High Performance Message Dispatch

A Case Study in Zero-cost Abstractions

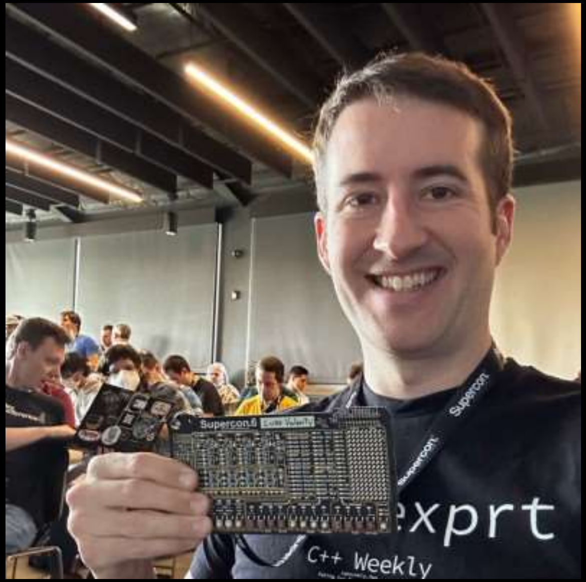
Luke Valenty

May 1, 2024

C++Now

Background

- Principal Engineer at Intel
  - Develop power management firmware and features for Intel chipsets
  - C++ advocate
- Dad
  - Two young kiddos
- Star Trek fan
  - Captain Pike might just be my favorite
  - Strange New Worlds and Lower Decks cross-over episode is the GOAT



lukevalenty
@lukevalenty@vivaldi.net
lukevalenty

Some notes before we begin...

I will omit `constexpr`, `[[nodiscard]]`, and `noexcept` in most cases to preserve screen space. Assume I would add them in the way you prefer.

A portion of the presentation is adapted from Ben Deane's "Message Handling with Boolean Implication" talk. You will see Ben's name on each slide adapted and you will notice me looking to him for approval.

Prologue

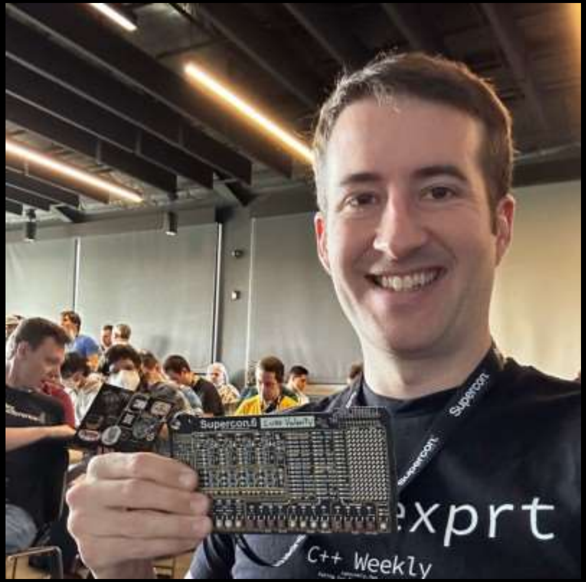
Requirements and constraints shape design

Background

- Principal Engineer at Intel

Develop power management firmware and features for Intel chipsets

- Dad
  - Two young kiddos
- Star Trek fan
  - Captain Pike might just be my favorite
  - Strange New Worlds and Lower Decks cross-over episode is the GOAT



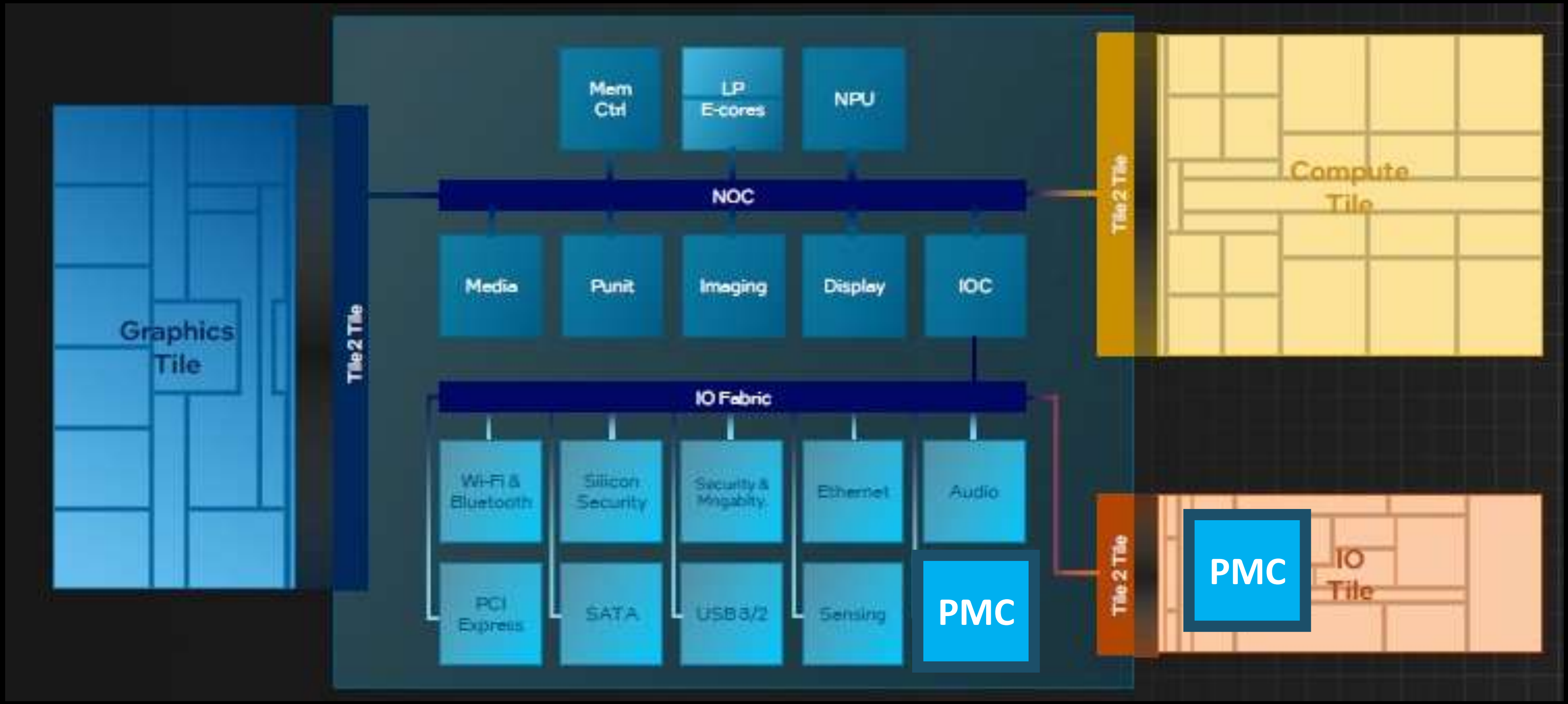
lukevalenty
@lukevalenty@vivaldi.net
lukevalenty

Power Management Firmware

1. Coordinate **power on and off** events for components in a system
2. Manage **active power vs. performance** tradeoffs
3. Report **energy usage**
4. Protect against **over-current and over-temperature** conditions

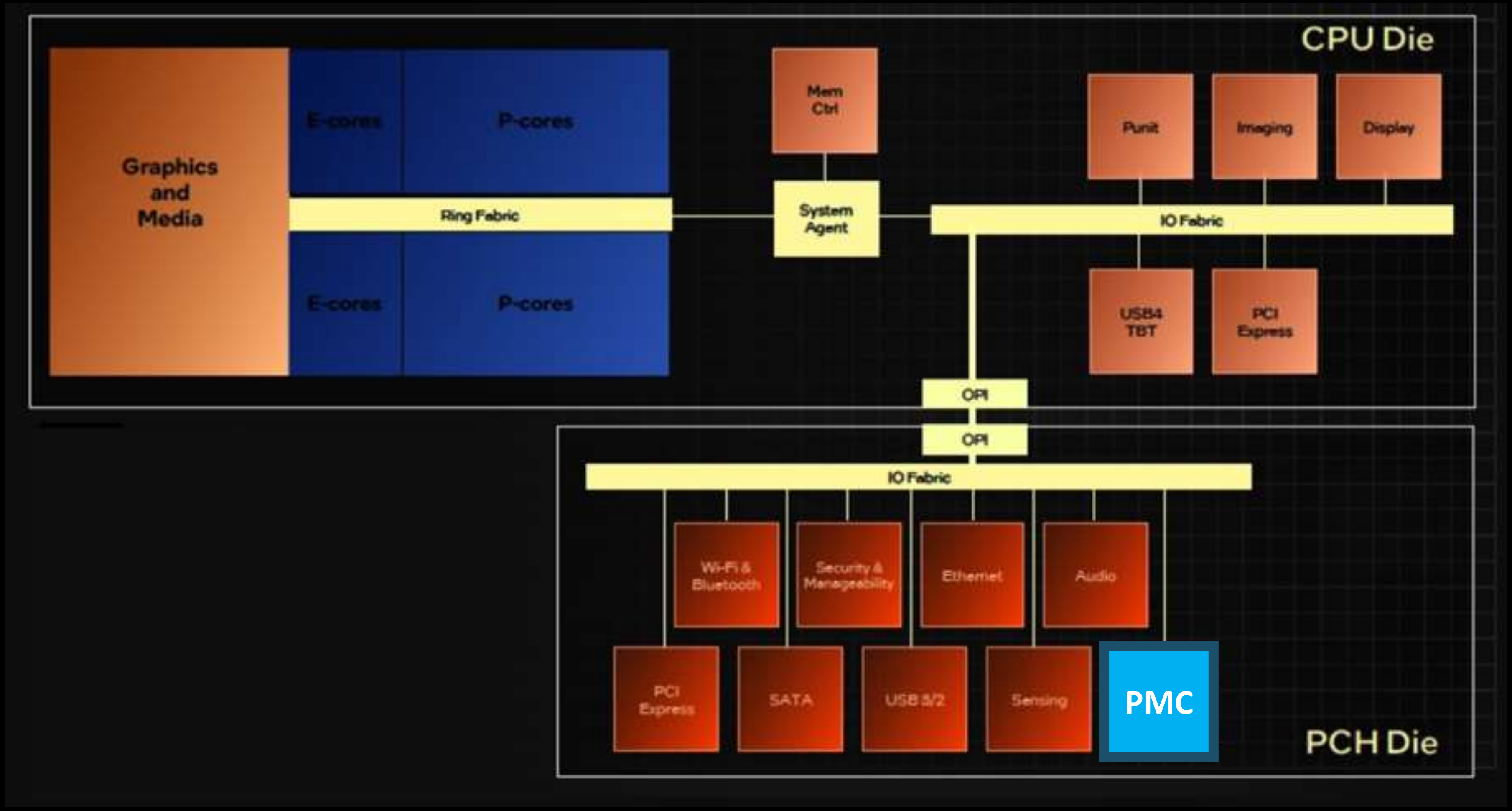
Robust *Performant* *Power Efficient* *Malleable*

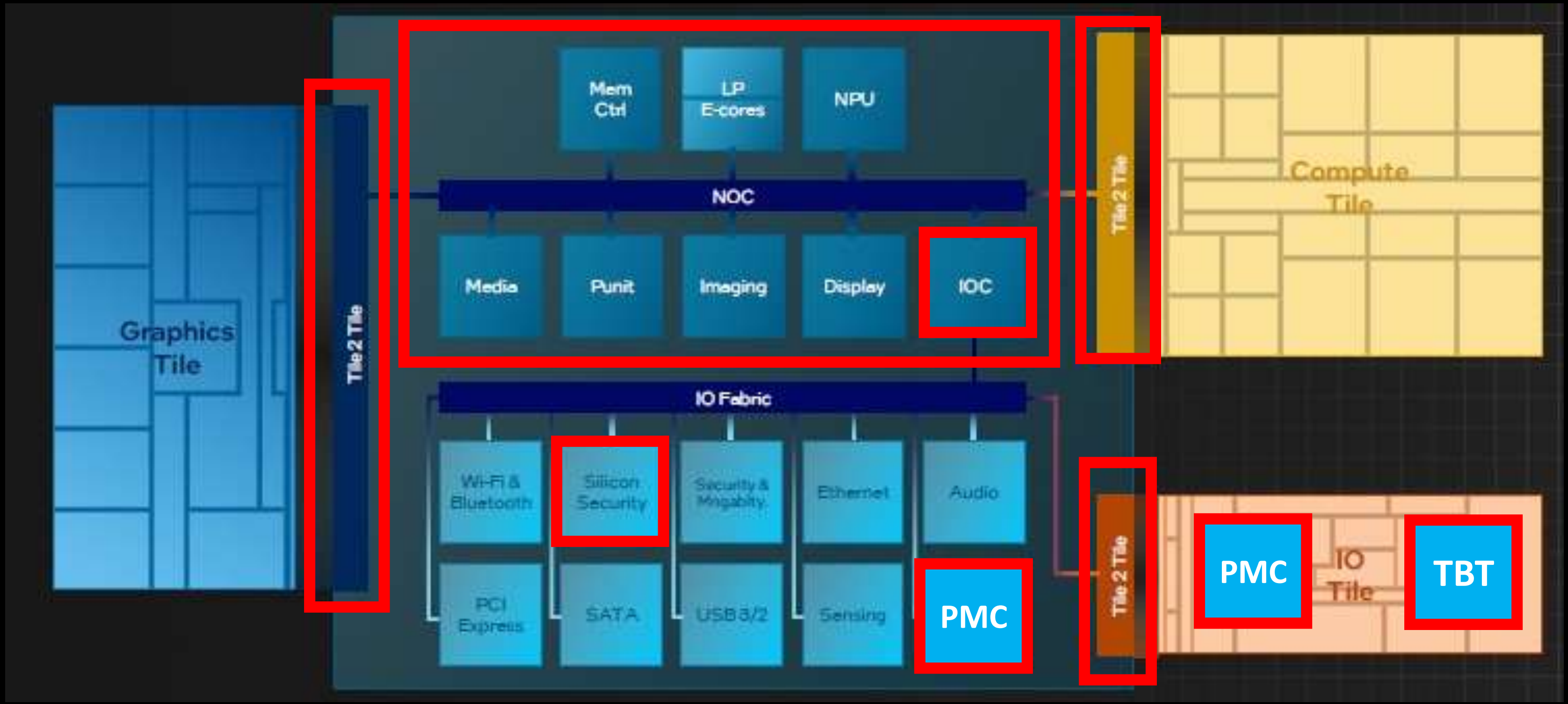
Malleable



Graphics Tile
Tile 2 Tile
Mem Ctrl
LP E-cores
NPU
NOC
Media
Punit
Imaging
Display
IOC
IO Fabric
Wi-Fi & Bluetooth
Silicon Security
Security & Mngabilty
Ethernet
Audio
PCI Express
SATA
USB 3/2
Sensing
PMC
Tile 2 Tile
Compute Tile
Tile 2 Tile
PMC
IO Tile

Malleable


CPU Die
Graphics and Media
E-cores
P-cores
Ring Fabric
E-cores
P-cores
Mem Ctrl
System Agent
Punit
Imaging
Display
IO Fabric
USB4 TBT
PCI Express
OPI
OPI
IO Fabric
Wi-Fi & Bluetooth
Security & Manageability
Ethernet
Audio
PCI Express
SATA
USB 3/2
Sensing
PMC
PCH Die

