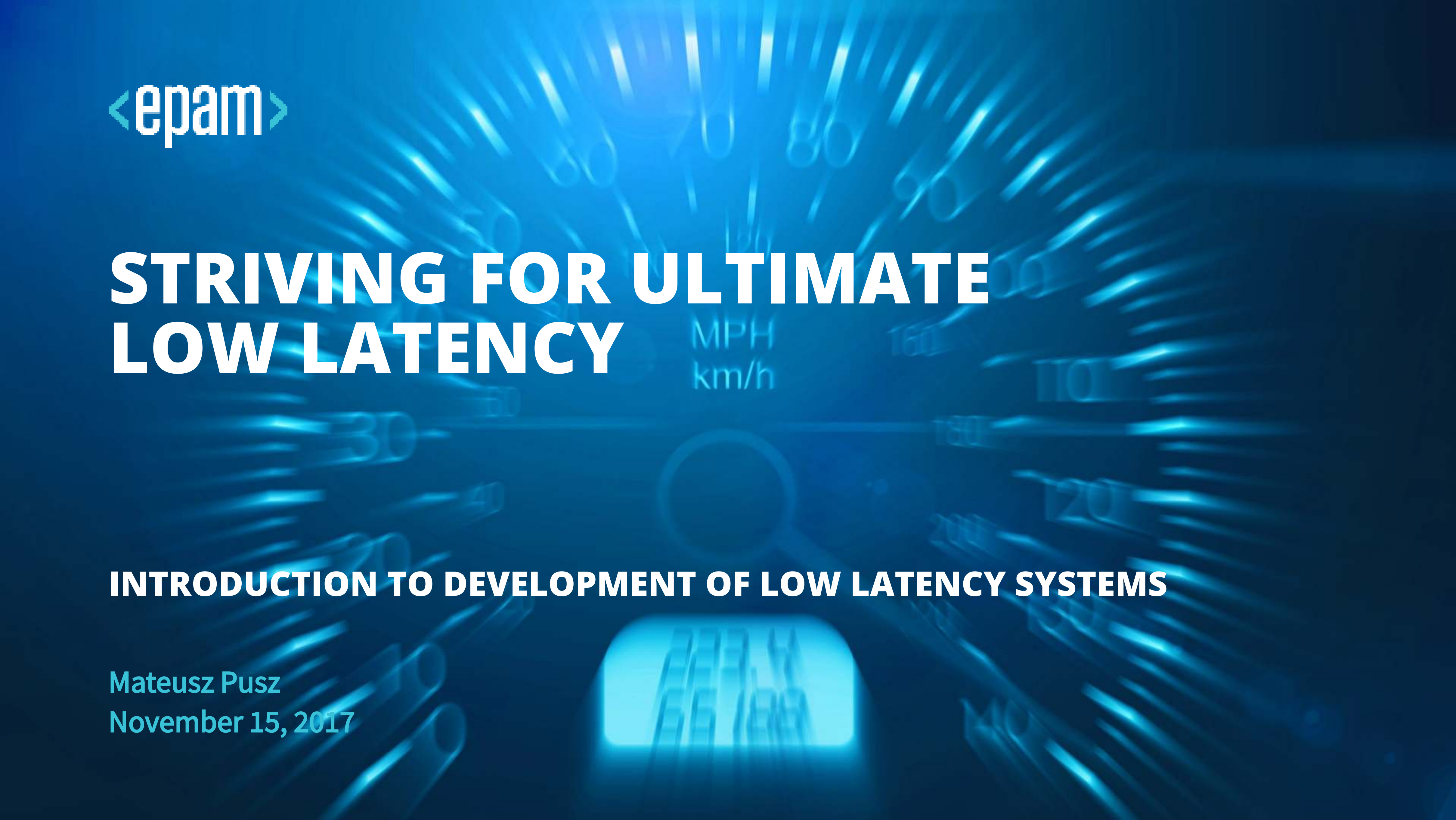


STRIVING FOR ULTIMATE LOW LATENCY

INTRODUCTION TO DEVELOPMENT OF LOW LATENCY SYSTEMS

Mateusz Pusz

November 15, 2017

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)

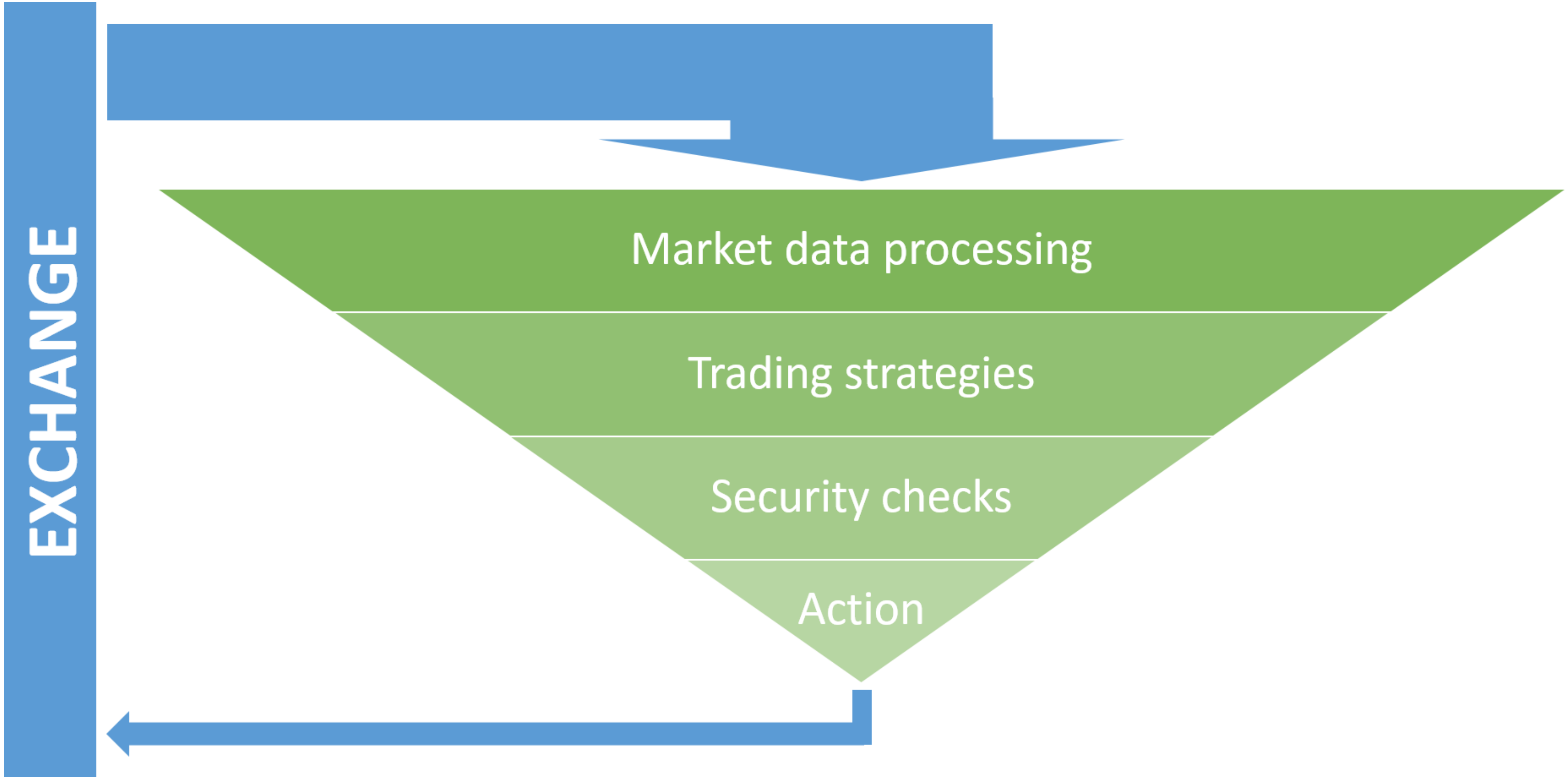
> A program trading platform that uses powerful computers to transact a large number of orders at very fast speeds
>
> -- *Investopedia*

