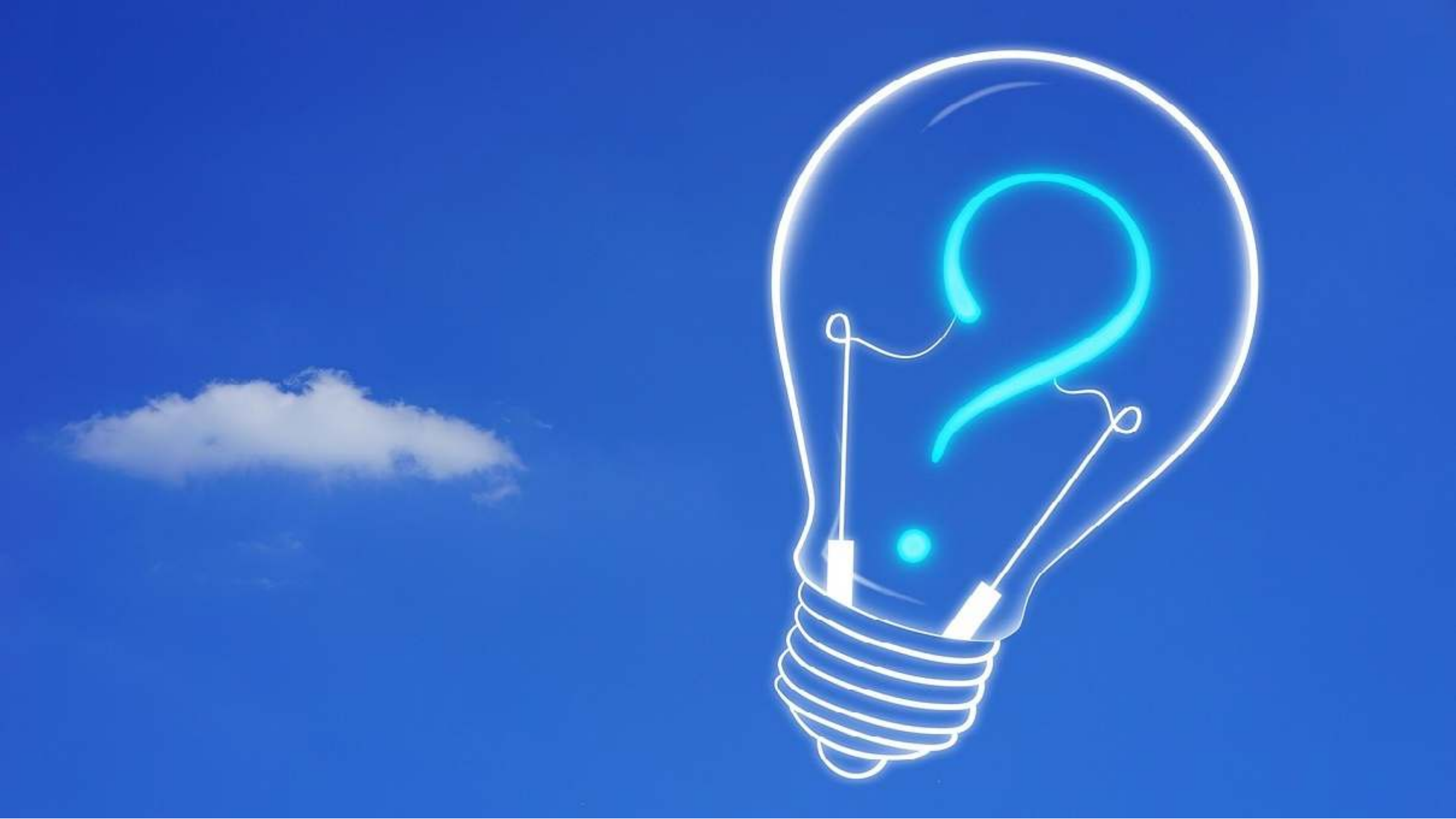
- Prefer *hardware based black box* performance meassurements
- In case that is not possible or you want to debug specific performance issue use *profiler*
- To gather meaningful stack traces *preserve frame pointer*

```
cmake -DCMAKE_BUILD_TYPE=RelWithDebInfo -DCMAKE_CXX_FLAGS="-fno-omit-frame-pointer" ..
```

- Familiarize yourself with linux perf tools (xperf on Windows) and flame graphs
- Use tools like Intel VTune
- *Verify output assembly code*

Flamegraph


Flame Graph
Search
u..
v..
SY..
sy..
trac..
__xsta..
unary_..
unary_..
two_ar..
posixt..
test_co..
test_bu..
p..
execute_builtin
expand_words
gl..
expand_word_list_internal
shell_expand_word_list
expan..
expand_word_internal
d..
path..
do_f..
do_sy..
sys_o..
tracesys
red..
e..
e..
x..
c..
s..
tra..
__GI___l..
__GI___li..
do_redirection_internal
_.. _.. _..
do_redirections
execute_builtin_or_function
execute_simple_command
execute_command_internal
execute_command
execute_while_or_until
execute_while_command
execute_command_internal
execute_command
reader_loop
main
__libc_start_main
bash
d..
i..
_..
_..
__dup2
d..
do_redirecti..
cleanup_redi..
g..
ex..
gener..
__gene..
generi..
ext4_f..
do_sync..
vfs_write
sys_write
tracesys
__GI___libc_..
[unknown]
S..
s..
tra..
__xst..
x..
c..
__..




CAUTION
Programming
is addictive
(and too much fun)