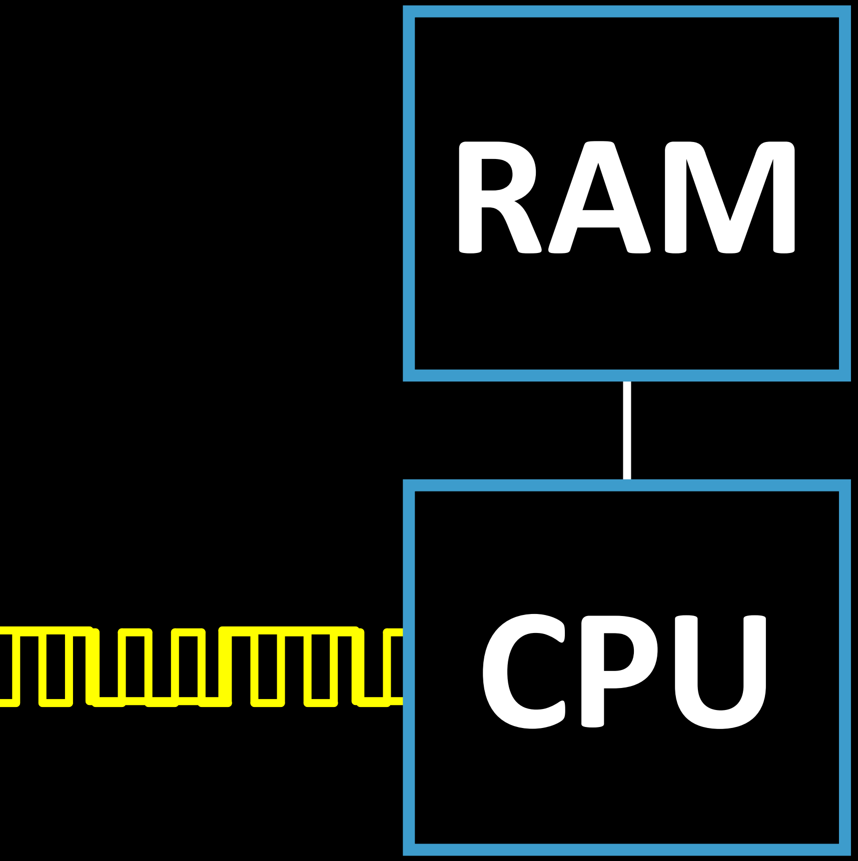


Malleable

Open to change

Decoupled components

Efficiently execute next *N* components per generation



Power Efficient

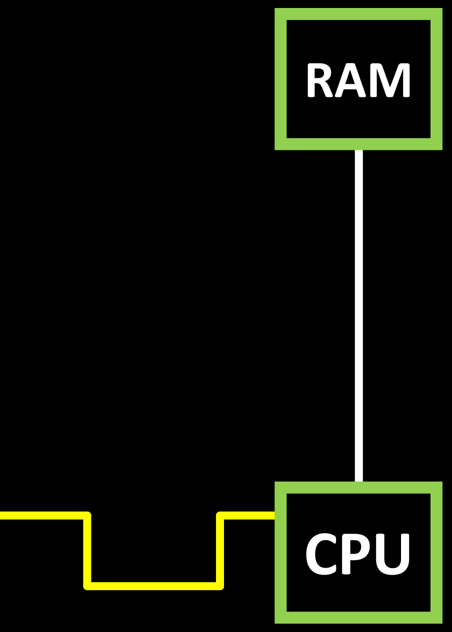
Small memories

Power efficient processor

Slow clocks

Manage our own power

Performant



Low latency

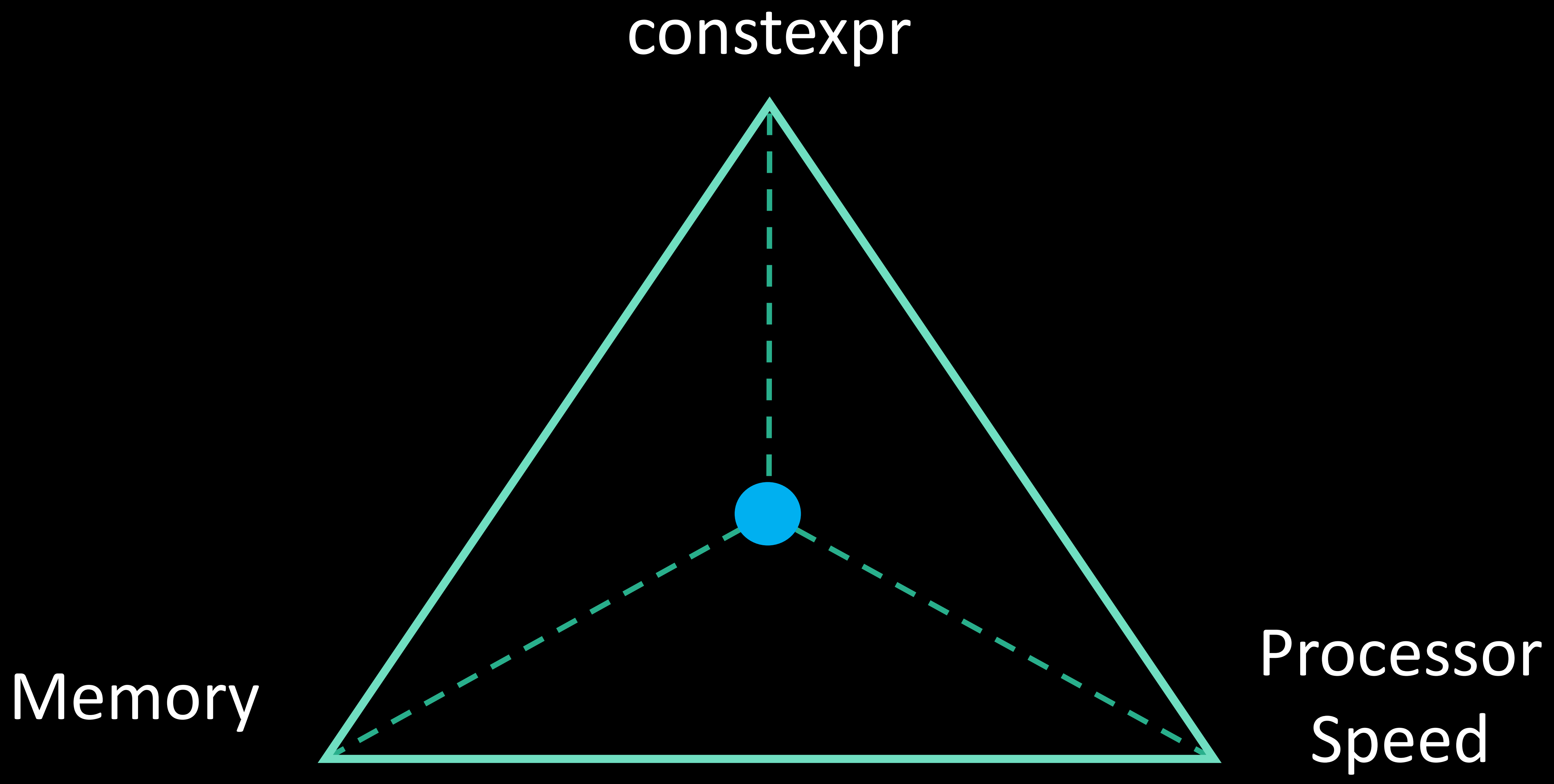
“High” throughput

Zero and
negative cost
abstractions

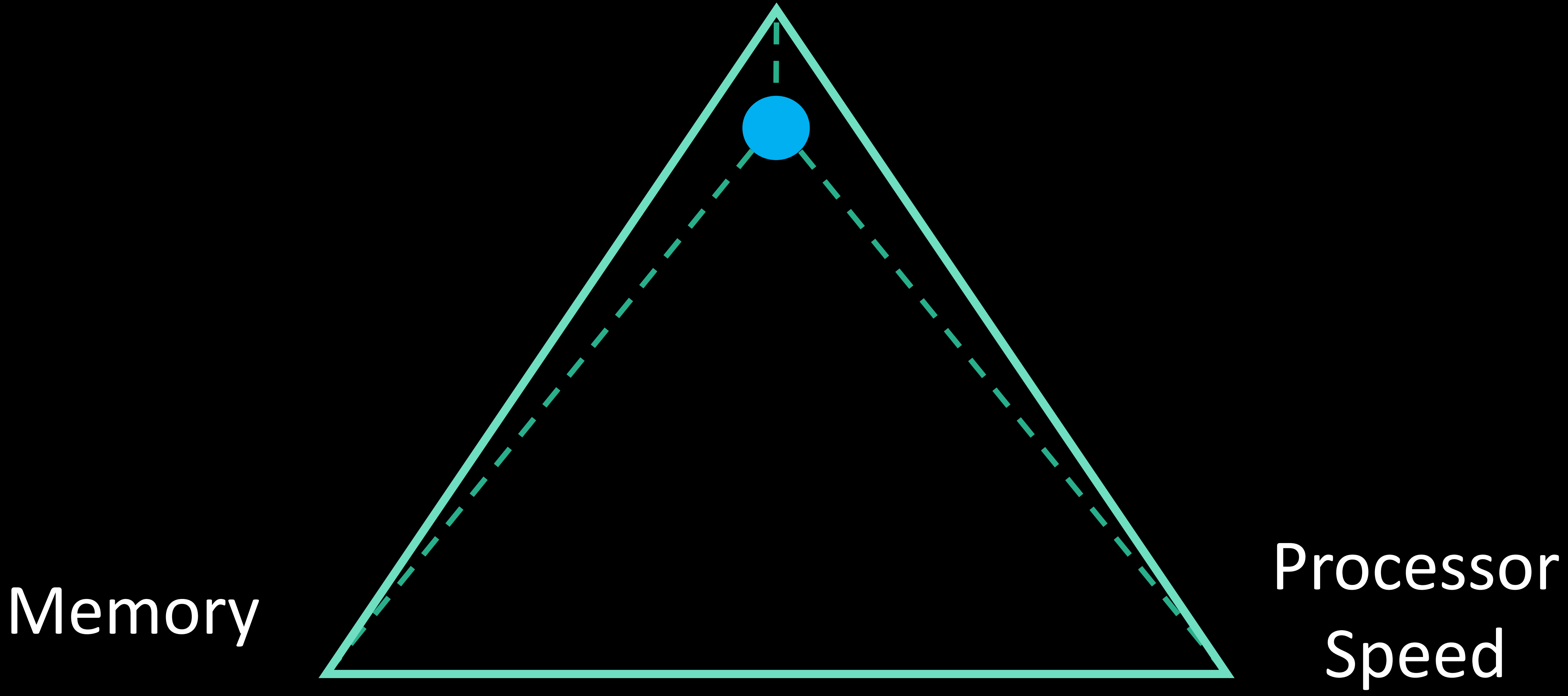
Math

Science

C++


constexpr
Memory
Processor
Speed

Science constexpr Math

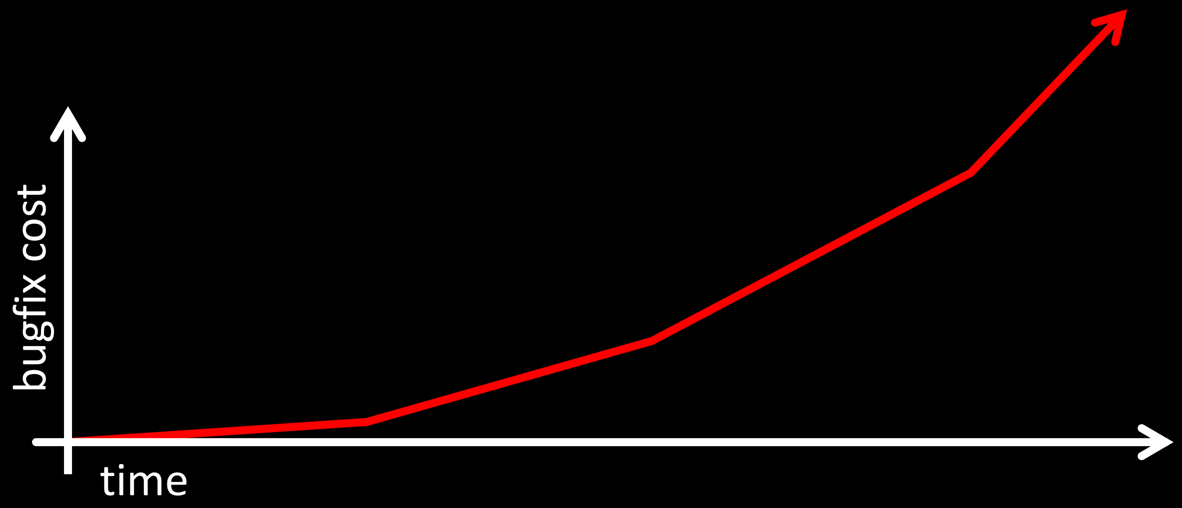


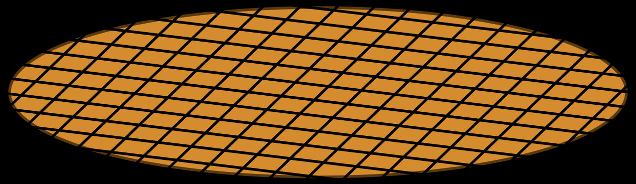
Robust

Correct by construction

Shift-left verification

Survivability





How do these constraints influence our firmware *design*?

No-cost/negative-cost mechanisms for **composition** and **extension**

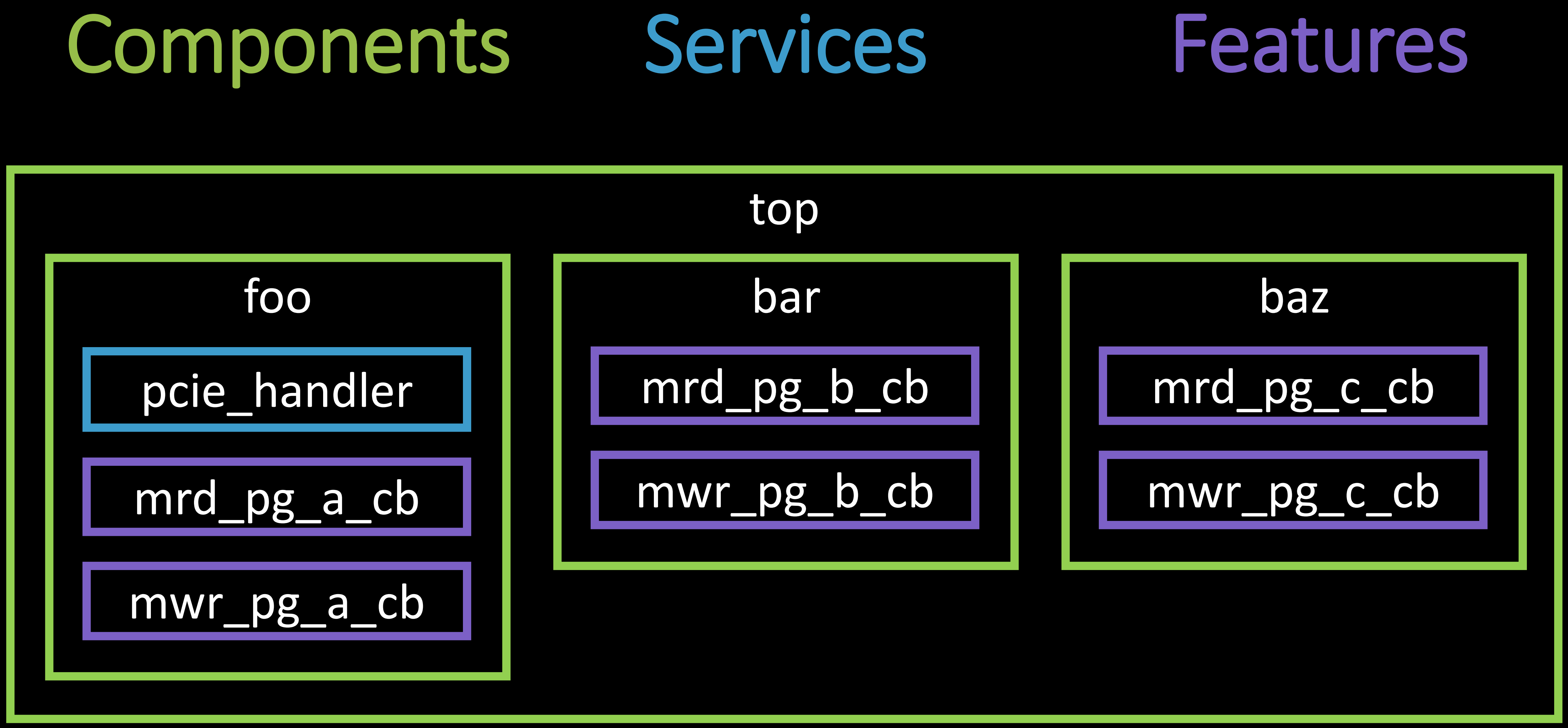
Stable, **extensible**, and declarative interfaces

Enable **correct-by-construction** techniques

CIB Compile-time Initialization and Build

C++ library for composing modular firmware at compile-time.

https://github.com/intel/compile-time-init-build


Components
Services
Features
top
foo
pcie_handler
mrd_pg_a_cb
mwr_pg_a_cb
bar
mrd_pg_b_cb
mwr_pg_b_cb
baz
mrd_pg_c_cb
mwr_pg_c_cb

top

↓

cib::nexus

↓

cib::service<S>

Services

Features

pcie_handler

mrd_pg_a_cb

mwr_pg_a_cb

mrd_pg_b_cb

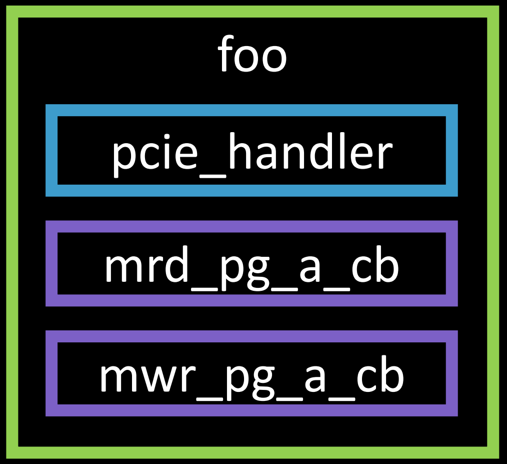
mwr_pg_b_cb

mrd_pg_c_cb

mwr_pg_c_cb

- **Features**
  - Controlled by customer
  - Constant change
  - Provide value to customer
- **Services**
  - Controlled by developer
  - Stable interface
  - Provide value to developer

Components




```
struct foo {
    constexpr static auto config = cib::config(
        cib::exports<pcie_handler>,
        cib::extend<pcie_handler>(
            mrd_page_a_cb,
            mwr_page_a_cb
        );
    );
};
```

Service

Service name

```
struct pcie_handler :
    msg::service<std::array<uint32_t, 6>> {};
```

Type

Configuration

pcie_handler

A service name declares an extensible interface that conforms to a service type. The service type connects a generic interface with a specific implementation.

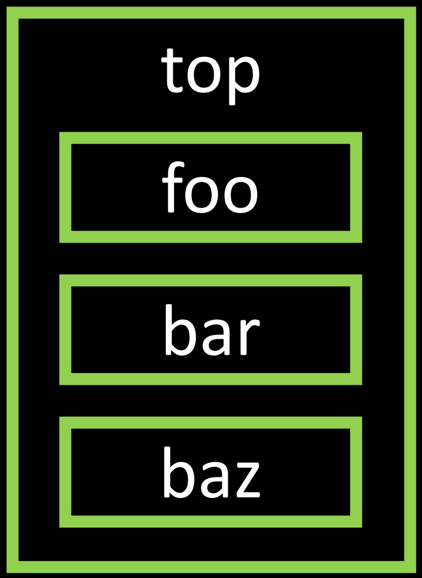
Feature

mrd_pg_a_cb

```
constexpr auto mrd_page_a_cb =
    msg::callback<"mrd page a", mrd32_defn>(
        "page"_f == 0_c,
        [](auto msg){
            // process page 'a' register read
        }
    );
```

A feature provides an extension for a specific service type. Features may declare additional information on what they extend within a service.

Components




```
struct top {
    constexpr static auto config =
        cib::components<
            foo,
            bar,
            baz
        >;
};
```

Nexus

```
cib::nexus<top> nexus{};

void incoming_pcie_msg(std::array<uint32_t, 6> msg) {
    cib::service<pcie_handler>->handle(msg);
}
```

Enables dependency inversion at compile-time

Service source code knows nothing about feature source code. Services are extended with features at compile-time.

It took several iterations to distill and extract the *cib* pattern....

1. Specific implementation to register callbacks with dependencies on other callbacks within a flow...
2. Generalized to support message handling as well
3. Fully generalized, but configuration was imperative, features were type-erased when added to services
4. ....

https://github.com/intel/compile-time-init-build

README Code of conduct BSL-1.0 license Security

cib - Compile-time Initialization and Build

Contributor Covenant 2.1 Unit Tests passing

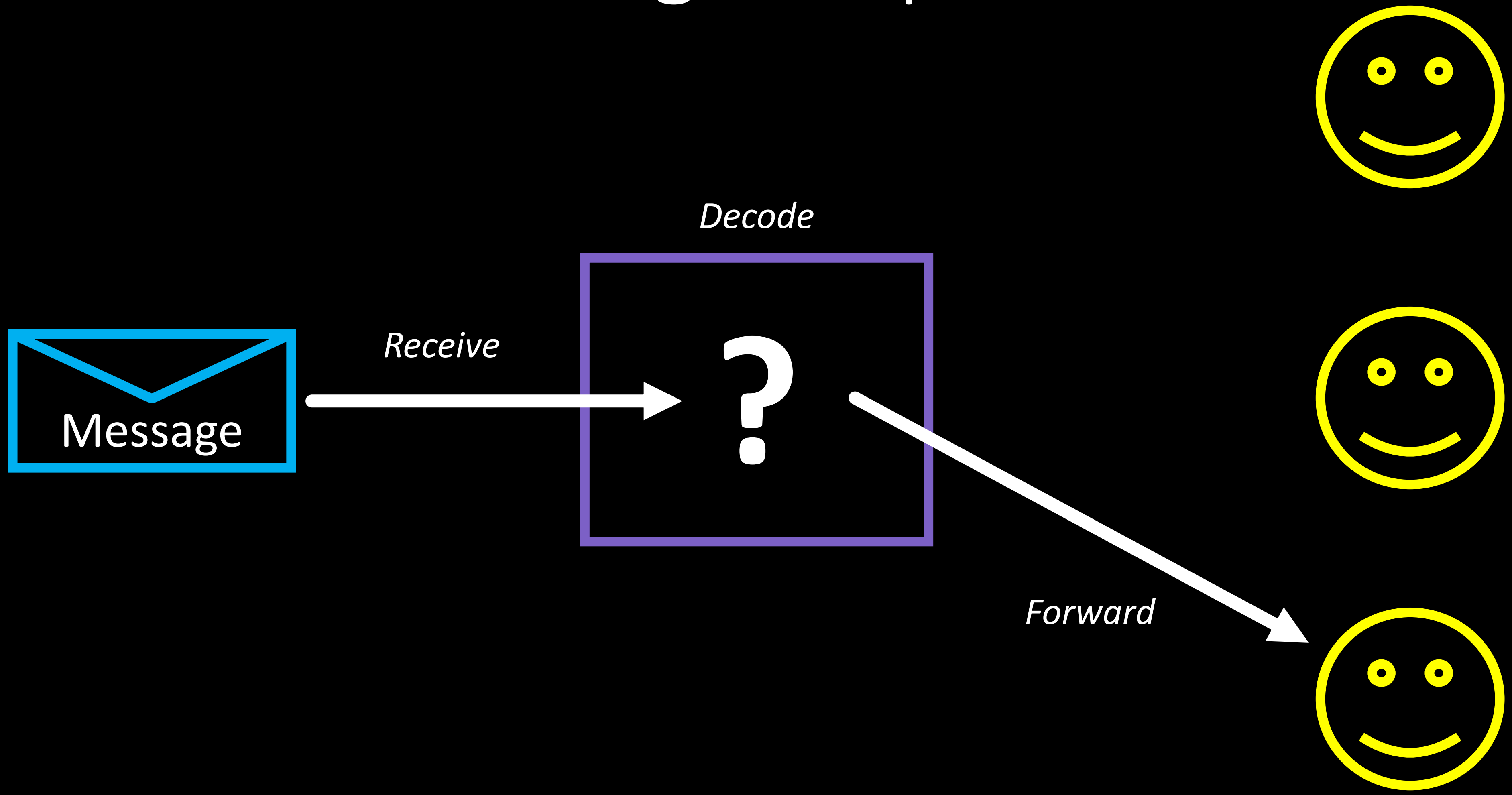
cib is a C++ header-only library for building embedded firmware with reusable components. It implements the compile-time initialization and build pattern. Instead of initializing components and registering callbacks at runtime, this process is executed at compile-time using constexpr or consteval functions.