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
- 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



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

- 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**

-- LinkedIn

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

- Low Latency network
- Modern hardware
- BIOS profiling
- Kernel profiling
- 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 (fast path)

CHARACTERISTICS OF LOW LATENCY SOFTWARE

- Typically only a small part of code is really important (fast path)
- That 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** (fast path)
- That 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
  - it is about 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** (fast path)
- That 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
  - it is about 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 costly**
  - 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
- This is not only the task for developers but also for code architects

THINGS TO AVOID ON THE FAST PATH

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

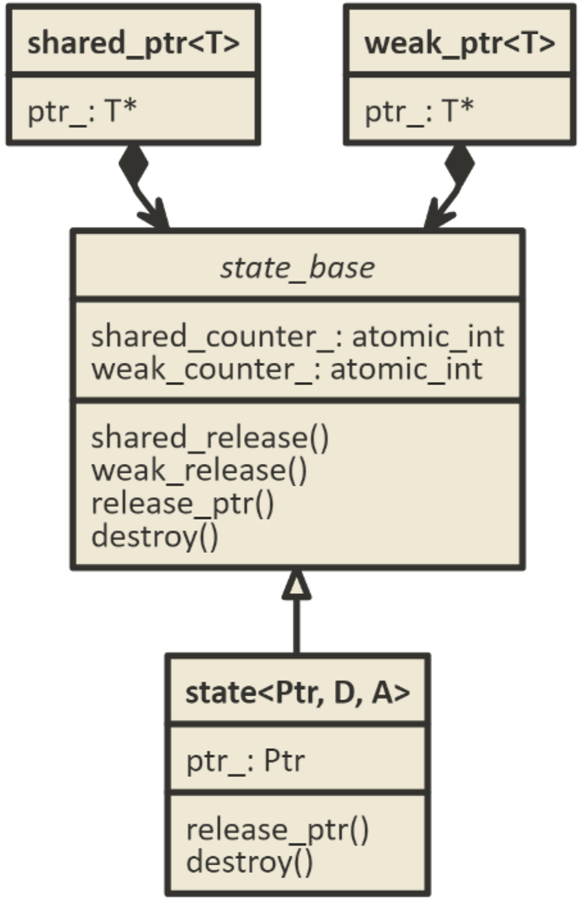
Too often overused by C++ programmers

QUESTION: WHAT IS THE DIFFERENCE HERE?

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

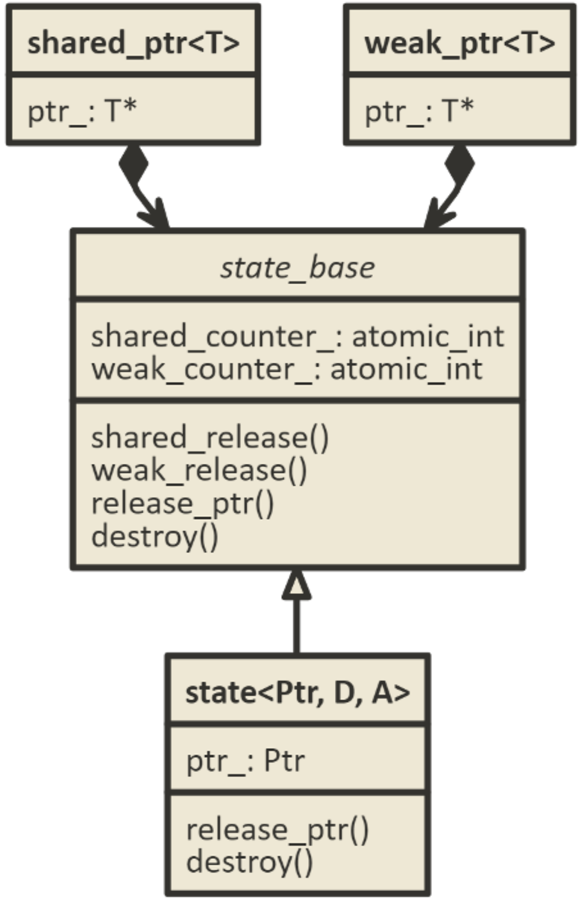
```
void foo()
{
  std::shared_ptr<int> ptr{new int{1}};
  // some code using 'ptr'
}
```