Firmware using *cib* is implemented as a collection of **components**. Each component provides **services** and **features** to the build. Component **features** extend and provide new functionality to **services**.

Message Dispatch

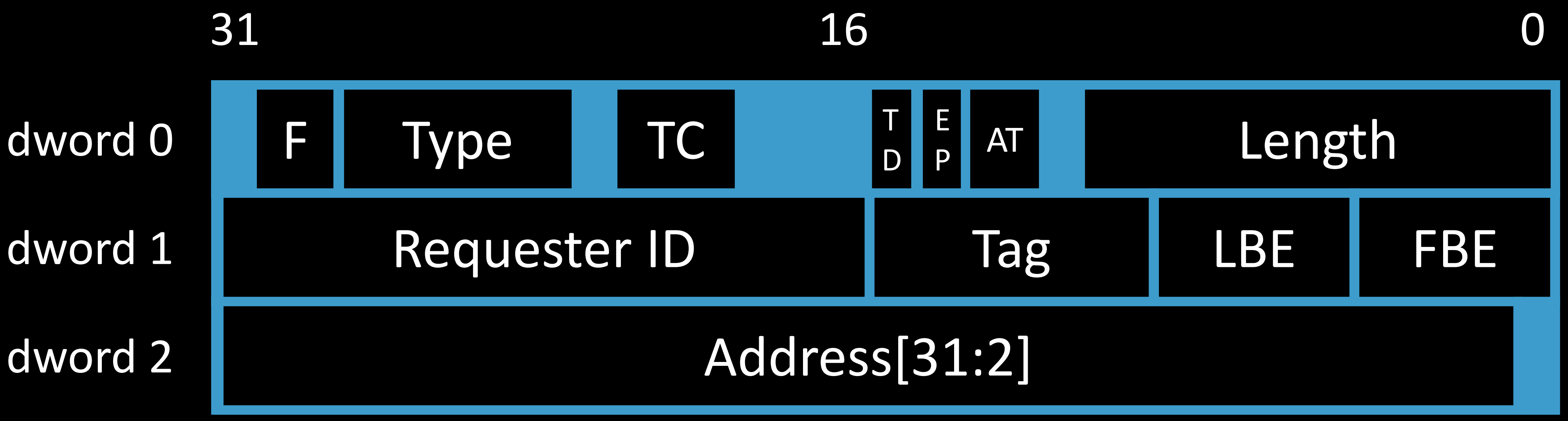
Case study in decoding and processing messages

Message Dispatch

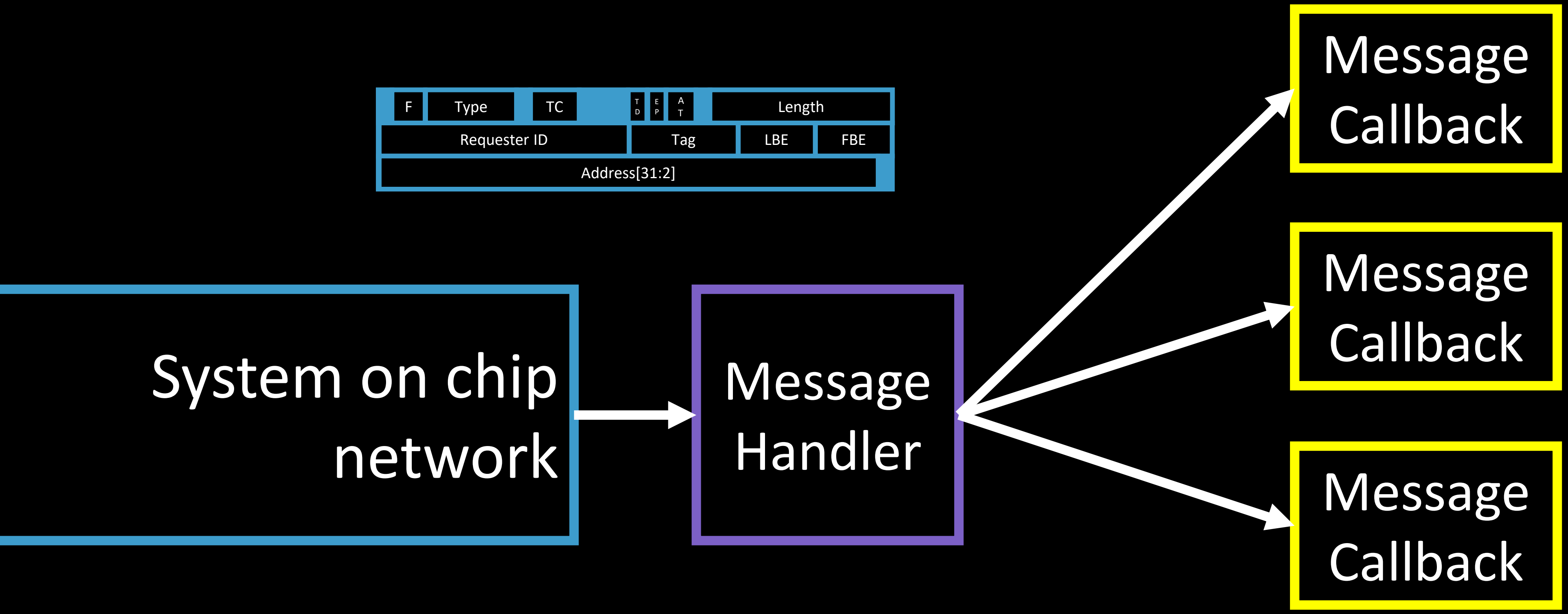


PCIe Packet Example

Memory read, 32-bit



Message Dispatch



More Message Dispatch Examples...

Software Design Patterns
- Event Bus
- Message Queue Architecture

Hardware Bus Protocols
- SPI Bus Target
- I2C Bus Target
- CAN Bus Node

Message dispatch should be...

Correct
Maintainable
Scalable
Fast

Not type safe

Not maintainable

Error prone

Not scalable

Obtuse!

```
void dispatch(uint32_t* data, size_t len) {
    uint32_t pkt_type = (data[0] >> 24) & 0b11111;
    uint32_t pkt_fmt =  (data[0] >> 29) & 0b11;

    if (pkt_type == 0) {
        if (pkt_fmt == 0) {
            process_mrd32(data, len);
        } else if (pkt_fmt == 2) {
            process_mwr32(data, len);
        }
    } else if (pkt_type == 2) {
        if (pkt_fmt == 0) {
            process_iord(data, len);
        } else if (pkt_fmt == 2) {
            process_iowr(data, len);
        }
    } else if (pkt_type == 10) {
        if (pkt_fmt == 2) {
            process_cpld(data, len);
        }
    }
}
```

How can we improve this?

1. Leverage type-safety
2. Compile-time programming
3. Identify and lift-up fundamental concepts
4. Use declarative-style
5. Enable decoupled components
6. Implement fast dispatch algorithm

```c
void dispatch(uint32_t* data, size_t len) {
    uint32_t pkt_type = (data[0] >> 24) & 0b1111111;

    if (pkt_type == 0x00) {
        process_mrd32(data, len);
    } else if (pkt_type == 0x40) {
        process_mwr32(data, len);
    } else if (pkt_type == 0x02) {
        process_iord(data, len);
    } else if (pkt_type == 0x42) {
        process_iowr(data, len);
    } else if (pkt_type == 0x4a) {
        process_cpld(data, len);
    }
}
```

```cpp
void dispatch(uint32_t* data, size_t len) {
    auto const pkt_type =
        static_cast<pkt_t>((data[0] >> 24) & 0b1111111);

    switch (pkt_type) {
        case pkt_t::mrd32:
            process_mrd32(data, len);
            break;
        case pkt_t::mwr32:
            process_mwr32(data, len);
            break;
        case pkt_t::iord:
            process_iord(data, len);
            break;
        case pkt_t::iowr:
            process_iowr(data, len);
            break;
        case pkt_t::cpld:
            process_cpld(data, len);
            break;
    }
}
```

```cpp
void dispatch(std::span<uint32_t, 6> const & msg) {
    auto const pkt_type = get_pkt_type(msg);

    switch (pkt_type) {
        case pkt_t::mrd32:
            process_mrd32(msg);
            break;
        case pkt_t::mwr32:
            process_mwr32(msg);
            break;
        case pkt_t::iord:
            process_iord(msg);
            break;
        case pkt_t::iowr:
            process_iowr(msg);
            break;
        case pkt_t::cpld:
            process_cpld(msg);
            break;
    }
}
```

PCIe Packet

```
using length_f = field<"length",  u16>::located<at{0_dw,  9_msb,  0_lsb}>;
using type_f   = field<"type",  pkt_t>::located<at{0_dw, 28_msb, 24_lsb}>;
using fmt_f    = field<"fmt",   fmt_t>::located<at{0_dw, 30_msb, 29_lsb}>;

using fbe_f    = field<"fbe",      u8>::located<at{1_dw,  3_msb,  0_lsb}>;
using lbe_f    = field<"lbe",      u8>::located<at{1_dw,  7_msb,  4_lsb}>;
using tag_f    = field<"tag",      u8>::located<at{1_dw, 15_msb,  8_lsb}>;
using reqid_f  = field<"reqid",   u16>::located<at{1_dw, 31_msb, 16_lsb}>;

using addr_f   = field<"addr",    u32>::located<at{2_dw, 31_msb,  0_lsb}>;

using mrd32_defn = message<"mrd32", type_f, fmt_f, length_f, attr_f,
                                    ep_f, td_f, tc_f, fbe_f, lbe_f, tag_f,
                                    reqid_f, addr_f>;
```

Type safe?

```
using hdr = message<"hdr", type_f>;

void dispatch(const_view<hdr> msg) {
    switch (msg.get("type"_f)) {
        case pkt_t::mrd32:
            process_mrd32(msg);
            break;
        case pkt_t::mwr32:
            process_mwr32(msg);
            break;
        case pkt_t::iord:
            process_iord(msg);
            break;
        case pkt_t::iowr:
            process_iowr(msg);
            break;
        case pkt_t::cpld:
            process_cpld(msg);
            break;
    }
}
```

Not type-safe

process_* functions need to “cast” to the more specific message type with the appropriate fields.

```
void process_mrd32(const_view<hdr> msg_hdr) {
    owning<mrd32> const msg{msg_hdr};

    // do something with msg
}
```

Scalable?

Maintainable?

```
using hdr = message<"hdr", type_f>;

void dispatch(const_view<hdr> msg) {
    switch (msg.get("type"_f)) {
        case pkt_t::mrd32:
            process_mrd32(msg);
            break;
        case pkt_t::mwr32:
            process_mwr32(msg);
            break;
        case pkt_t::iord:
            process_iord(msg);
            break;
        case pkt_t::iowr:
            process_iowr(msg);
            break;
        case pkt_t::cpld:
            process_cpld(msg);
            break;
    }
}
```

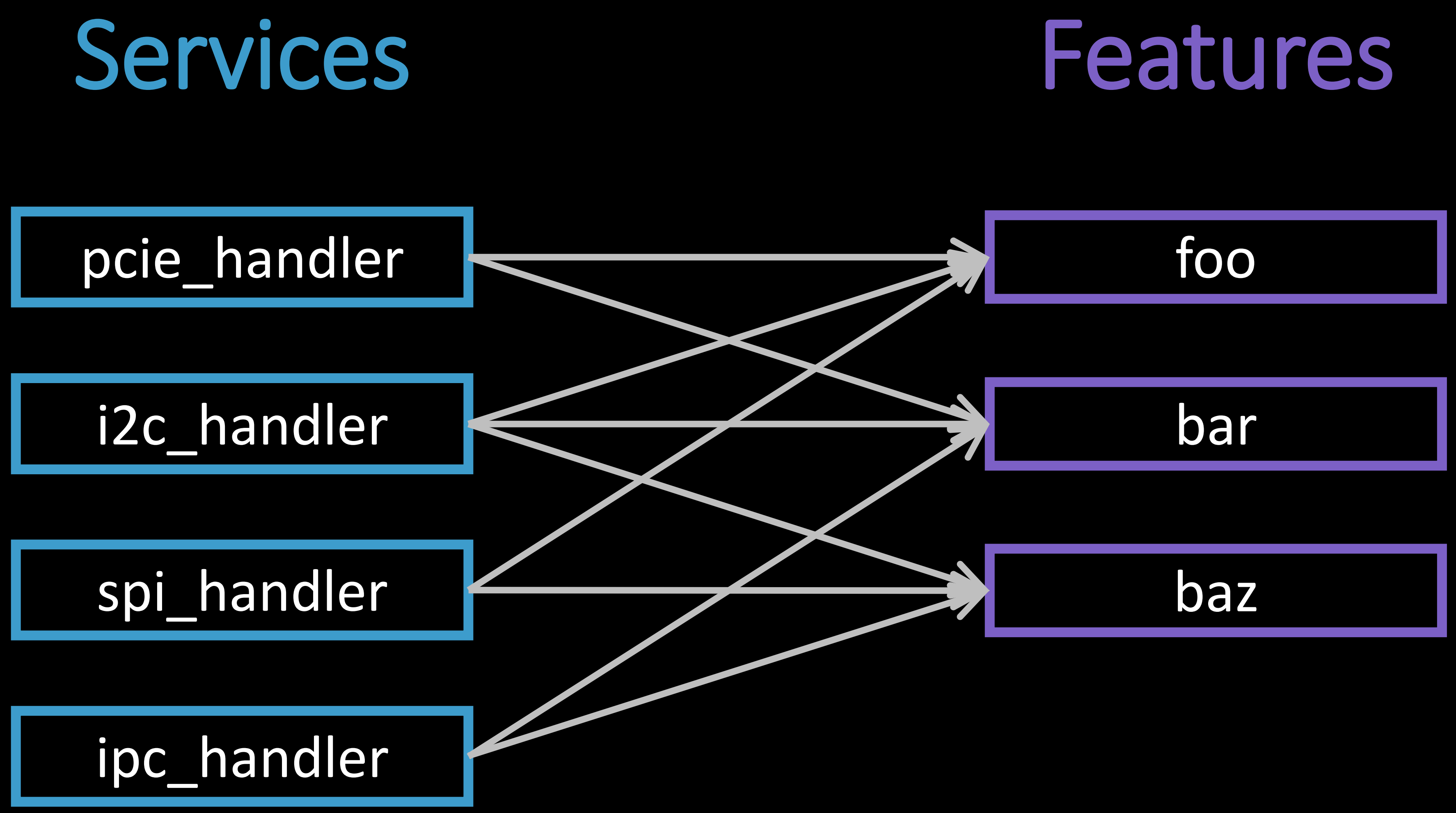
Not scalable

Not maintainable

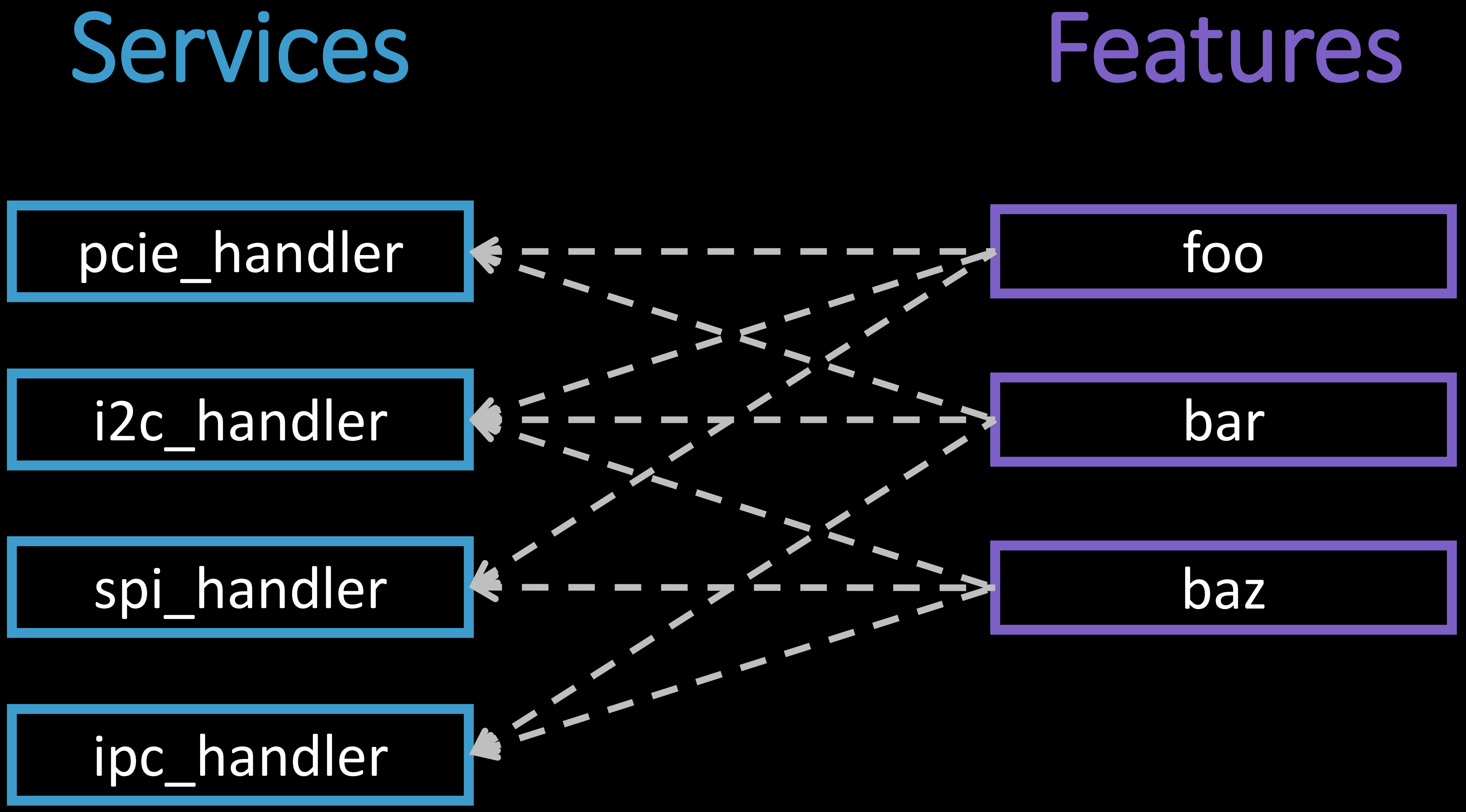
Core message dispatch is coupled to every feature that needs to process messages. Decoding strategies are implemented manually.

```
void process_mrd32(const_view<hdr> msg_hdr) {
    owning<mrd32> const msg{msg_hdr};

    auto const addr = msg.get("addr"_f);
    auto const page = addr >> 10;

    if      (page == 0) process_page_a(msg);
    else if (page == 1) process_page_b(msg);
    else if (...) ...;
    else                process_page_error(msg);
}
```

Not scalable, **Not** maintainable



Scalable and maintainable



Fixing this will require a larger change...

Message Services

Message Callbacks

Message Callback

Create callback template type

```
constexpr auto mrd_page_a_cb =
    msg::callback([](const_view<mrd32> msg){
        if (
            msg.get("type"_f) == pkt_t::mrd32 and
            msg.get("page"_f) == 0
        ) {
            // process page 'a' register read
        }
    });
```

Message Callback

Add callback name for logging

```
constexpr auto mrd_page_a_cb =
    msg::callback<"mrd page a">([](const_view<mrd32> msg){
        if (
            msg.get("type"_f) == pkt_t::mrd32 and
            msg.get("page"_f) == 0
        ) {
            // process page 'a' register read
        }
    });
```

Message Callback

Move message definition into template parameter

```
constexpr auto mrd_page_a_cb =
    msg::callback<"mrd page a", mrd32>([](auto msg){
        if (
            msg.get("type"_f) == pkt_t::mrd32 and
            msg.get("page"_f) == 0
        ) {
            // process page 'a' register read
        }
    });
```

Message Callback

Move callback predicate into an argument

```
constexpr auto mrd_page_a_cb =
    msg::callback<"mrd page a", mrd32>(
        "type"_f == constant<pkt_t::mrd32> and
        "page"_f == 0_c,
        [](auto msg){
            // process page 'a' register read
        }
    );
```

Message Callback

Move message requirements into message definition.

```
using mrd32_defn = message<"mrd32",
    type_f::with_required<pkt_t::mrd32>, ...>;

constexpr auto mrd_page_a_cb =
    msg::callback<"mrd page a", mrd32_defn>(
        "page"_f == 0_c,
        [](auto msg){
            // process page 'a' register read
        }
    );
```

Name

Message Type

Predicate

Action

```
constexpr auto mrd_page_a_cb =
    msg::callback<
        "mrd page a",
        mrd32_defn
    >(
        "page"_f == 0_c,

        [](auto msg){
            // process page 'a' register read
        }
    );
```

Type-safe

Declarative

Concise

Message Declaration

What is a message?

Fields

A *field* is composed of:

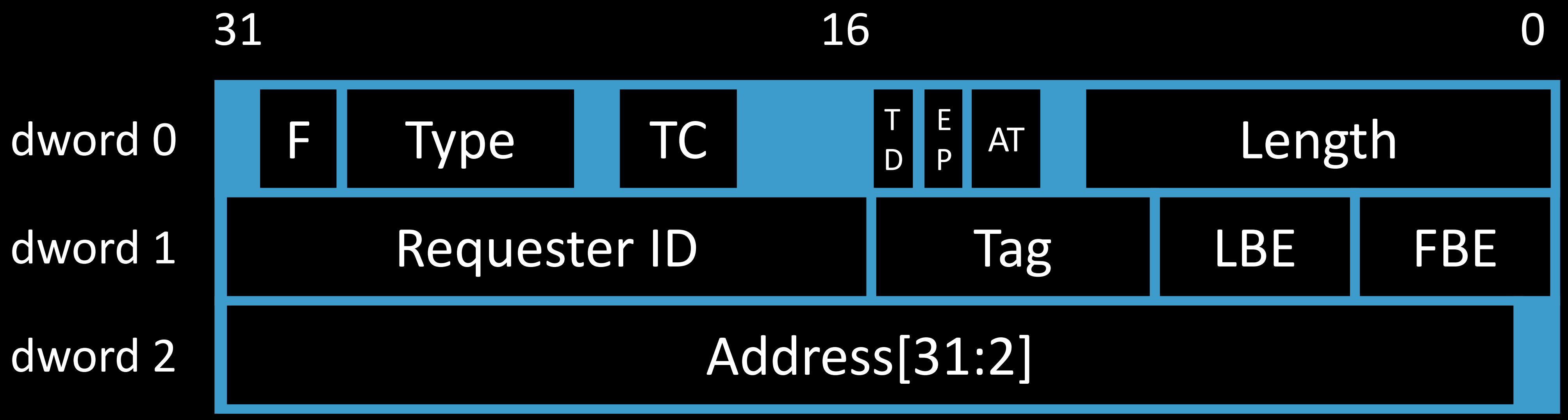
- Human-readable name
- *Type* (commonly integral or enumeration)
- One or more *locators* that define the layout

Field *locators* are specified with three items:

- Double-word index (in 32-bit dwords)
- Most significant bit (inclusive!)
- Least significant bit

PCIe Packet Example

Memory read, 32-bit



PCIe 32-bit Memory Read Packet Fields

Tag *Name* *Type* *Locator*

```
using length_f = field<"length",  u16>::located<at{0_dw,  9_msb,  0_lsb}>;
using type_f   = field<"type",   pkt_t>::located<at{0_dw, 28_msb, 24_lsb}>;
using fmt_f    = field<"fmt",    fmt_t>::located<at{0_dw, 30_msb, 29_lsb}>;

using fbe_f    = field<"fbe",       u8>::located<at{1_dw,  3_msb,  0_lsb}>;
using lbe_f    = field<"lbe",       u8>::located<at{1_dw,  7_msb,  4_lsb}>;
using tag_f    = field<"tag",       u8>::located<at{1_dw, 15_msb,  8_lsb}>;
using reqid_f  = field<"reqid",    u16>::located<at{1_dw, 31_msb, 16_lsb}>;

using addr_f   = field<"addr",     u32>::located<at{2_dw, 31_msb,  0_lsb}>;
```

Fields have two methods

```
template <range R>
static auto extract(R && r) -> value_type;

template <range R>
static auto insert(R && r, value_type const & value) -> void;
```

Given a range over underlying data, a `field` can read and write its value in the layout.

Note: *a* `field` *is an empty type! Definition, not data.*

Messages

Ben Deane

Messages are defined as a collection of `fields`. This definition is also an empty type. A message instance is a combination of the definition with some storage.

```
using mrd32_defn = message<"mrd32", type_f, fmt_f, length_f, attr_f, ep_f,
                                    td_f, tc_f, fbe_f, lbe_f, tag_f, reqid_f,
                                    addr_f>;

owning<mrd32_defn> m{}; // defn + right-sized array

auto type = m.get("type"_f);
m.set("addr"_f = 0xc0010u);
```

A `message` instance has `get` and `set` functions which delegate to `extract` and `insert` functions on the appropriate `field`.

Matchers

To determine the correct message type for data arrived off the wire, we use a *matcher*.

```
template <typename T>
concept matcher =
    requires(T const & t, message const & msg) { // for a prototypical message
        { t(msg) } -> std::convertible_to<bool>;
    };
```

A matcher is a predicate on a message.

A simple matcher

```
template <typename Field, auto Value> struct equal_to_t {
    auto operator()(auto const & msg) const -> bool {
        return Value == msg.template get<Field>();
    }
};
```

This matcher tests that a given field matches a given value.

Ben Deane

A more generic matcher...

```
template <typename RelOp, typename Field, auto Value>
struct relational_matcher_t {
    auto operator()(auto const & msg) const -> bool {
        return Relop{}(Value, msg.template get<Field>());
    }
};
```

And then we have:

```
template <typename Field, auto Value>
using equal_to_t = relational_matcher_t<std::equal_to<>, Field, Value>;

template <typename Field, auto Value>
using less_than_t = relational_matcher_t<std::less<>, Field, Value>;

// etc...
```

Matchers *are* Boolean values

We can treat a function that returns a `bool` just like a `bool`. They are isomorphic.

All we need to do is to provide `and`, `or`, and `not` class templates that wrap matchers and implement `operator()` appropriately.

The resulting matcher will be an expression template.

Ben Deane

and Matcher

```
template <matcher L, matcher R>
struct and_t {
    L lhs;
    R rhs;

    auto operator()(auto const & msg) const -> bool {
        return lhs(msg) and rhs(msg);
    }
};
```

Ben Deane

or Matcher

```
template <matcher L, matcher R>
struct or_t {
    L lhs;
    R rhs;

    auto operator()(auto const & msg) const -> bool {
        return lhs(msg) or rhs(msg);
    }
};
```

not Matcher

```
template <matcher M>
struct not_t {
    M m;
    auto operator()(auto const & msg) const -> bool {
        return not m(msg);
    }
};
```

Ben Deane

The usual Boolean operators

```
template <matcher L, matcher R>
auto operator and(L const & lhs, R const & rhs) {
    return and_t{lhs, rhs};
}

template <matcher L, matcher R>
auto operator or(L const & lhs, R const & rhs) {
    return or_t{lhs, rhs};
}

template <matcher M>
auto operator not(M const & m) {
    return not_t{m};
}
```

Operator overloads for and, or and not on matchers.

The library in context

Now we have:

- Ways to define fields and messages
- Ways to match on messages
- Ways to combine matchers

Let’s look at some usage in context.

The library in context

```
// define the fields and message
using type_f = field<"type", std::uint8_t>::located<at{0_dw, 7_msb, 0_lsb}>;
using msg_defn = message<"msg", type_f>; // and more fields...

// a callback is a matcher and a function to call
auto cb = callback<"name", msg_defn>(
    "type"_f == 42_c, // a DSL that defines a matcher
    [] (const_view<msg_defn> msg) { /* do something */ });

// a handler encapsulates callbacks
auto h = handler(cb);

// when we handle some data, message views are overlayed
// and functions are called based on which matchers match the data
auto data = std::array{ ... };
h.handle(data);
```

Ben Deane

DSL for easy composition

```
auto cb = callback<"name", msg_defn>(
    "type"_f == 42_c
    and (
        "subtype"_f == 17_c or
        "subtype"_f == 21_c
    )
    and "other"_f > 9_c,
    [] (const_view<msg_defn> msg) {
        // do something
    }
);
```

We've made it easy for users to express constraints and match on messages.

The resulting matcher is an expression template.

Similar to *cib*, it took several iterations to distill and extract the message library design....

1. First version with no type-safety...it was embarressing
2. Second version with built-in storage and type-safety, but some unfortunate quirks.
3. Current open-source version, which has been iteratively improved

Gotta Go Fast!

Think hard and think good.

Fast Algorithm?

Databases use indices to speed up queries...can we do the same for message dispatch somehow?

Fast Algorithm?

Databases use indices to accelerate matching one query against many rows in a table

Can we flip that around to accelerate matching one row against many queries?

Yes!

Simple case: one index only

```
auto map = ...; // opcode -> callbacks...

auto dispatch(auto msg) -> void {
    auto const opcode = msg.get("opcode"_f);

    for (auto callback : map[opcode]) {
        callback(msg);
    }
}
```

This is a relatively simple and common pattern. In many cases the map can be as simple as an array.

Two indices

```
constexpr auto opcode_map = ...;
constexpr auto dest_map = ...;

auto dispatch(auto msg) -> void {
    auto const callbacks = intersection(
        opcode_map[msg.get("opcode"_f)],
        dest_map[msg.get("dest"_f)]
    );

    for (auto callback : callbacks) {
        callback(msg);
    }
}
```

This algorithm scales up to *N* indices...just add more maps and extend the intersection.

It depends on fast algorithms for the maps and intersection operation.

mem_read	b	untrusted_id

Callback	Opcode	Page	Source ID
mrd_pg_a_cb	mem_read	a	-
mwr_pg_a_cb	mem_write	a	-
mrd_pg_b_cb	mem_read	b	-
mwr_pg_b_cb	mem_write	b	-
mrd_pg_c_cb	mem_read	c	secure_id
mwr_pg_c_cb	mem_write	c	secure_id
log_msg	-	-	-

Opcode Index Lookup

Key	Value
mem_read	mrd_pg_a_cb, mrd_pg_b_cb, mrd_pg_c_cb, log_msg
mem_write	mwr_pg_a_cb, mwr_pg_b_cb, mwr_pg_c_cb, log_msg
-	log_msg

mem_read

Page Index Lookup

Key	Value
a	mrd_pg_a_cb, mwr_pg_a_cb, log_msg
b	mrd_pg_b_cb, mwr_pg_b_cb, log_msg
c	mrd_pg_c_cb, mwr_pg_c_cb, log_msg
-	log_msg

b

Source ID Index Lookup

Key	Value
secure_id	mrd_pg_c_cb, mwr_pg_c_cb, log_msg
-	mrd_pg_a_cb, mwr_pg_a_cb, mrd_pg_b_cb, mwr_pg_b_cb, log_msg

untrusted_id

Intersection

mem_read → mrd_pg_a_cb, $\boxed{\text{mrd\_pg\_b\_cb,}}$ mrd_pg_c_cb, $\boxed{\text{log\_msg}}$

b → $\boxed{\text{mrd\_pg\_b\_cb,}}$ mwr_pg_b_cb, $\boxed{\text{log\_msg}}$

untrusted_id → mrd_pg_a_cb, mwr_pg_a_cb, $\boxed{\text{mrd\_pg\_b\_cb,}}$ mwr_pg_b_cb, $\boxed{\text{log\_msg}}$

mrd_pg_b_cb, **log_msg**

We should be able to do this fast...

- Fast intersection calculation
  - Bitset intersection is bitwise AND
  - RLE of bitset should be fast and memory efficient
  - Hierarchical bitset is another option
- Fast index lookup
  - We know the keys at compile-time and can optimize our lookup algorithm based on that data

Predicate Structure

We need to manipulate the callback predicates:

- Extract specific requirements on indexed fields.
  - For example: The “type” field must be equal to `pkt_t::mrd32`. These requirements need to be added to the indices.
- Separate matchers on indexed vs. non-indexed fields.

Extracting some sort of table of matchers from a predicate would be ideal.

Extracting a table

```
(
  "type"_f == constant<pkt_t::mrd32>
) and (
  (
    "page"_f == 0_c and
    "tc"_f == 1_c
  ) or "page"_f == 1_c
)
```

Indexed		*Non-indexed*
type	page	tc

Extracting a table

```
(
  "type"_f == constant<pkt_t::mrd32>
) and (
  (
    "page"_f == 0_c and
    "tc"_f == 1_c
  ) or "page"_f == 1_c
)
```

Indexed		*Non-indexed*
type	page	tc
pkt_t::mrd32		
pkt_t::mrd32		

Extracting a table

```
(
  "type"_f == constant<pkt_t::mrd32>
) and (
  (
    "page"_f == 0_c and
    "tc"_f == 1_c
  ) or "page"_f == 1_c
)
```

Indexed		*Non-indexed*
type	page	tc
pkt_t::mrd32	0	
pkt_t::mrd32	1	

Extracting a table

```
(
  "type"_f == constant<pkt_t::mrd32>
) and (
  (
    "page"_f == 0_c and
    "tc"_f == 1_c
  ) or "page"_f == 1_c
)
```

Indexed		Non-indexed
type	page	tc
pkt_t::mrd32	0	1
pkt_t::mrd32	1	-

OR

Indexed		*Non-indexed*
type	page	tc
pkt_t::mrd32	0	1
pkt_t::mrd32	1	-

AND

```
(
    "type"_f == constant<pkt_t::mrd32> and
    "page"_f == 0_c and
    "tc"_f == 1_c
) or (
    "type"_f == constant<pkt_t::mrd32> and
    "page"_f == 1_c
)
```

Indexed		*Non-indexed*
type	page	tc
pkt_t::mrd32	0	1
pkt_t::mrd32	1	-

Sum of products form

Any Boolean formula can be converted to sum of products form.

I remember this term and what it is, I know it can be done with **Math**,
We will come back to this.

Full transform example

```
constexpr auto mrd_page_a_cb =
    msg::callback<
        "mrd page a and b", mrd32_defn
    >(
        (
            "page"_f == 0_c and
            "tc"_f == 1_c
        ) or "page"_f == 1_c
        [](auto msg){
            // ...
        }
    );
```

We have an algorithm, and our message callbacks have the information we need.

Let’s run through the transforms to get the predicate into a form we can index.

Example...

```
constexpr auto mrd_page_a_cb =
    msg::callback<"mrd page a and b", mrd32_defn>(
        (
            "page"_f == 0_c and
            "tc"_f == 1_c
        ) or "page"_f == 1_c
        [](auto msg){
            // ...
        }
    );
```

Combine msg requirements with predicate

```
(
    msg_requirements
) and (
    (
        "page"_f == 0_c and
        "tc"_f == 1_c
    ) or "page"_f == 1_c
)
```

Combine msg requirements with predicate

```
(
    "type"_f == constant<pkt_t::mrd32>
) and (
    (
        "page"_f == 0_c and
        "tc"_f == 1_c
    ) or "page"_f == 1_c
)
```

Convert to sum of products form

```
(
    "type"_f == constant<pkt_t::mrd32> and
    "page"_f == 0_c and
    "tc"_f == 1_c
) or (
    "type"_f == constant<pkt_t::mrd32> and
    "page"_f == 1_c
)
```

Make each product term it’s own callback

```
msg::callback<...>(
    "type"_f == constant<pkt_t::mrd32> and
    "page"_f == 0_c and
    "tc"_f == 1_c
    callback_function
)

msg::callback<...>(
    "type"_f == constant<pkt_t::mrd32> and
    "page"_f == 1_c,
    callback_function
)
```

Add evaluation for unindexed fields

```
msg::callback<...>(
    "type"_f == constant<pkt_t::mrd32> and
    "page"_f == 0_c and
    "tc"_f == 1_c,
    callback_function
)
```

Add evaluation for unindexed fields

```
msg::callback<...>(
    "type"_f == constant<pkt_t::mrd32> and
    "page"_f == 0_c
    [](auto msg){
        if (("tc"_f == 1_c)(msg))
            callback_function(msg);
    }
)
```

Extract indexed field requirements

Type

key	value
mrd32	cb_0, cb_1
mwr32	…
iord	…
…	…

Page

key	value
0	cb_0
1	cb_1
2	…
…	…

```
msg::callback<...>(
    "type"_f == constant<pkt_t::mrd32> and
    "page"_f == 0_c,
    cb_0 = [](auto msg){
        if (("tc"_f == 1_c)(msg))
            callback_function(msg);
    }
)

msg::callback<...>(
    "type"_f == constant<pkt_t::mrd32> and
    "page"_f == 1_c,
    cb_1 = callback_function
)
```

Fill in indices for all callbacks...

Type Field Index

key	callbacks
mrd32	cb_0, cb_1, cb_2
mwr32	cb_3
iord	-
iowr	-

Page Field Index

key	callbacks
0	cb_0
1	cb_1
2	cb_2, cb_3
3	-

Awesome!

Now how do I get the library to convert an arbitrary Boolean expression into sum of products form?

“Hey Ben...”

lukevalenty commented on Sep 19, 2023 Member ...