KEY `std::shared_ptr<T>` ISSUES

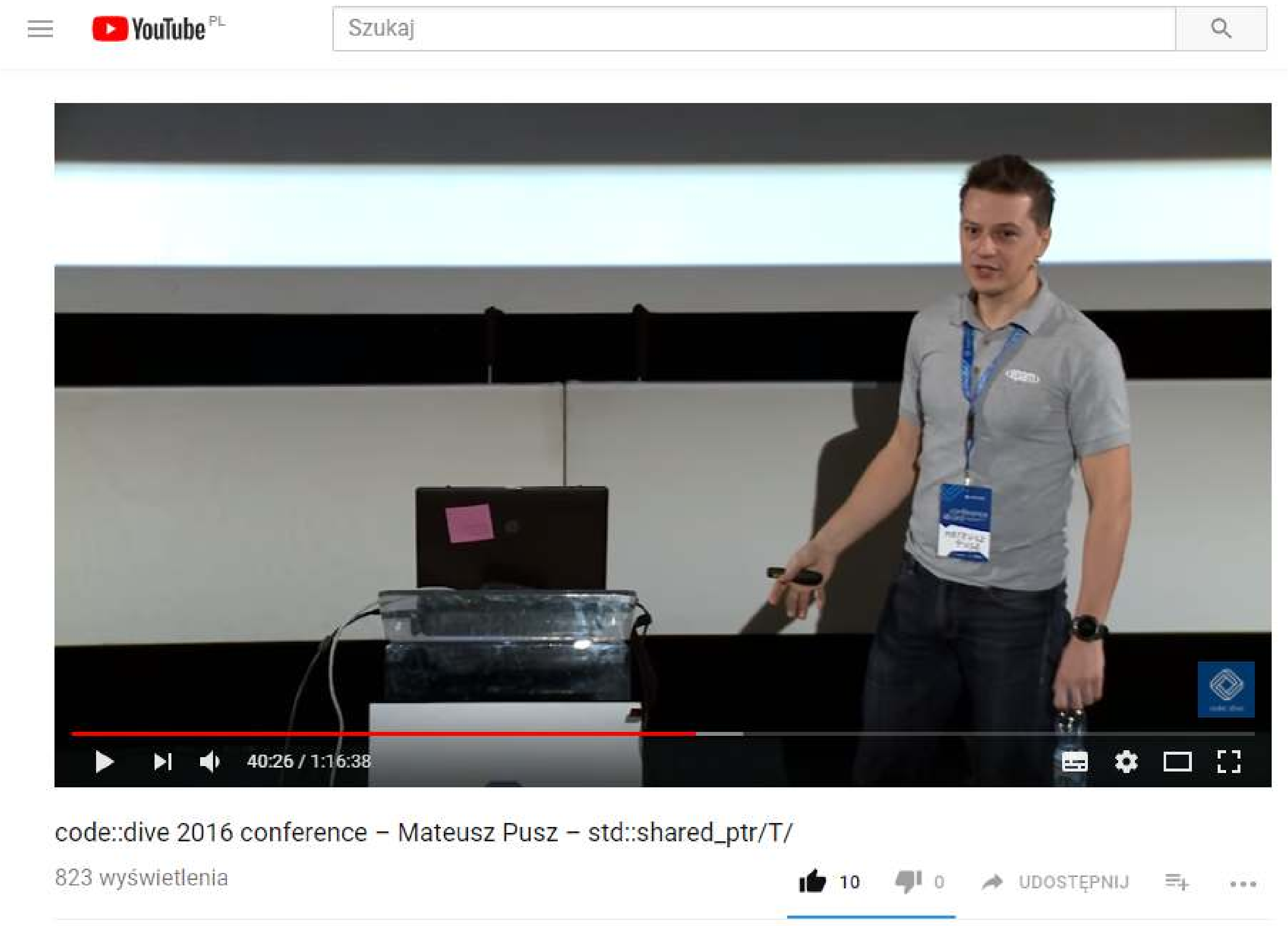


KEY `std::shared_ptr<T>` ISSUES

- 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



code::dive 2016 conference – Mateusz Pusz – std::shared_ptr/T/

823 wyświetlenia

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
  - (if not thrown) actually can improve application performance
  - cannot be ignored!
  - simplify interfaces
  - 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
  - (if not thrown) actually can improve application performance
  - cannot be ignored!
  - simplify interfaces
  - make source code of likely path easier to reason about

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

EXCEPTION SAFETY GUARANTEES

1. Nothrow (or nofail) exception guarantee
2. Strong exception guarantee
3. Basic exception guarantee
4. No exception guarantee

EXCEPTION SAFETY GUARANTEES

1. Nothrow (or nofail) exception guarantee
2. Strong exception guarantee
3. Basic exception guarantee
4. No exception guarantee

Only in case of "No exception guarantee" if the function throws an exception, the program may not be in a valid state: resource leaks, memory corruption, or other invariant-destroying errors may have occurred.

POLYMORPHISM

POLYMORPHISM

DYNAMIC

```cpp
class base {
  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

```cpp
class base {
  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

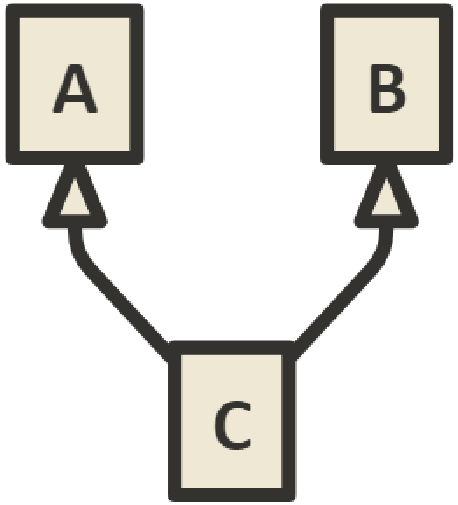
DYNAMIC

```cpp
class base {
  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

```cpp
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() { /* ... */ }
};
```

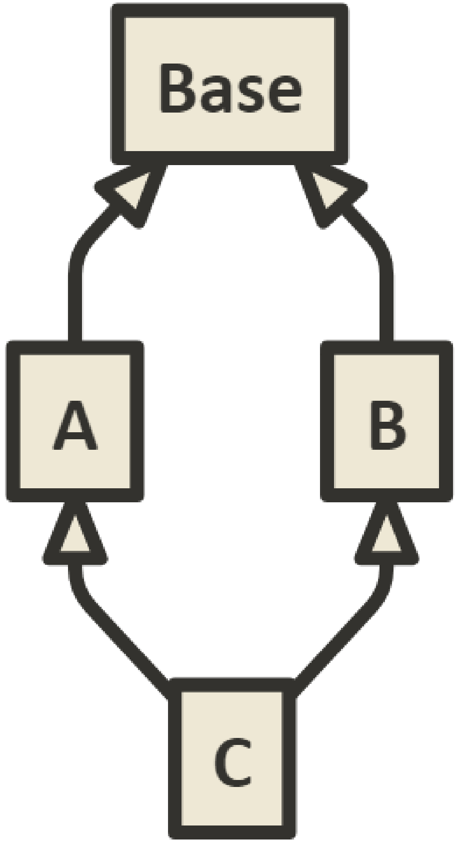
MULTIPLE INHERITANCE



MULTIPLE INHERITANCE

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

MULTIPLE INHERITANCE



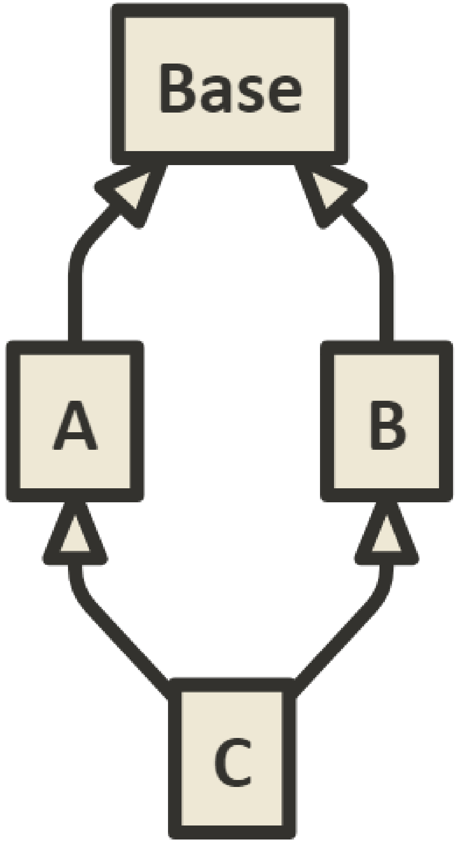
MULTIPLE INHERITANCE

- **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)

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

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

RUNTIME TYPE IDENTIFICATION (RTTI)

```
class base {
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 {
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 {
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 {
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*
- *Memory leaks* possible if not properly handled
- May *fail* (error handling is needed)

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

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

```cpp
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:
  size_t capacity() const { return data_ == u_.sso_ ? sizeof(u_.sso_) - 1 : u_.capacity_; }
  // ...
};
```

NO DYNAMIC ALLOCATION

```cpp
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

```
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 memory allocations or pointer indirections guaranteed with the cost of possibly bigger memory usage

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

EXAMPLE OF SAFE TO USE C++ TOOLS

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

DO YOU AGREE?

The fastest programs are those that do nothing

constexpr FUNCTION

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

volatile int k = 8;
std::cout << factorial(k) << '\n';    // runtime
```

constexpr FUNCTION

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

volatile int k = 8;
std::cout << factorial(k) << '\n';    // runtime
```

C++11

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

constexpr FUNCTION

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

volatile int k = 8;
std::cout << factorial(k) << '\n';    // runtime
```

C++11

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

C++14

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

constexpr FUNCTION

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

volatile int k = 8;
std::cout << factorial(k) << '\n';    // runtime
```

C++11

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

C++14

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

No need to create and use manually precalculated tables anymore

C++20 SPOILER ALERT ;-)

Support for **`constexpr`** dynamic memory allocation and deallocation added in Albuquerque, NM.

Possibility to create **`constexpr std::vector`** or maybe even **`std::string`**.

COMPILE TIME DISPATCH

```
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

```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
void destroy(std::true_type) noexcept
{
  delete[] ptr_;
}
void destroy(std::false_type) noexcept
{
  delete ptr_;
}
void destroy() noexcept
{
  destroy(is_array<T>());
}
```

COMPILE TIME DISPATCH

```
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[]>);
```

```
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.

C++17 COMPILE TIME DISPATCH

```
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[]>);
```

```
void destroy() noexcept
{
  if constexpr(is_array_v<T>)
    delete[] ptr_;
  else
    delete ptr_;
}
```

TYPE TRAITS: A NEGATIVE OVERHEAD ABSTRACTION

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

```cpp
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 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

```cpp
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

```cpp
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!

LAKOS'17 EXERCISE

LAKOS'17 EXERCISE

Please everybody stand up

QUIZ: HOW MUCH SLOWER IS THE BAD CASE?

QUIZ: HOW MUCH SLOWER IS THE BAD CASE?

- Less than 2x

QUIZ: HOW MUCH SLOWER IS THE BAD CASE?

- Less than 2x
- Less than 5x

QUIZ: HOW MUCH SLOWER IS THE BAD CASE?

- Less than 2x
- Less than 5x
- Less than 10x

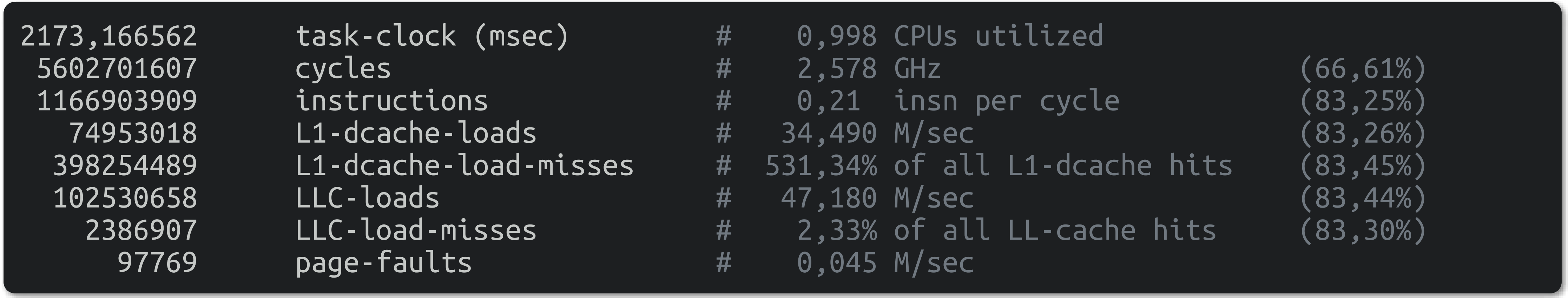
QUIZ: HOW MUCH SLOWER IS THE BAD CASE?

- Less than 2x
- Less than 5x
- Less than 10x
- Less than 20x

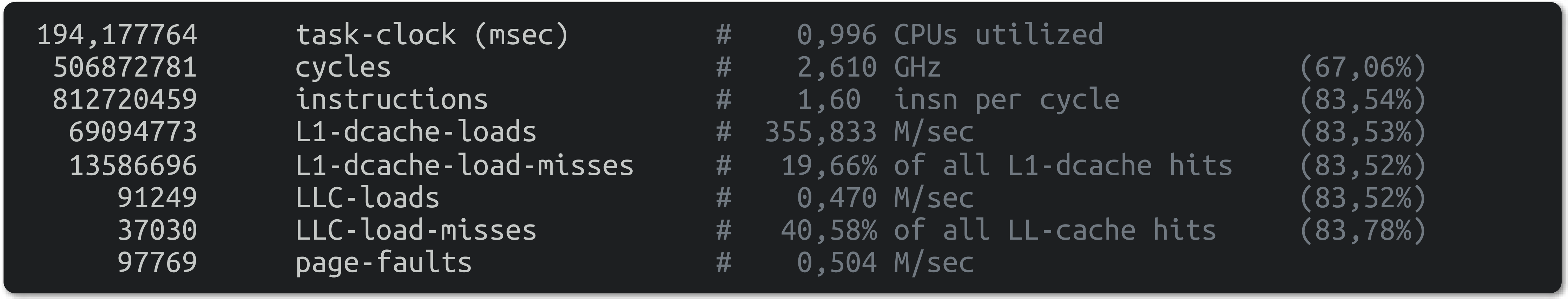
QUIZ: HOW MUCH SLOWER IS THE BAD CASE?

- Less than 2x
- Less than 5x
- Less than 10x
- Less than 20x
- Less than 50x

CPU CACHE

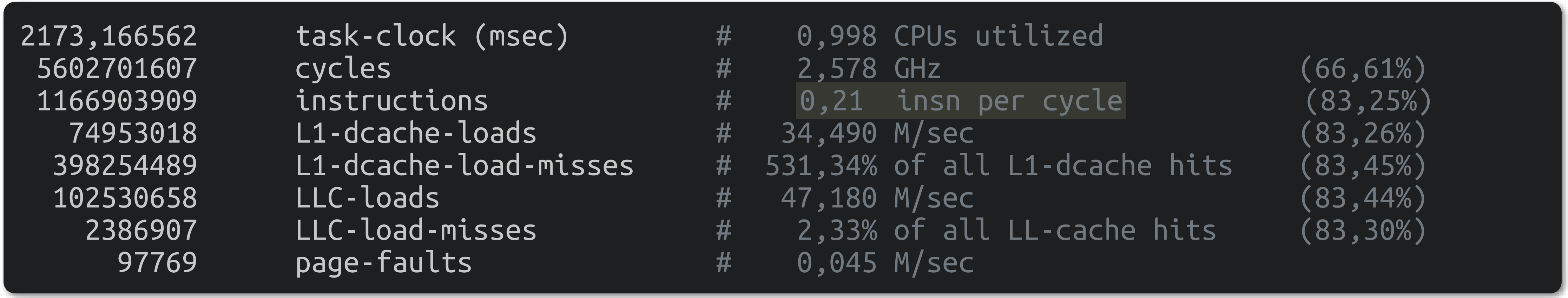