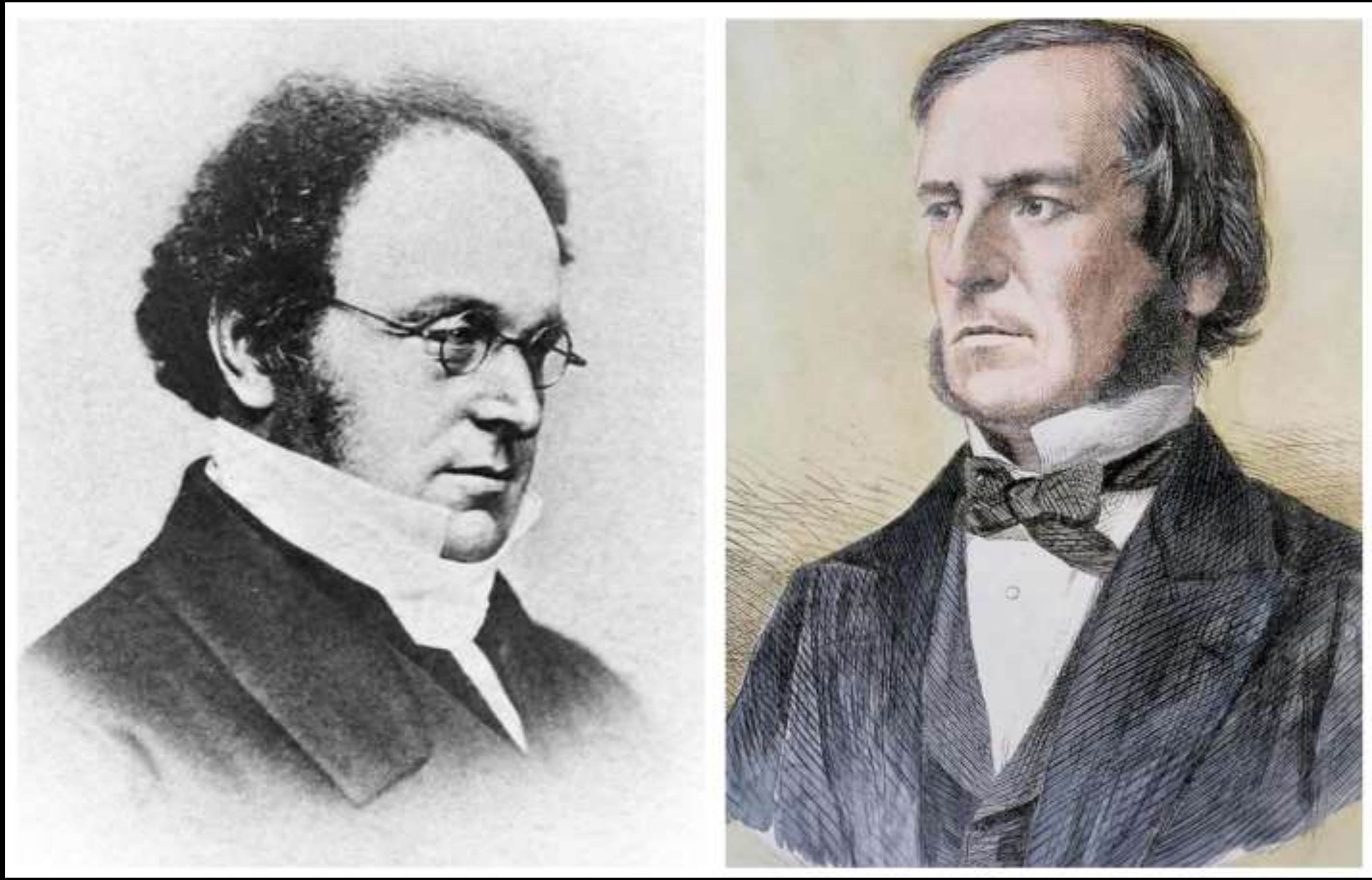
Once #373 is implemented, it will be helpful for arbitrary matcher expressions to automatically be factored into "sum of product" expressions for efficient indexed lookup.

lukevalenty assigned **elbeno** now

Math

Ben uses math for great good

Giants



We stand on their shoulders.
(Augustus de Morgan & George Boole)

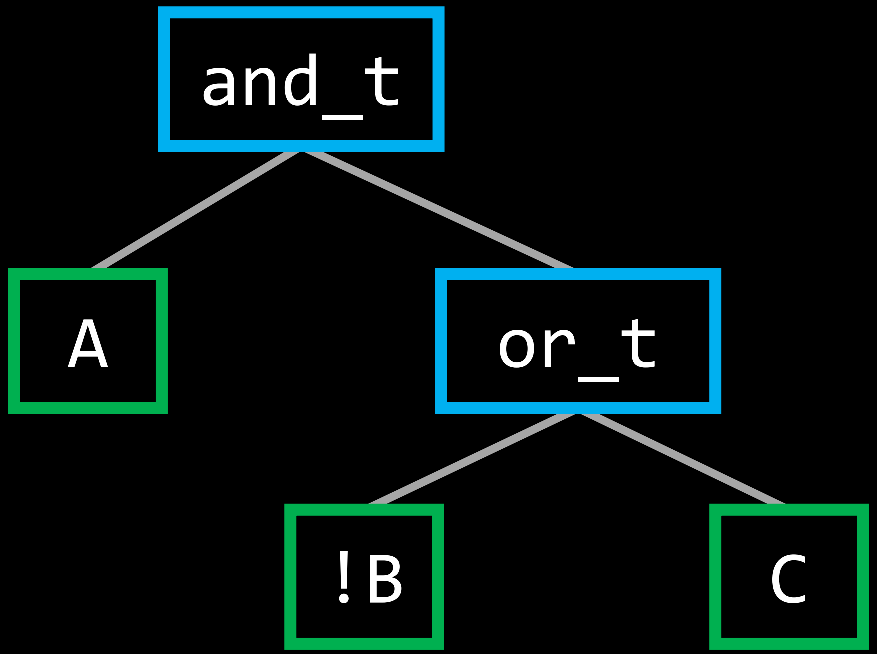
Ben Deane

Revisting matcher's Boolean operators

```
template <matcher L, matcher R>
auto operator and(L const & lhs, R const & rhs) {
    return and_t{lhs, rhs};
}

template <matcher L, matcher R>
auto operator or(L const & lhs, R const & rhs) {
    return or_t{lhs, rhs};
}

template <matcher M>
auto operator not(M const & m) {
    return not_t{m};
}
```

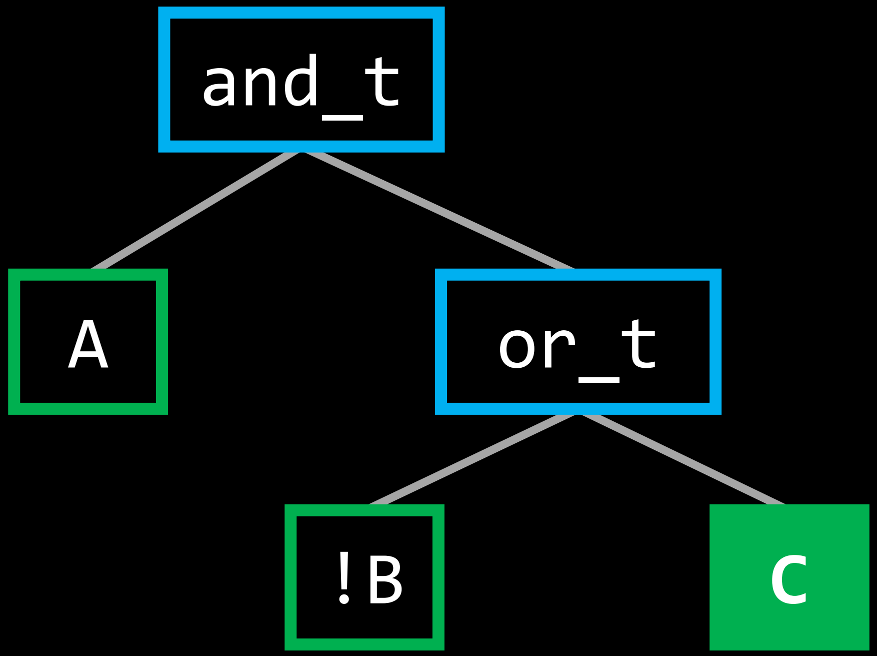
Operator overloads for and, or and not on matchers.

OK, now what?



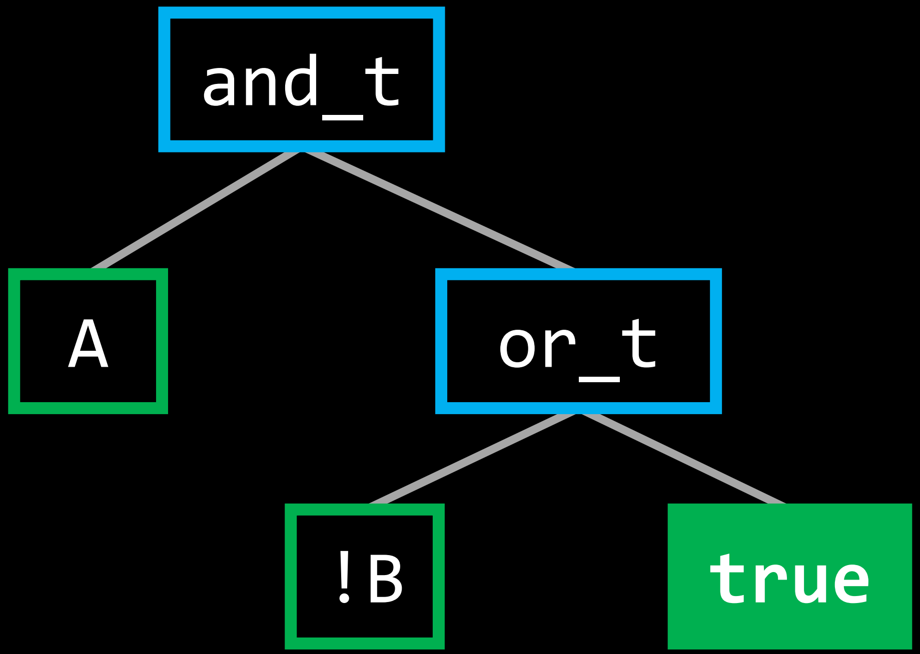
We have an expression template representing a combination of matchers that can be run on a message.

OK, now what?

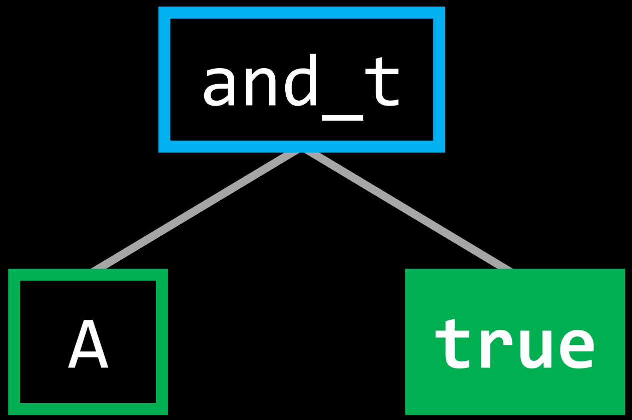


Sometimes, we know before evaluation that a matcher is going to be true. In a given code path this could be known at compile time.

OK, now what?



OK, now what?



OK, now what?

$\boxed{A}$

Simplify and transform

We can reason about how to simplify and transform the Boolean expressions, but how do we implement this in C++?

What does the library need to know about matchers in order to simplify them? How can it know?

Example matcher type problems

How does the library know how to simplify things like:

```
less_than<5> or greater_equal<5>
less_than<5> and less_than<7>
not less_than<5> and greater_equal<5>
```

The message callback interface accepts arbitrary Boolean expressions. We made it easy for complex and potentially contradicting expressions.

Example matcher type problems

How does the library know how to simplify things like:

```
less_than<5> or greater_equal<5>
less_than<5> and less_than<7>
not less_than<5> and greater_equal<5>
```

The message callback interface accepts arbitrary Boolean expressions. We made it easy for complex and potentially contradicting expressions.

Let’s start easy

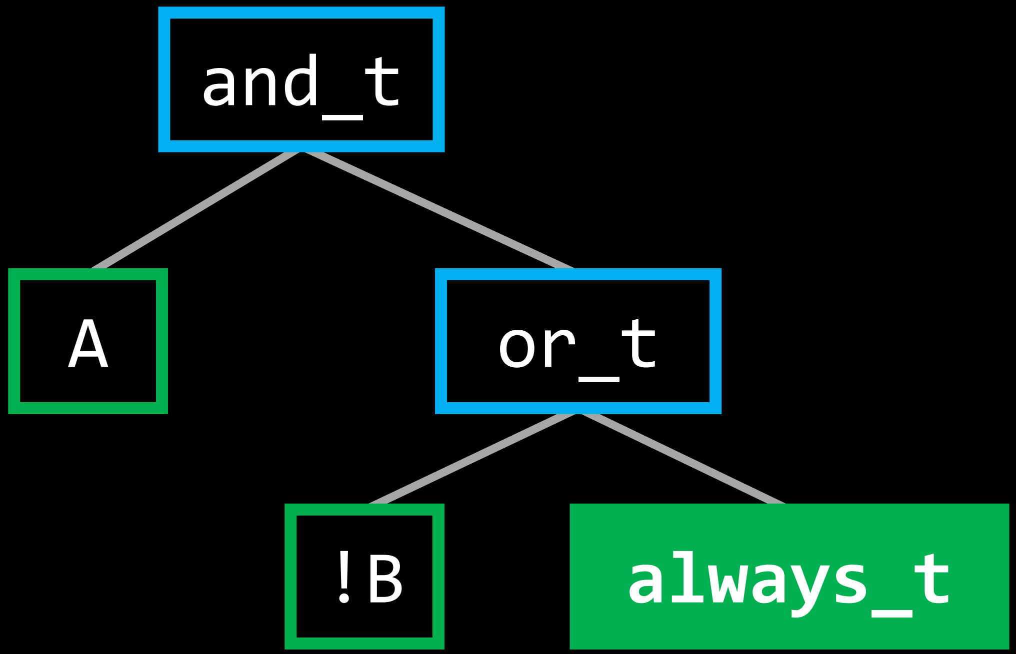
Let’s take a step back and define two matchers.

```
struct always_t {
    auto operator()(auto const &) const -> bool {
        return true;
    }
} always{};

struct never_t {
    auto operator()(auto const &) const -> bool {
        return false;
    }
} never{};
```

These seem like they will be useful. When in doubt, start with the basics.

Ben Deane

If we can simplify



If we know before evaluation that C is `true`, we can replace it with an `always_t`.

Let’s write a simplify function

We’ll make simplify a customization point that each matcher type can overload according to its own needs.

```
template <matcher M>
auto simplify(M const & m) -> M {
    return m;
}
```

The default simplification is no simplification. Just the identity function.

Simplification – and

Now that we have `always_t` and `never_t`, we can simplify `and_t` and `or_t`.

```
template <matcher L, matcher R>
auto simplify(and_t<L, R> const & m) {
    auto l = simplify(m.lhs);
    auto r = simplify(m.rhs);
    if constexpr (/* l is a never_t or r is a never_t */) {
        return never;
    } else if constexpr (/* r is an always_t */) {
        return l;
    } else if constexpr (/* l is an always_t */) {
        return r;
    } else {
        return and_t{l, r};
    }
}
```

Ben Deane

Simplification – `or`

Now that we have `always_t` and `never_t`, we can simplify `and_t` and `or_t`.

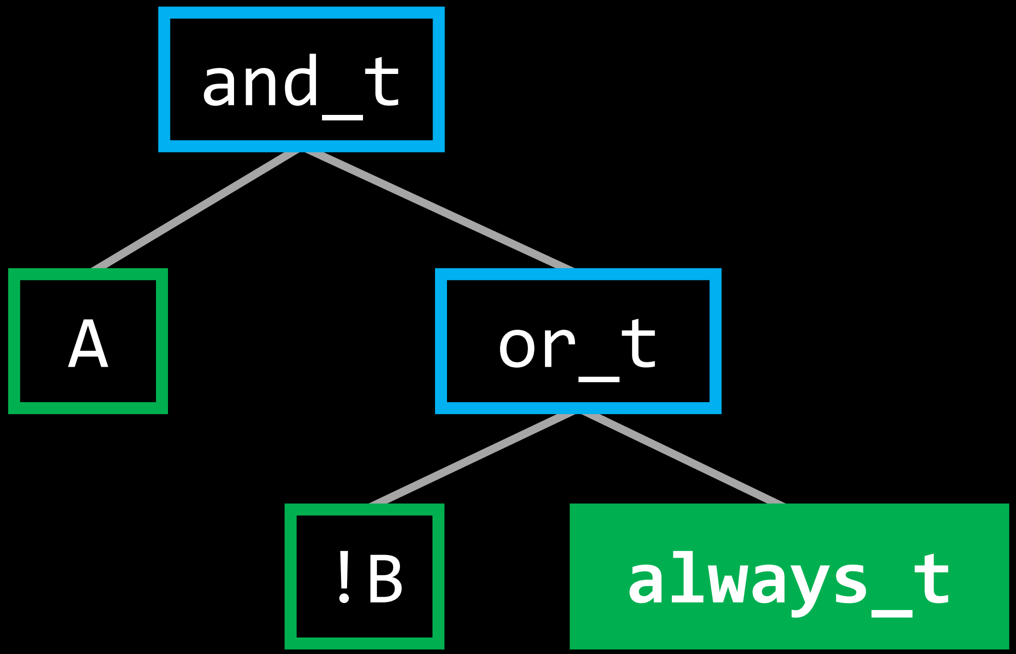
```
template <matcher L, matcher R>
auto simplify(or_t<L, R> const & m) {
    auto l = simplify(m.lhs);
    auto r = simplify(m.rhs);
    if constexpr (/* l is an always_t or r is an always_t */) {
        return always;
    } else if constexpr (/* r is a never_t */) {
        return l;
    } else if constexpr (/* l is a never_t */) {
        return r;
    } else {
        return or_t{l, r};
    }
}
```

youtu.be/H6GQUg5JquU?t=1864

Aside: Ben talks *much* more about simplification of inequality operators and implication in his Pure Virtual C++ talk “Message Handling with Boolean Implication”

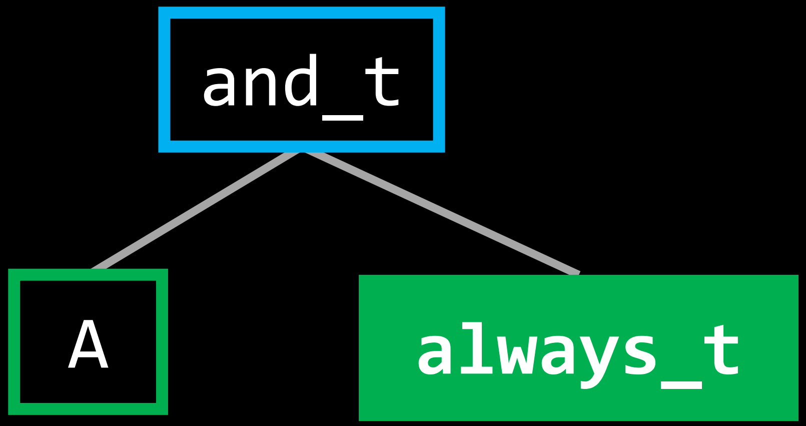
Watch it, it is *excellent*!

Now our library knows how to simplify



Ben Deane

Now our library knows how to simplify



Now our library knows how to simplify

$\boxed{A}$

Disjunctive Normal Form

(a.k.a. sum of products, `or` of `and`s)

A Boolean expression is in disjunctive normal form when it is a disjunction (`or`) of conjunctions (`and`s) or single terms.

- There are no `or` terms inside of `and` terms
- Any `not`s only apply to single terms

(This is what we need for the indexed message dispatch!)

Ben Deane

Disjunctive Normal Form

For example:

Expression	DNF?
`(A and !B and !C) or (!D and E and F)`	Yes
`(A and B) or C`	Yes
`!(A or B)`	No
`A and (B or (C and D))`	No

The transformation

To transform an expression into this form, we will recursively apply two transformations where necessary.

Distributive law:

```
A and (B or C) -> (A and B) or (A and C)
```

De Morgan’s laws:

```
!(A and B) -> !A  or !B
!(A  or B) -> !A and !B
```

Ben Deane

Sum of Products

The actual customization points for simplify and sum of products are using tag invoke...but I will choose direct invocation for slideware.

```
template <matcher M>
auto sum_of_products(M const & m) -> M {
    return m;
}
```

OK, so we just need to implement the transformation overloads for `and_t`, `or_t`, and `not_t`.

Ben Deane

Sum of Products: `not`

```
template <matcher M>
auto sum_of_products(not_t<M> const & n) {
    if constexpr (/* M is an and_t */) {
        return or_t{
            sum_of_products(negate(n.m.lhs)),
            sum_of_products(negate(n.m.rhs))};
    } else if constexpr (/* M is an or_t */) {
        return sum_of_products(and_t{
            sum_of_products(negate(n.m.lhs)),
            sum_of_products(negate(n.m.rhs))});
    } else {
        return n;
    }
}
```

Looks like De Morgan's laws to me.

Notice the extra call to `sum_of_products` in the case where we make an `and_t`: there might still be an `or` inside it.

Ben Deane

Sum of Products: or

```
template <matcher L, matcher R>
auto sum_of_products(or_t<L, R> const & m) {
    auto l = sum_of_products(m.lhs);
    auto r = sum_of_products(m.rhs);
    return or_t{l, r};
}
```

Almost nothing to do here; just recursively apply.

youtu.be/RVsJ3bGDCrM

But wait there's more! Ben has another talk he presented at the Denver C++ Meetup where he goes into detail on how to remove terms that we don't need and much more detail in everything.

Fast Lookup

Computer scieeeeeeeeeeeeeeeeeeeeeeeeence!

Lookup algorithms

Entire algorithm hinges on *fast* lookup algorithms.

Luckily there are a lot of resources on the topic and algorithms can be tested and benchmarked.

Survey of options

1. Simple linear search
   1. `std::find_if`
2. Binary search
   1. `std::map`
   2. `frozen::map`
   3. `std::binary_search`
3. Hash Lookup
   1. `std::unordered_map`
   2. `frozen::unordered_map`
   3. `Roll your own`

std::map implementations are not constexpr in C++20.

Survey of options

1. Simple linear search
   1. `std::find_if`
2. Binary search
   1. ~~`std::map`~~
   2. `frozen::map`
   3. `std::binary_search`
3. Hash Lookup
   1. ~~`std::unordered_map`~~
   2. `frozen::unordered_map`
   3. Roll your own

Linear search is great for small numbers of elements.

Binary search is actually kinda slow if you don’t need ordering.

Hash lookup algorithms should be the best option for most lookups.

github.com/serge-sans-paille/frozen

README   Apache-2.0 license

Frozen

image

Header-only library that provides 0 cost initialization for immutable containers, fixed-size containers, and various algorithms.

Frozen provides:

- immutable (a.k.a. frozen), `constexpr` -compatible versions of `std::set`, `std::unordered_set`, `std::map` and `std::unordered_map`.
- fixed-capacity, `constinit` -compatible versions of `std::map` and `std::unordered_map` with immutable, compile-time selected keys mapped to mutable values.
- 0-cost initialization version of `std::search` for frozen needles using Boyer-Moore or Knuth-Morris-Pratt algorithms.

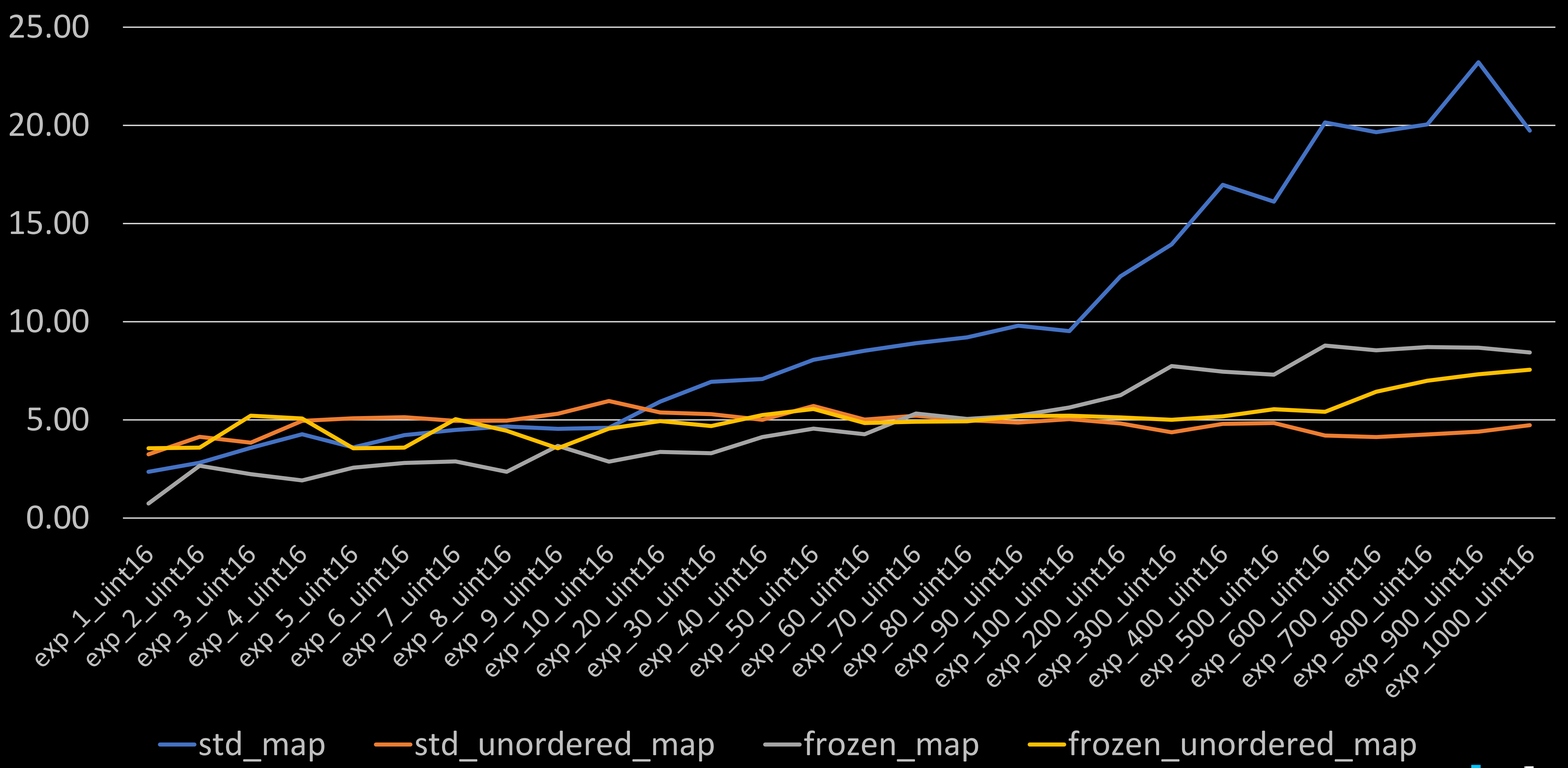
github.com/serge-sans-paille/**frozen**

The `unordered_*` containers are guaranteed *perfect* (a.k.a. no hash collision) and the extra storage is linear with respect to the number of keys.

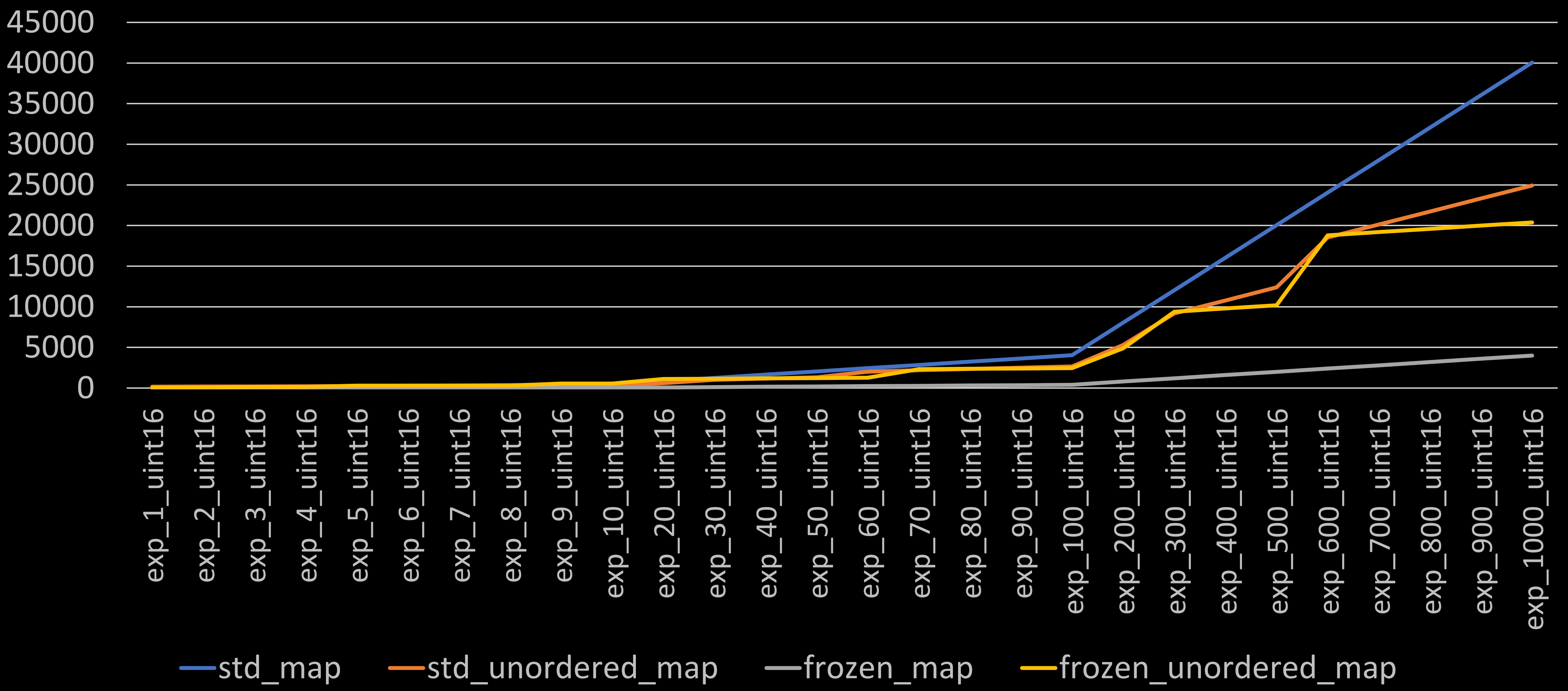
Once initialized, the container keys cannot be updated, and in exchange, lookups are faster. And initialization is free when `constexpr` or `constinit` is used :-).

Minimal perfect hashing looks promising. We are highly constrained in memory and desire predictable memory usage.

Latency (ns/lookup) vs. Number of entries



Size (bytes) vs. Number of entries



Frozen it is! Right?

`frozen::unordered_map` size and performance look great on big CPU cores.

However, performance on our embedded core was not going to be good enough. In fact, even the size was larger than desired.

Then I happened upon Kris Jusiak’s boost-ext/mph library...

github.com/boost-ext/**mph**

README

license boost | version 2.0.1 | build | try it online

Perfect hash library

> https://en.wikipedia.org/wiki/Perfect_hash_function

Use case

> Given a list of N keys (known at compile-time) find a perfect hash (map keys to values).

Features

- Single header (https://raw.githubusercontent.com/boost-ext/mph/main/mph)
  - Easy integration (see FAQ)
- Self verification upon include (can be disabled by `DISABLE_STATIC_ASSERT_TESTS` - see

intel.

github.com/boost-ext/**mph**

```cpp
int main(int argc, char**)
  constexpr std::array ids{
    std::pair{ 54u,  91u},
    std::pair{324u,  54u},
    std::pair{ 64u, 324u},
    std::pair{234u,  64u},
    std::pair{ 91u, 234u},
  };

  static_assert(0u   == mph::hash<ids>(0u));
  static_assert(91u  == mph::hash<ids>(54u));
  static_assert(54u  == mph::hash<ids>(324u));
  static_assert(324u == mph::hash<ids>(64u));
  static_assert(64u  == mph::hash<ids>(234u));
  static_assert(234u == mph::hash<ids>(91u));

  return mph::hash<ids>(argc);
}
```

```asm
main(int): // g++ -DNDEBUG -std=c++20 -O3 -march=skylake
  movl $7, %edx
  xorl %eax, %eax
  pext %edx, %edi, %edx
  movl %edx, %edx
  cmpl %edi, lut(,%rdx,8)
  cmove lut+4(,%rdx,8), %eax
  ret
```

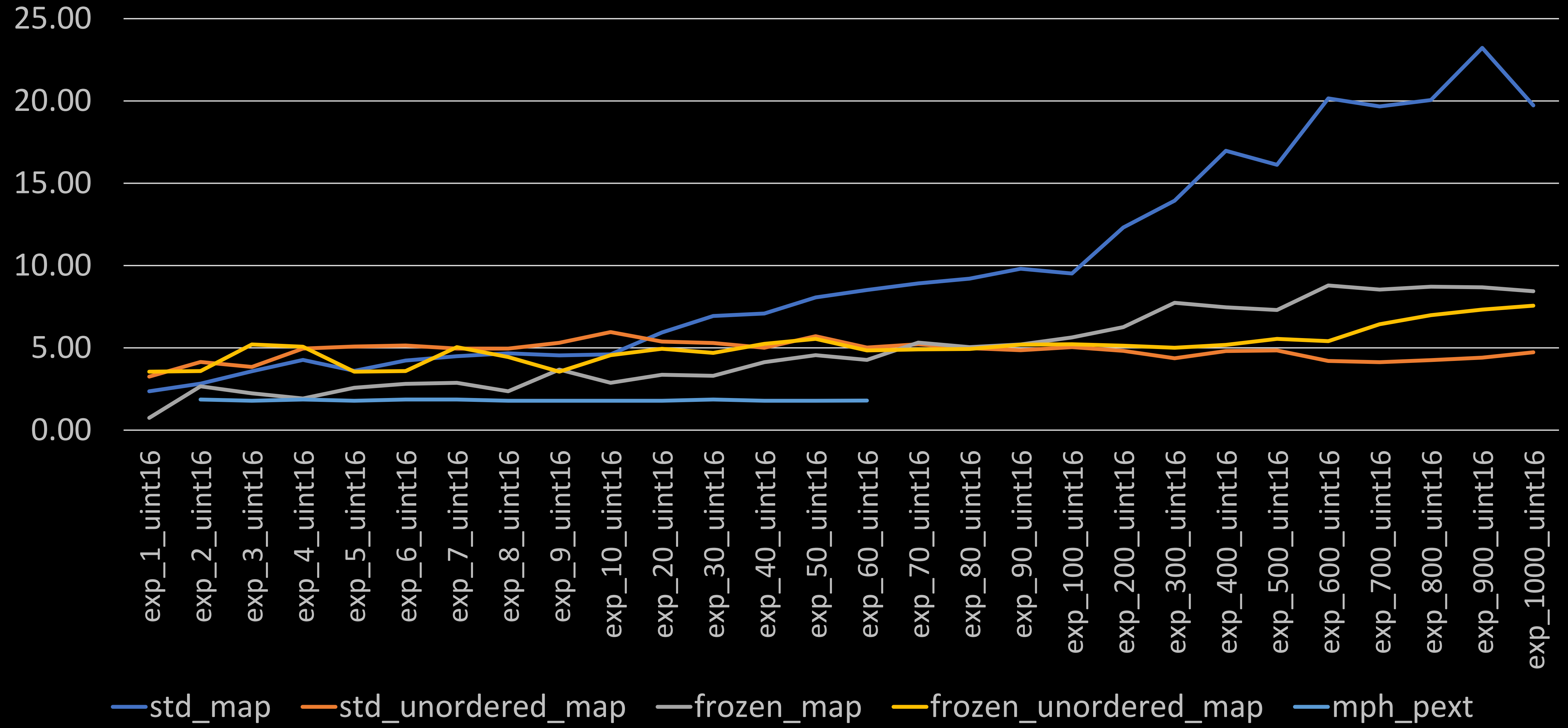
github.com/boost-ext/**mph**

```
main(int): // g++ -DNDEBUG -std=c++20 -O3 -march=skylake
  movl $7, %edx
  xorl %eax, %eax
  pext %edx, %edi, %edx
  movl %edx, %edx
  cmpl %edi, lut(,%rdx,8)
  cmove lut+4(,%rdx,8), %eax
  ret
```

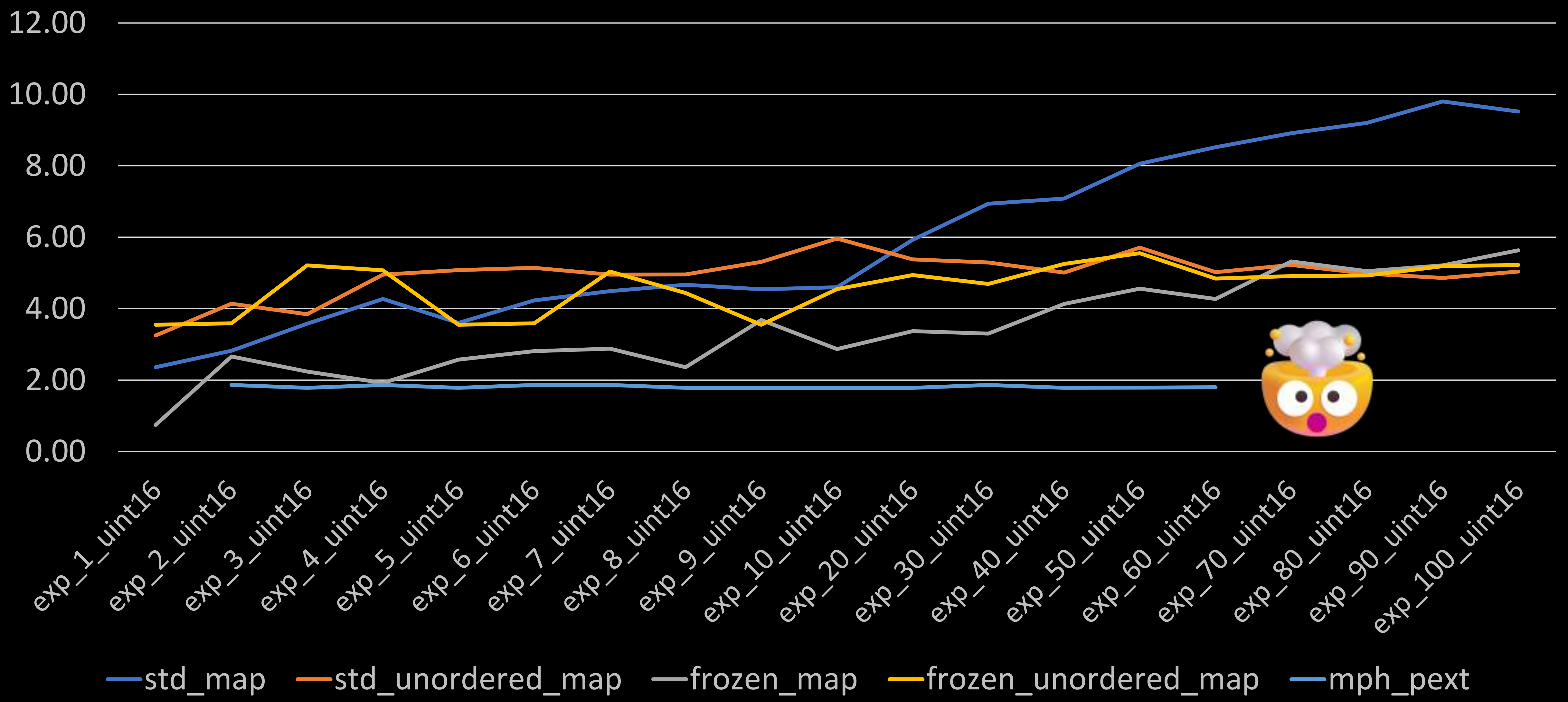
How is this so small!?

What is PEXT!?

Latency (ns/lookup) vs. Number of entries



Latency (ns/lookup) vs. Number of entries



http://0x80.pl/notesen/2023-04-30-lookup-in-strings.html

PEXT recap

The instruction `PEXT` gets two arguments: the input word and the input mask. Bits from the input word for which the input mask is 1 are copied to the output. For example:

```
word:   0010101011010111
mask:   0011100100100010
masked: __101__0__0___1_
PEXT:   __________101001
```

Ohh...pext is a specialized bit manipulation instruction that I don't have on my embedded processor. Hmm...

PEXT hashing algorithm mask generation

```
std::unsigned_integral auto find_mask(auto keys) {
    auto mask = all_bits_set();

    for (auto bit : mask) {
        auto try_mask = mask.without(bit);

        if (all_keys_unique_with_mask(keys, try_mask)) {
            mask = try_mask;
        }
    }

    return mask;
}
```

Find a mask with the least number of bits set and all keys are still unique when bitwise ANDed with mask.