```
2173,166562      task-clock (msec)          #    0,998 CPUs utilized
 5602701607      cycles                     #    2,578 GHz                        (66,61%)
 1166903909      instructions               #    0,21  insn per cycle             (83,25%)
   74953018      L1-dcache-loads            #   34,490 M/sec                      (83,26%)
  398254489      L1-dcache-load-misses      #  531,34% of all L1-dcache hits      (83,45%)
  102530658      LLC-loads                  #   47,180 M/sec                      (83,44%)
    2386907      LLC-load-misses            #    2,33% of all LL-cache hits       (83,30%)
      97769      page-faults                #    0,045 M/sec
```

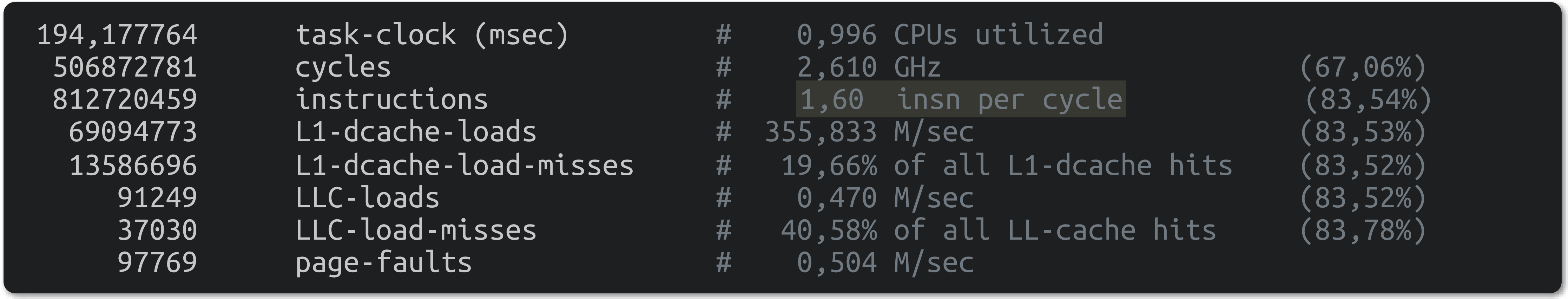


```
 194,177764      task-clock (msec)          #    0,996 CPUs utilized
  506872781      cycles                     #    2,610 GHz                        (67,06%)
  812720459      instructions               #    1,60  insn per cycle             (83,54%)
   69094773      L1-dcache-loads            #  355,833 M/sec                      (83,53%)
   13586696      L1-dcache-load-misses      #   19,66% of all L1-dcache hits      (83,52%)
      91249      LLC-loads                  #    0,470 M/sec                      (83,52%)
      37030      LLC-load-misses            #   40,58% of all LL-cache hits       (83,78%)
      97769      page-faults                #    0,504 M/sec
```

CPU CACHE

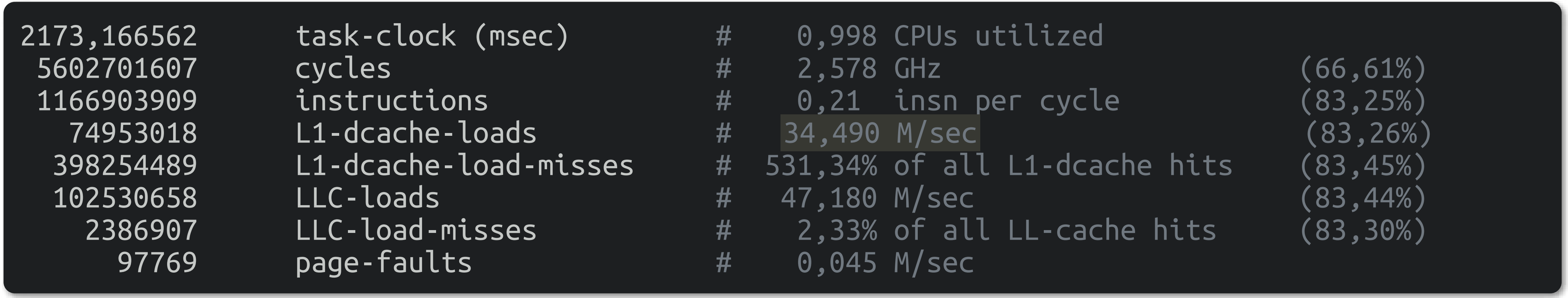