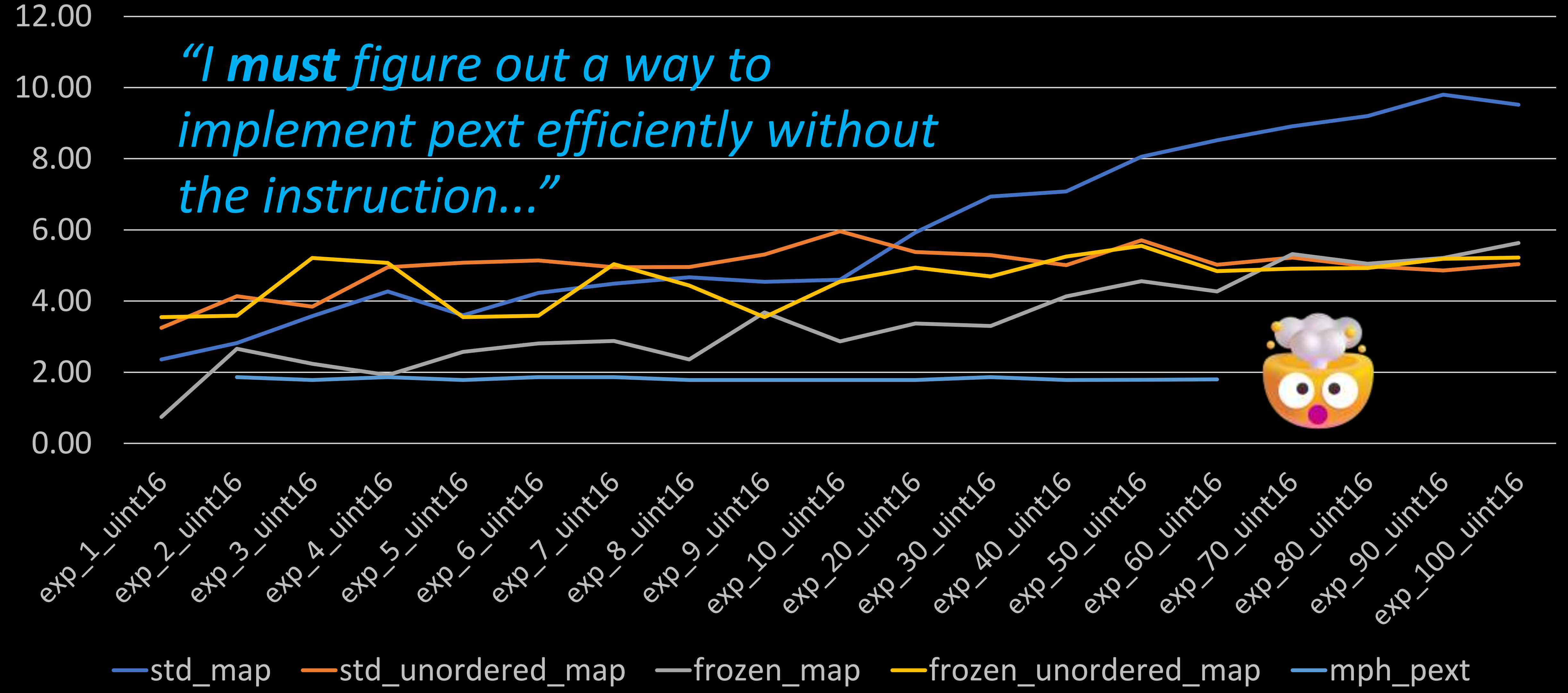
PEXT hashing algorithm table generation

```
auto gen_table(auto entries, auto mask) {
    auto table = std::array<entry_type, entries.size()>{};

    std::fill(table.begin(), table.end(), default_entry);

    for (auto entry : entries) {
        auto idx = pext(mask, entry.key);
        table[idx] = entry;
    }

    return table;
}
```

PEXT hashing algorithm lookup

```
auto lookup(key_type key) -> value_type {
    auto idx = pext(mask, key);
    auto entry = table[idx];

    if (entry.key == key) {
        return entry.value;
    } else {
        return default_value;
    }
}
```

Latency (ns/lookup) vs. Number of entries



PEXT extracts and packs the bits

```
auto pext_impl(uint32_t mask, uint32_t value) -> uint32_t {
    uint32_t result = 0u;
    auto dst = 0;
    for (auto i = 0; i < 32; i++) {
        if (mask & (1 << i)) {
            auto b = (value >> i) & 1;
            result += (b << dst++);
        }
    }
    return result;
}
```

Effectively it masks, shifts, and merges each extracted bit.

But this is too slow...

stackoverflow.com/questions/14547087

Extracting bits with a single multiplication

Asked 11 years, 3 months ago   Modified 11 years ago   Viewed 25k times

328

I saw an interesting technique used in an answer to another question, and would like to understand it a little better.

We're given an unsigned 64-bit integer, and we are interested in the following bits:

```
1.......2.......3.......4.......5.......6.......7.......8.......
```

Specifically, we'd like to move them to the top eight positions, like so:

```
12345678........................................................
```

We don't care about the value of the bits indicated by `.`, and they don't have to be preserved.

Multiply is just shifts and adds!

```
    12 *     5 =      60
0b1100 * 0b101 = 0b111100

  0b001100 = (12 << 0)
+ 0b110000 = (12 << 2)
  --------
  0b111100
```

Generic 4-bit multiplication

```
  0b0000abcd = (lhs << 0) if (rhs[0])
  0b000abcd0 = (lhs << 1) if (rhs[1])
  0b00abcd00 = (lhs << 2) if (rhs[2])
+ 0b0abcd000 = (lhs << 3) if (rhs[3])
  ----------
  0bxxxxxxxx
```

pseudo_pext algorithm

```
auto operator()(T value) -> T {
    auto interesting_bits = value & mask;
    auto packed = interesting_bits * magic_num;
    return (packed >> gap_bits) & final_mask;
}
```

Multiplication is quite fast, even on smaller embedded processors. The compiler is also very good at optimizing multiplication by certain constants.

Extract 0b1001 from a value

```
  bits = value & 0b1001
  magic_num = 5

  0b0000a00d = (bits << 0)
+ 0b00a00d00 = (bits << 2)
  ----------
  0b00a0ad0d = packed
```

Extract `0b1001` from a value (cont.)

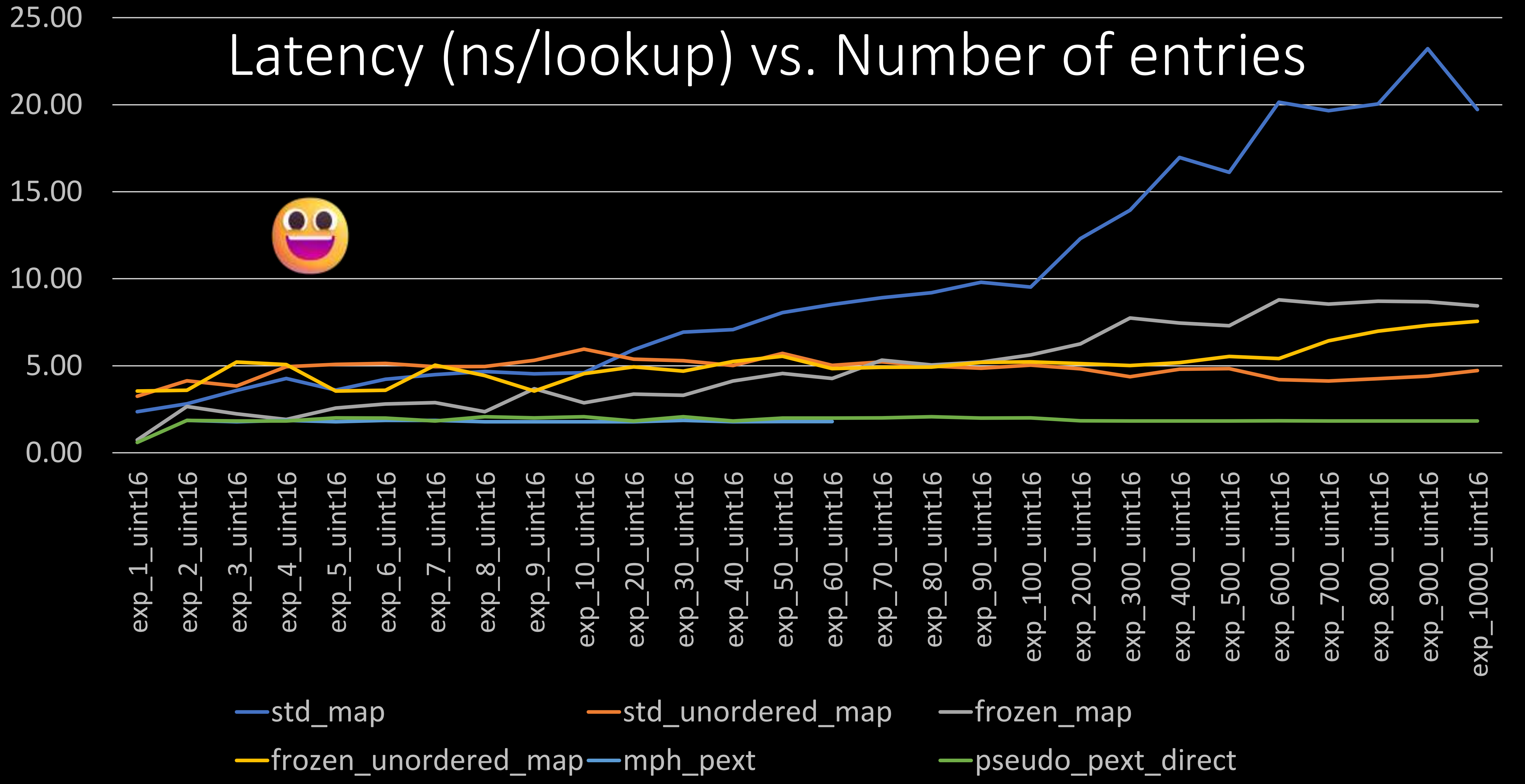
```
 packed                 = 0b00a0ad0d
 packed >> 2            =   0b00a0ad
(packed >> 2) & 0b11    =       0bad
```

Find the magic_num

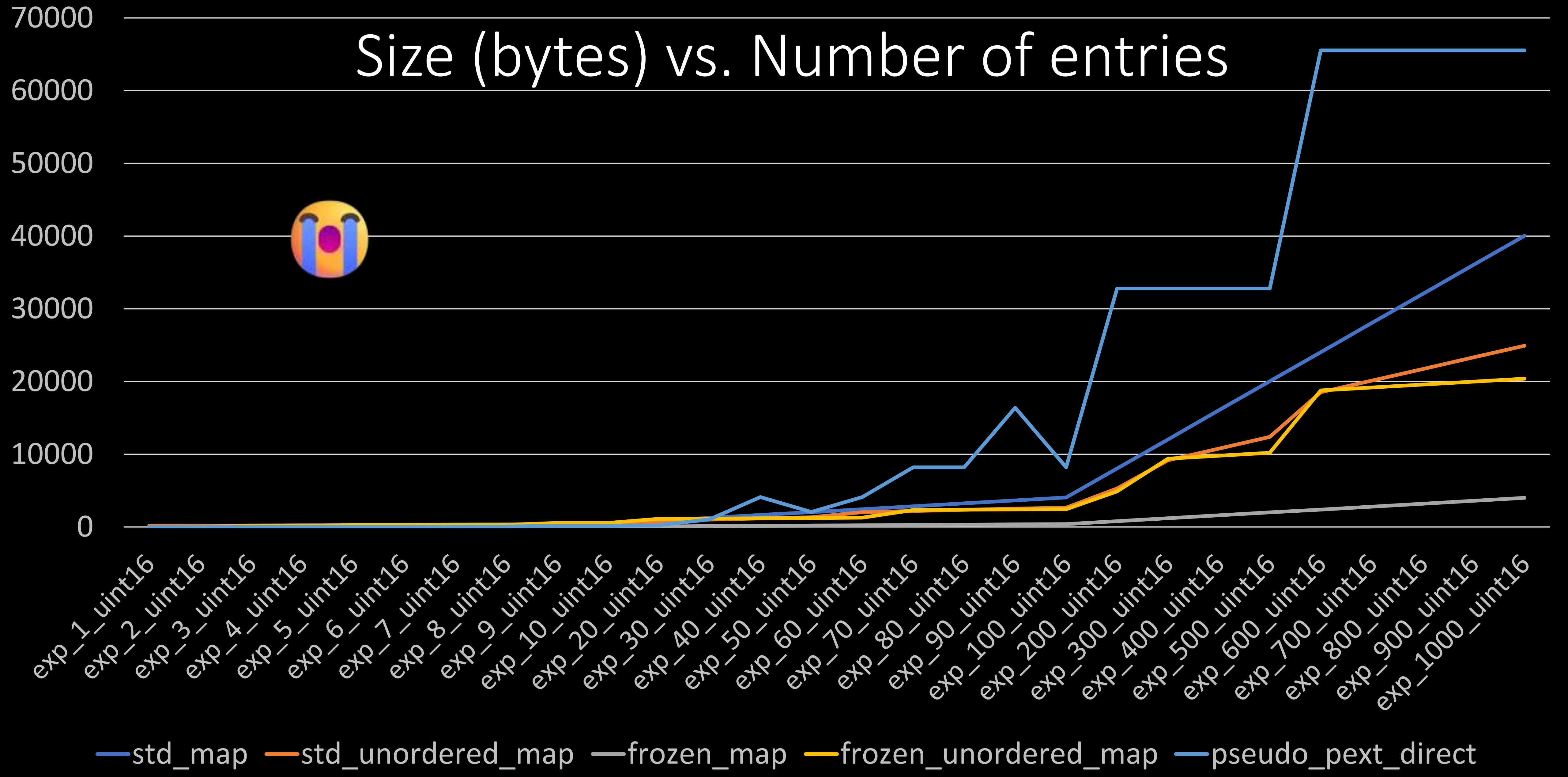
```
template <typename T>
auto compute_magic_num(std::size_t dst, T const mask) -> T {
    constexpr auto t_digits = std::numeric_limits<T>::digits;

    auto magic_num = stdx::bitset<t_digits>{};
    auto const mask_bits = stdx::bitset<t_digits>{mask};

    bool prev_src_bit_set = false;
    for (auto src = std::size_t{}; src < t_digits; src++) {
        bool const curr_src_bit_set = mask_bits[src];
        bool const new_stretch = curr_src_bit_set and not prev_src_bit_set;

        if (new_stretch) {
            magic_num.set(dst - src);
        }

        if (curr_src_bit_set) {
            dst += 1;
        }

        prev_src_bit_set = curr_src_bit_set;
    }

    return magic_num.template to<T>();
}
```

- *Find each consecutive run of '1's in the mask.*
- *Calculate how far it needs to be shifted, accumulate the shift amount to the power of 2.*
- *Add up all the shift amounts.*

This is not perfect, there can be overlap of interesting bits that causes carries...it is not exactly pext.



Latency (ns/lookup) vs. Number of entries
25.00
20.00
15.00
10.00
5.00
0.00
exp_1_uint16
exp_2_uint16
exp_3_uint16
exp_4_uint16
exp_5_uint16
exp_6_uint16
exp_7_uint16
exp_8_uint16
exp_9_uint16
exp_10_uint16
exp_20_uint16
exp_30_uint16
exp_40_uint16
exp_50_uint16
exp_60_uint16
exp_70_uint16
exp_80_uint16
exp_90_uint16
exp_100_uint16
exp_200_uint16
exp_300_uint16
exp_400_uint16
exp_500_uint16
exp_600_uint16
exp_700_uint16
exp_800_uint16
exp_900_uint16
exp_1000_uint16
std_map
std_unordered_map
frozen_map
frozen_unordered_map
mph_pext
pseudo_pext_direct

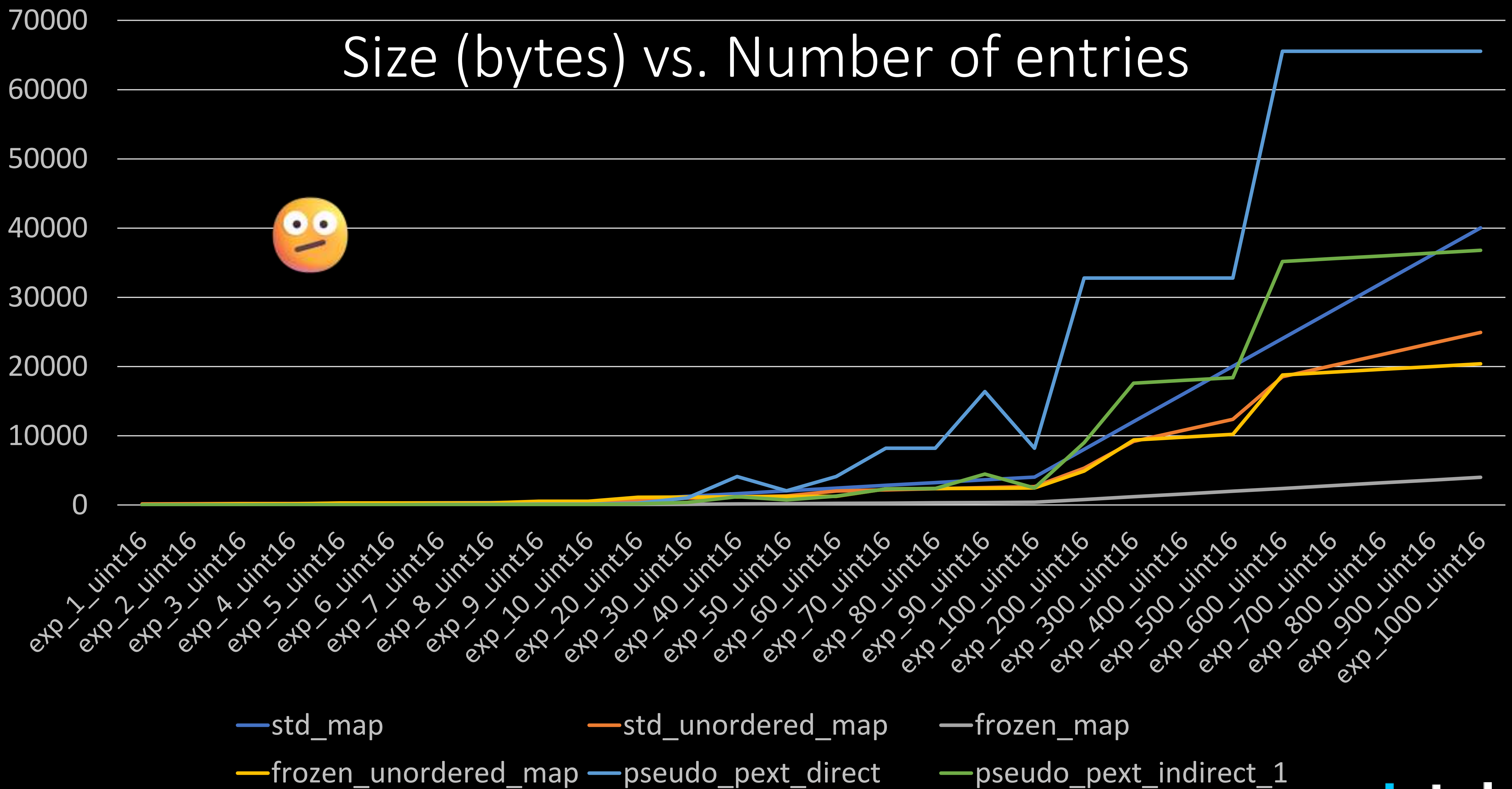

Size (bytes) vs. Number of entries
70000
60000
50000
40000
30000
20000
10000
0
exp_1_uint16
exp_2_uint16
exp_3_uint16
exp_4_uint16
exp_5_uint16
exp_6_uint16
exp_7_uint16
exp_8_uint16
exp_9_uint16
exp_10_uint16
exp_20_uint16
exp_30_uint16
exp_40_uint16
exp_50_uint16
exp_60_uint16
exp_70_uint16
exp_80_uint16
exp_90_uint16
exp_100_uint16
exp_200_uint16
exp_300_uint16
exp_400_uint16
exp_500_uint16
exp_600_uint16
exp_700_uint16
exp_800_uint16
exp_900_uint16
exp_1000_uint16
std_map
std_unordered_map
frozen_map
frozen_unordered_map
pseudo_pext_direct

So there’s a problem...

The algorithm is “perfect”, meaning there are no collisions. What if we allow for a limited number of collisions?

Additionally, the lookup table contains full entries, maybe adding level of indirection that points to a packed table of entries would be helpful.

Let’s add an indirect lookup whose values point to the entry in a table of packed entries...