```
2173,166562      task-clock (msec)        #    0,998 CPUs utilized
 5602701607      cycles                   #    2,578 GHz                      (66,61%)
 1166903909      instructions             #    0,21  insn per cycle           (83,25%)
   74953018      L1-dcache-loads          #   34,490 M/sec                    (83,26%)
  398254489      L1-dcache-load-misses    #  531,34% of all L1-dcache hits    (83,45%)
  102530658      LLC-loads                #   47,180 M/sec                    (83,44%)
    2386907      LLC-load-misses          #    2,33% of all LL-cache hits     (83,30%)
      97769      page-faults              #    0,045 M/sec
```

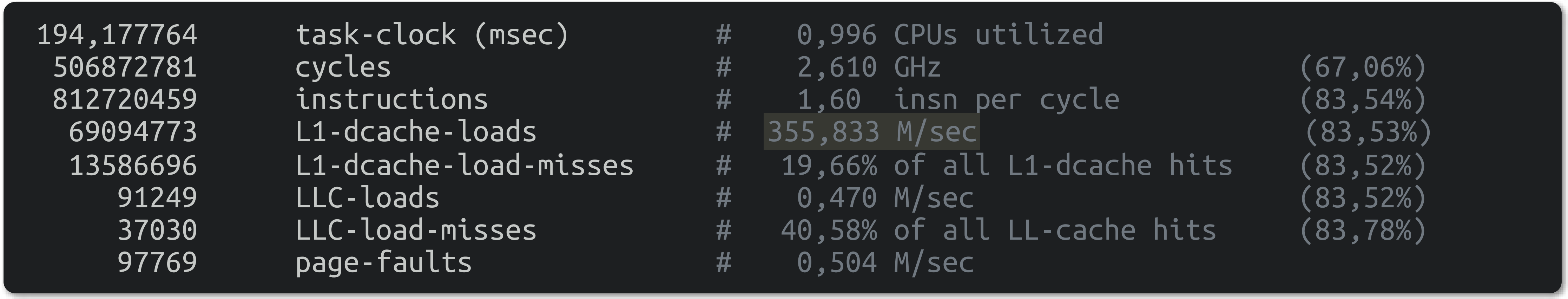


```
 194,177764      task-clock (msec)        #    0,996 CPUs utilized
  506872781      cycles                   #    2,610 GHz                      (67,06%)
  812720459      instructions             #    1,60  insn per cycle           (83,54%)
   69094773      L1-dcache-loads          #  355,833 M/sec                    (83,53%)
   13586696      L1-dcache-load-misses    #   19,66% of all L1-dcache hits    (83,52%)
      91249      LLC-loads                #    0,470 M/sec                    (83,52%)
      37030      LLC-load-misses          #   40,58% of all LL-cache hits     (83,78%)
      97769      page-faults              #    0,504 M/sec
```

CPU CACHE

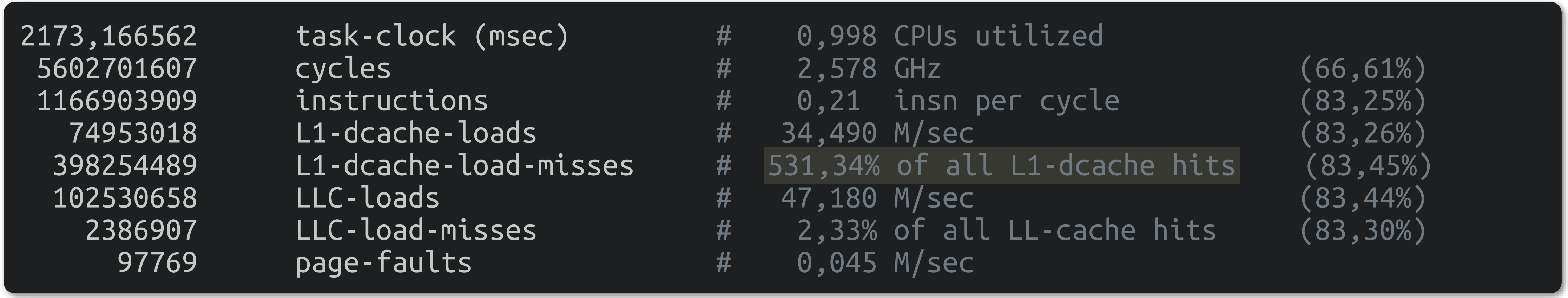


```
2173,166562      task-clock (msec)        #    0,998 CPUs utilized
 5602701607      cycles                   #    2,578 GHz                          (66,61%)
 1166903909      instructions             #    0,21  insn per cycle               (83,25%)
   74953018      L1-dcache-loads          #   34,490 M/sec                         (83,26%)
  398254489      L1-dcache-load-misses    #  531,34% of all L1-dcache hits        (83,45%)
  102530658      LLC-loads                #   47,180 M/sec                        (83,44%)
    2386907      LLC-load-misses          #    2,33% of all LL-cache hits         (83,30%)
      97769      page-faults              #    0,045 M/sec
```

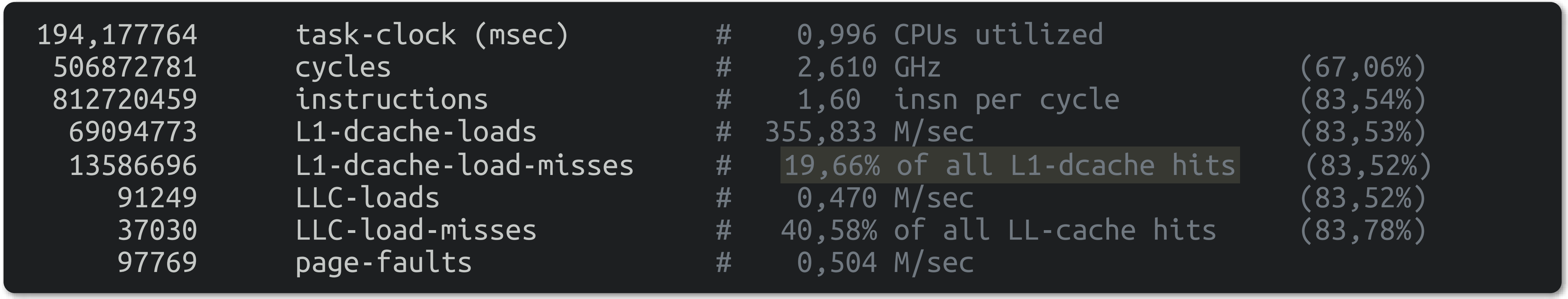


```
 194,177764      task-clock (msec)        #    0,996 CPUs utilized
  506872781      cycles                   #    2,610 GHz                          (67,06%)
  812720459      instructions             #    1,60  insn per cycle               (83,54%)
   69094773      L1-dcache-loads          #  355,833 M/sec                         (83,53%)
   13586696      L1-dcache-load-misses    #   19,66% of all L1-dcache hits        (83,52%)
      91249      LLC-loads                #    0,470 M/sec                        (83,52%)
      37030      LLC-load-misses          #   40,58% of all LL-cache hits         (83,78%)
      97769      page-faults              #    0,504 M/sec
```

CPU CACHE

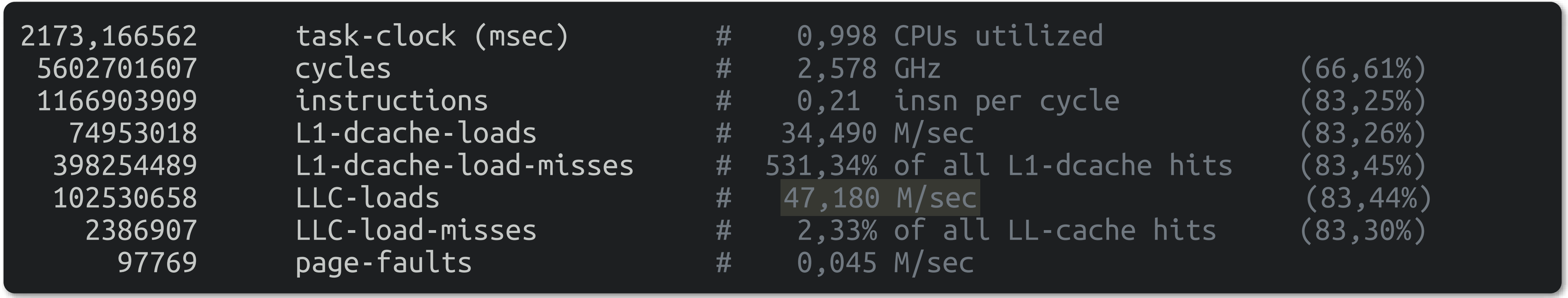


```
2173,166562      task-clock (msec)        #    0,998 CPUs utilized
 5602701607      cycles                   #    2,578 GHz                    (66,61%)
 1166903909      instructions             #    0,21  insn per cycle         (83,25%)
   74953018      L1-dcache-loads          #   34,490 M/sec                  (83,26%)
  398254489      L1-dcache-load-misses    #  531,34% of all L1-dcache hits   (83,45%)
  102530658      LLC-loads                #   47,180 M/sec                  (83,44%)
    2386907      LLC-load-misses          #    2,33% of all LL-cache hits   (83,30%)
      97769      page-faults              #    0,045 M/sec
```

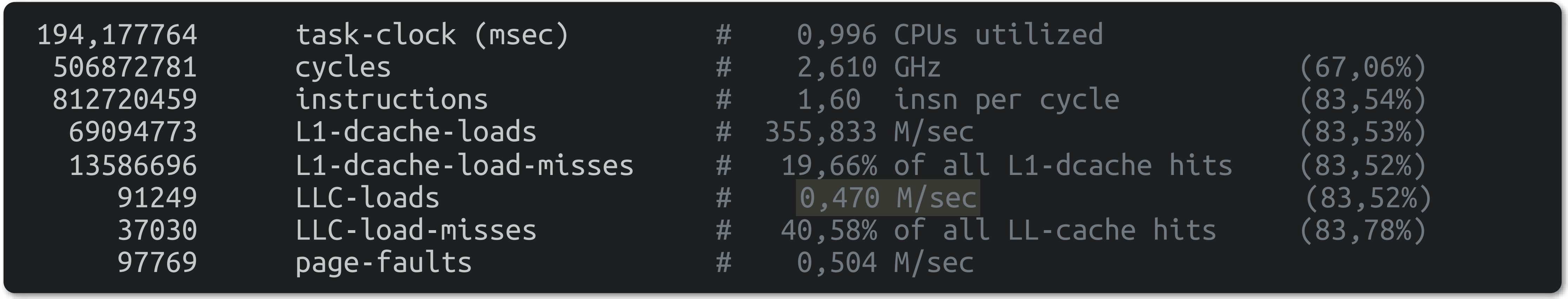


```
 194,177764      task-clock (msec)        #    0,996 CPUs utilized
  506872781      cycles                   #    2,610 GHz                    (67,06%)
  812720459      instructions             #    1,60  insn per cycle         (83,54%)
   69094773      L1-dcache-loads          #  355,833 M/sec                  (83,53%)
   13586696      L1-dcache-load-misses    #   19,66% of all L1-dcache hits   (83,52%)
      91249      LLC-loads                #    0,470 M/sec                  (83,52%)
      37030      LLC-load-misses          #   40,58% of all LL-cache hits   (83,78%)
      97769      page-faults              #    0,504 M/sec
```

CPU CACHE



```
2173,166562      task-clock (msec)          #    0,998 CPUs utilized
 5602701607      cycles                     #    2,578 GHz                      (66,61%)
 1166903909      instructions               #    0,21  insn per cycle           (83,25%)
   74953018      L1-dcache-loads            #   34,490 M/sec                    (83,26%)
  398254489      L1-dcache-load-misses      #  531,34% of all L1-dcache hits    (83,45%)
  102530658      LLC-loads                  #   47,180 M/sec                    (83,44%)
    2386907      LLC-load-misses            #    2,33% of all LL-cache hits     (83,30%)
      97769      page-faults                #    0,045 M/sec
```



```
 194,177764      task-clock (msec)          #    0,996 CPUs utilized
  506872781      cycles                     #    2,610 GHz                      (67,06%)
  812720459      instructions               #    1,60  insn per cycle           (83,54%)
   69094773      L1-dcache-loads            #  355,833 M/sec                    (83,53%)
   13586696      L1-dcache-load-misses      #   19,66% of all L1-dcache hits    (83,52%)
      91249      LLC-loads                  #    0,470 M/sec                    (83,52%)
      37030      LLC-load-misses            #   40,58% of all LL-cache hits     (83,78%)
      97769      page-faults                #    0,504 M/sec
```

ANOTHER EXAMPLE

```cpp
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)); }
}
```

NAIIVE 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; }
};
```

NAIIVE 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
- Focus on data layout
- Results in objects decomposition

DOD (DATA-ORIENTED DESIGN)

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

Keep data used together close to each other

OBJECT DECOMPOSITION EXAMPLE

```cpp
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;
  double d;
  short s;
  static double dd;
  int i;
};

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

MEMBERS ORDER MIGHT BE IMPORTANT

```cpp
struct A {
  char c;      // size=1, alignment=1, padding=7
  double d;    // size=8, alignment=8, padding=0
  short s;     // size=2, alignment=2, padding=2
  static double dd;
  int i;       // size=4, alignment=4, padding=0
}; // size=24, alignment=8

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

MEMBERS ORDER MIGHT BE IMPORTANT

```
struct A {
  char c;     // size=1, alignment=1, padding=7
  double d;   // size=8, alignment=8, padding=0
  short s;    // size=2, alignment=2, padding=2
  static double dd;
  int i;      // size=4, alignment=4, padding=0
}; // size=24, alignment=8

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

```
struct B {
  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;
}; // size=16, alignment=8

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

MEMBERS ORDER MIGHT BE IMPORTANT

```
struct A {
  char c;      // size=1, alignment=1, padding=7
  double d;    // size=8, alignment=8, padding=0
  short s;     // size=2, alignment=2, padding=2
  static double dd;
  int i;       // size=4, alignment=4, padding=0
}; // size=24, alignment=8

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

```
struct B {
  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;
}; // size=16, alignment=8

static_assert( sizeof(B) == 16);
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 B {
  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;
}; // size=16, alignment=8

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

BE AWARE OF ALIGNMENT SIDE EFFECTS

```
struct B {
  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;
}; // size=16, alignment=8

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

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

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

BE AWARE OF ALIGNMENT SIDE EFFECTS

```
struct B {
  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;
}; // size=16, alignment=8

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

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

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

```
struct B {
  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;
}; // size=16, alignment=8

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

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

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

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

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

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

PACKING

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

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

```
struct B {
  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;
}; // size=16, alignment=8

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

PACKING

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

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

```
struct B {
  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;
}; // size=16, alignment=8

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

On modern hardware may be faster than aligned structure.

PACKING

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

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

```
struct B {
  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;
}; // size=16, alignment=8

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

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	
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	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	

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_;
  }
  // ...
};
```

DIVIDE 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;
  }
  // ...
};
```

DIVIDE 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_;
  }
  // ...
};
```

DIVIDE 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!

DIVIDE 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 better inlining

TYPICAL VALIDATION AND ERROR HANDLING

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

```
Error make_error(const request& r)
{
  switch(r.type) {
  request::type_1:
    return /* complex error msg generation */;
  request::type_2:
    return /* complex error msg generation */;
  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
  - consider using **`tsl::hopscotch_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
  - consider using **`tsl::hopscotch_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
  - consider using **`tsl::hopscotch_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
  - consider using **`tsl::hopscotch_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

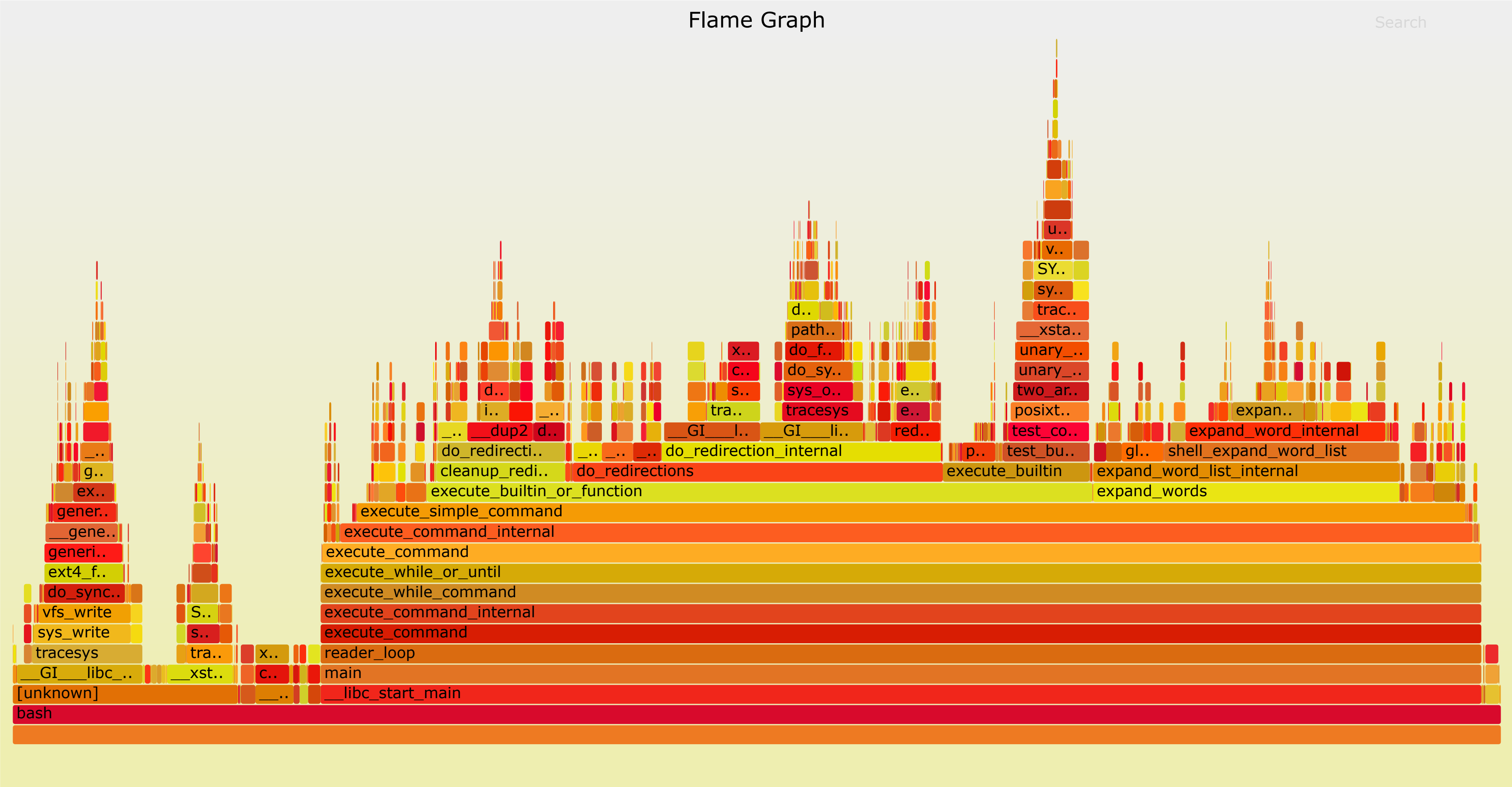
```
cmake -DCMAKE_BUILD_TYPE=Release ..
cmake -DCMAKE_BUILD_TYPE=RelWithDebInfo ..
```

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

CAUTION

Programming is addictive (and too much fun)