Size (bytes) vs. Number of entries
70000
60000
50000
40000
30000
20000
10000
0
exp_1_uint16
exp_2_uint16
exp_3_uint16
exp_4_uint16
exp_5_uint16
exp_6_uint16
exp_7_uint16
exp_8_uint16
exp_9_uint16
exp_10_uint16
exp_20_uint16
exp_30_uint16
exp_40_uint16
exp_50_uint16
exp_60_uint16
exp_70_uint16
exp_80_uint16
exp_90_uint16
exp_100_uint16
exp_200_uint16
exp_300_uint16
exp_400_uint16
exp_500_uint16
exp_600_uint16
exp_700_uint16
exp_800_uint16
exp_900_uint16
exp_1000_uint16
std_map
std_unordered_map
frozen_map
frozen_unordered_map
pseudo_pext_direct
pseudo_pext_indirect_1

Better...

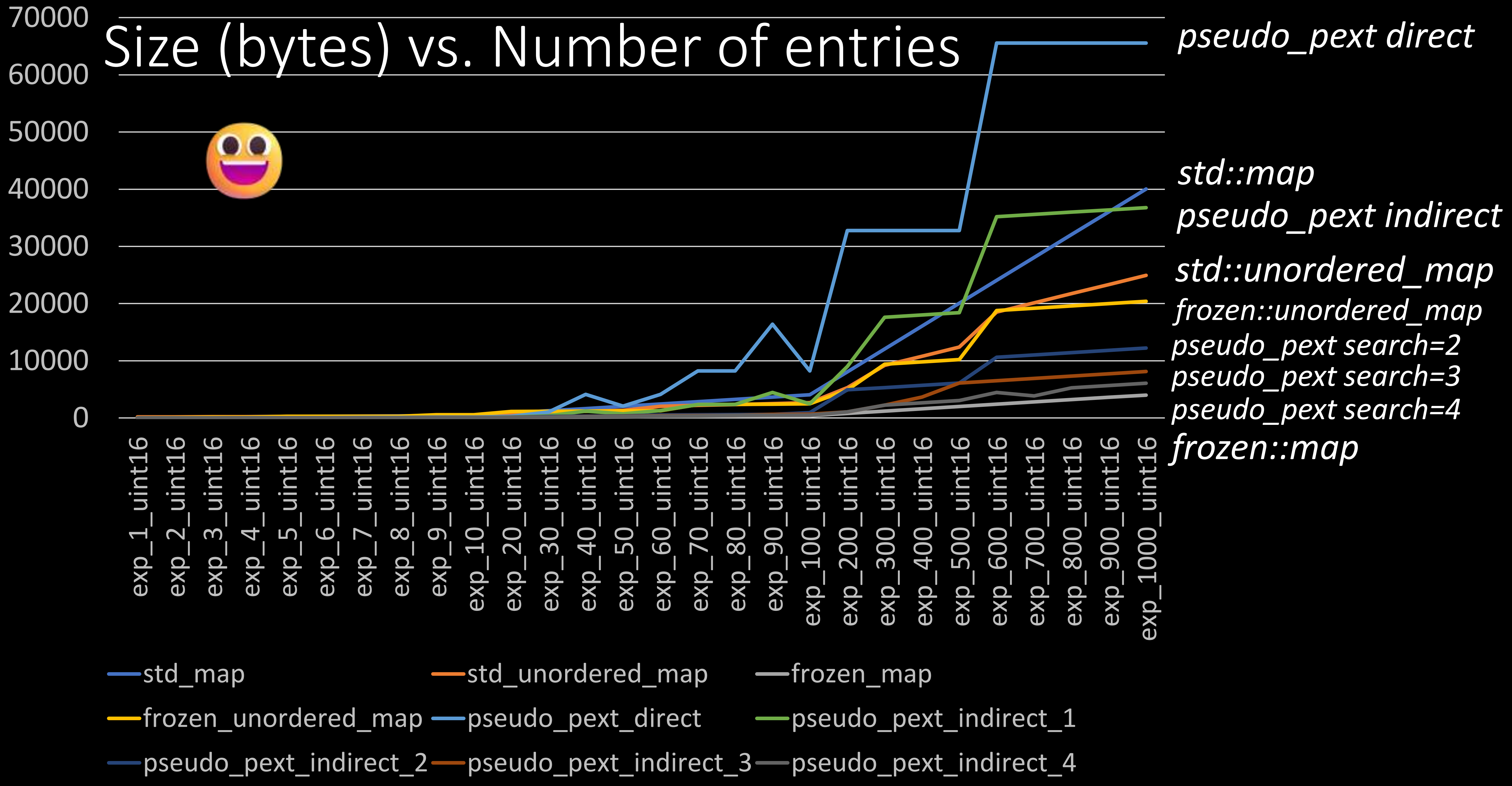
With one level of indirection we are saving the storage of many empty entries...instead storing many smaller offsets into the entry table. With the random datasets used, its storage is comparable to std::map.

If we allow for some collisions and limited search, we *should* get better memory efficiency.

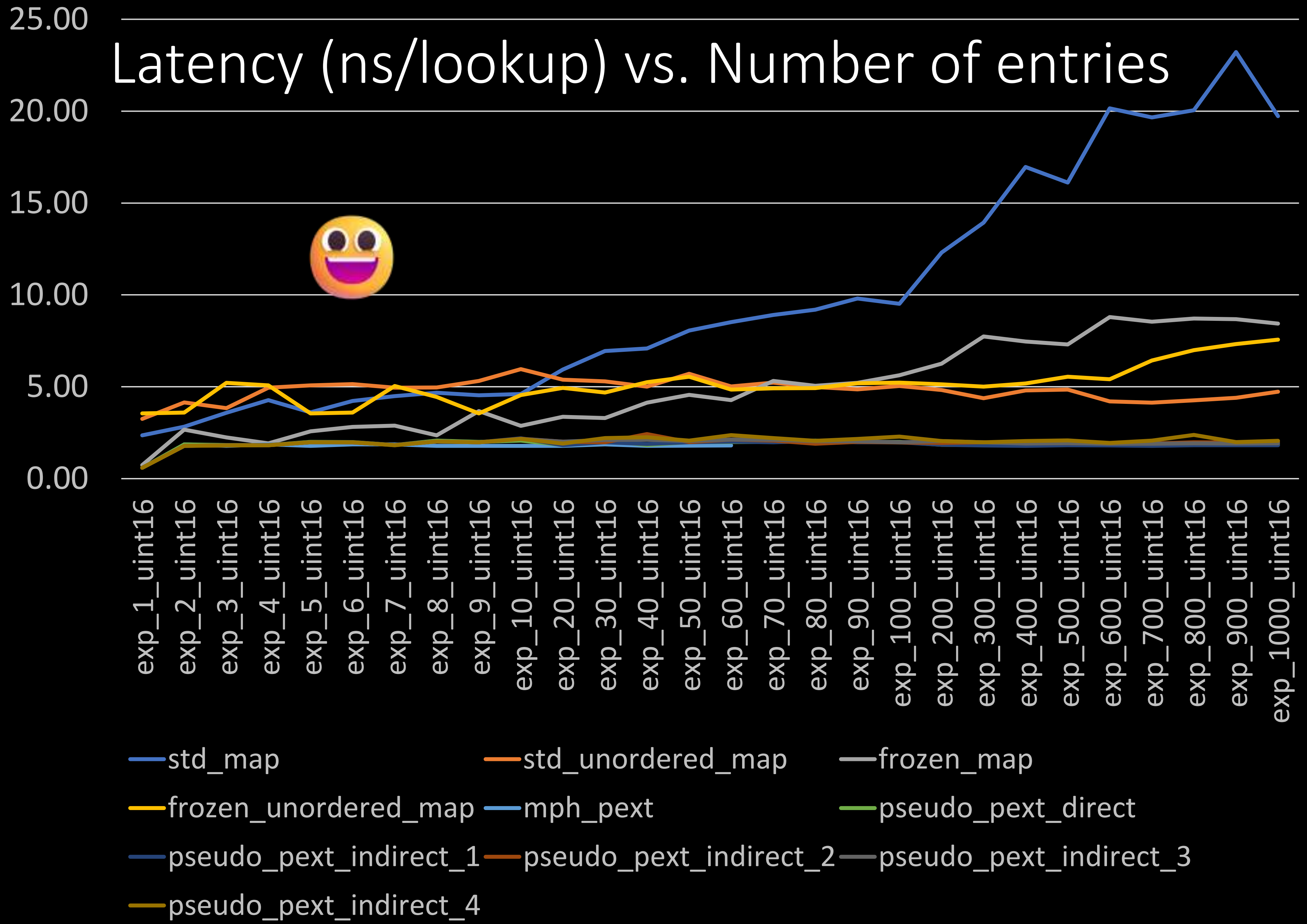
Now the mask search algorithm needs to be updated to allow for collisions.

```
auto find_mask_wcollisions(auto keys, auto search_len) {
  auto mask = find_mask(keys);

  while (search_len > 1 and popcount(mask) > 4) {
    auto bit = find_bit_with_fewest_collisions(mask, keys);
    auto try_mask = remove_bit(mask, bit);

    if (max_num_collisions(try_mask, keys) < search_len) {
      mask = try_mask;
    } else {
      return mask;
    }
  }
  return mask;
}
```

Size (bytes) vs. Number of entries



Latency (ns/lookup) vs. Number of entries



std::map

frozen::map

frozen::unordered_map

std::unordered_map

pseudo_pext indirect

pseudo_pext direct

pseudo_pext search=3

pseudo_pext search=2

pseudo_pext search=4

Analysis for 100 entries

Algorithm	Size (bytes)	Efficiency	ns/lookup
`frozen::map`	400	100.0%	5.63
`pseudo_pext indirect, search=4`	480	83.3%	2.28
`pseudo_pext indirect, search=3`	672	60.0%	1.98
`pseudo_pext indirect, search=2`	928	43.1%	1.97
`frozen::unordered_map`	2456	16.3%	5.22
`pseudo_pext indirect`	2464	16.2%	1.97
`std::unordered_map`	2672	15.0%	5.04
`std::map`	4048	9.9%	9.52
`pseudo_pext direct`	8208	4.9%	2.00

Analysis for 1000 entries

Algorithm	Size (bytes)	Efficiency	ns/lookup
`frozen::map`	4000	100.0%	8.44
`pseudo_pext indirect, search=4`	6064	66.0%	2.06
`pseudo_pext indirect, search=3`	8112	49.3%	1.92
`pseudo_pext indirect, search=2`	12208	32.8%	1.99
`frozen::unordered_map`	20392	19.6%	7.56
`std::unordered_map` ↑	24928	16.0%	4.73
`pseudo_pext indirect` ↓	36784	10.9%	1.81
`std::map`	40048	10.0%	19.73
`pseudo_pext direct`	65552	6.1%	1.83

Lookup optimization takeaways

- PEXT and pseudo PEXT algorithms let you construct a custom hash function for a given set of keys...you can control trade-offs between speed and memory efficiency
- Minimal and perfect hashing are not necessarily minimal storage or best performance!
- Applied **math** and computer **science** is both fun and useful!

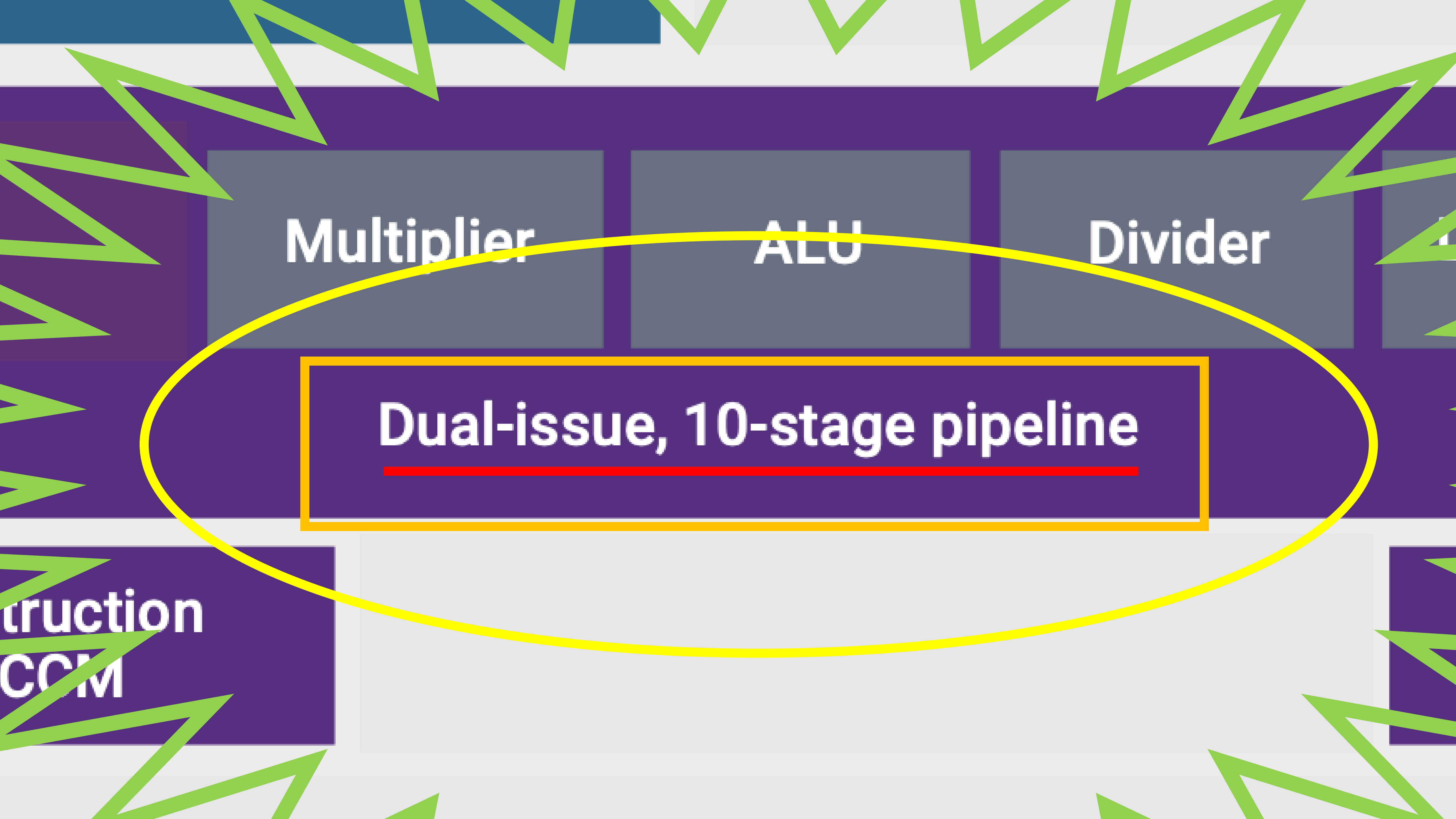
Putting it all together

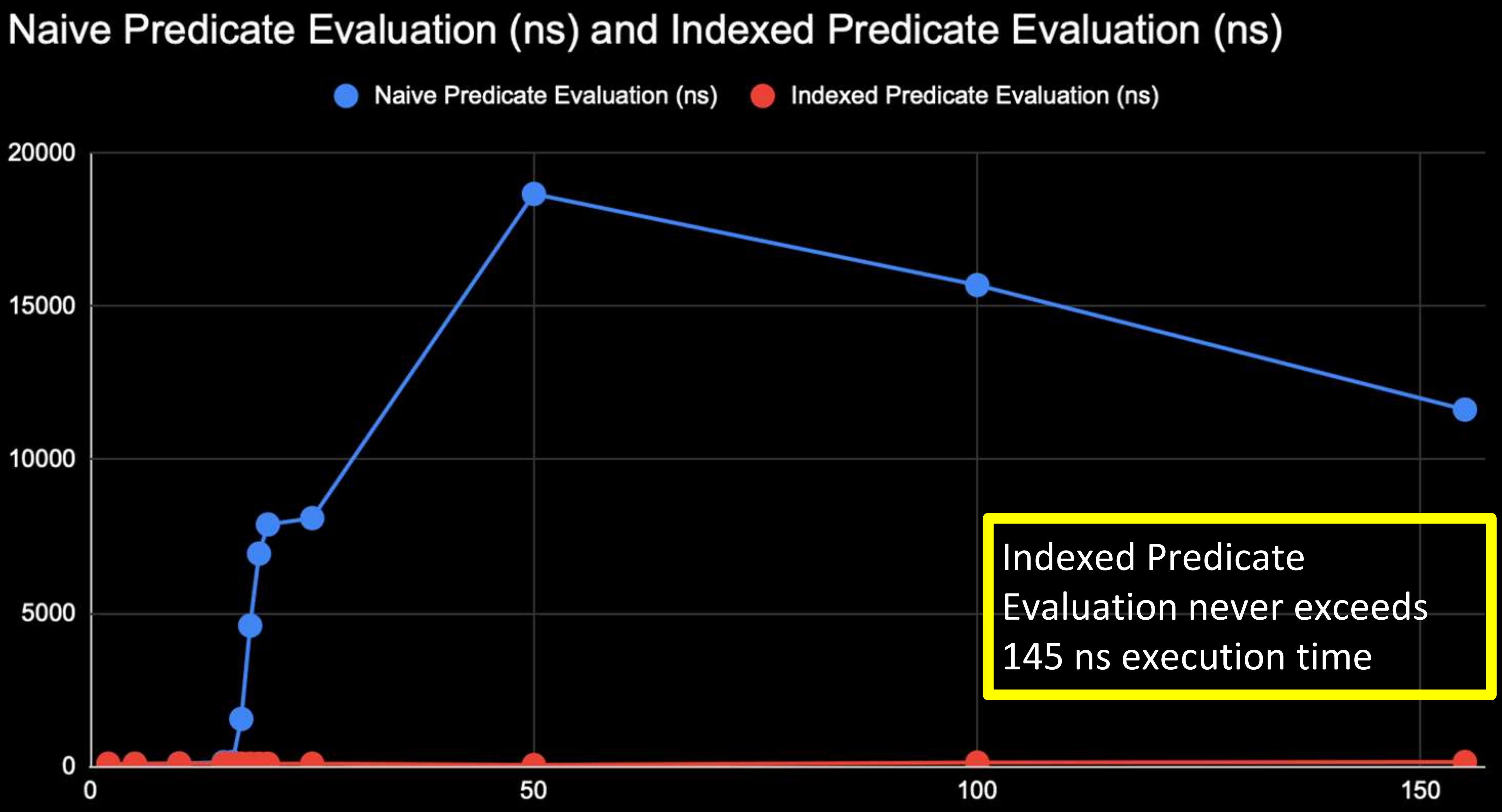
Math, computer science, and software design: three peas in a pod.

ARC HS4x Dev Board

https://www.synopsys.com/dw/ipdir.php?ds=arc-hs-development-kit

167


Multiplier
ALU
Divider
Dual-issue, 10-stage pipeline
truction
CCM



ARC HS4x CPU Core @ 400MHz

Big core, x86 Results

256 callbacks

ns/op	op/s	err%	total	benchmark
414.41	2,413,066.53	0.1%	9.90	`msgs`
20.12	49,713,138.59	0.1%	0.48	`msgs`
34.38	29,088,063.95	0.2%	0.82	`msgs`

Benchmark ran on Google Cloud n2 instance with a Cascade Lake CPU running at 2.8GHz.

Benchmark Analysis

- Good improvement, but less than expected results on large out-of-order CPU cores
  - Algorithm was designed and implementation optimized specifically for small, in-order CPUs
  - First time benchmarking full algorithm on out-of-order CPU cores
- A lot of time spent on optimizing the lookup, but the callback set representation, intersection, and iteration need more attention
  - Hierarchical bitset representation needs to be explored
  - Vectorization opportunities for faster “first bit set”
- Changes to the naïve implementation may allow compiler to better optimize across all callback predicates.
- Indexed handler stores function pointers to callbacks, may be paying high misprediction cost for indirect branches.

Wrapping up

Lessons learned...

Wrap-up

Interface design is critical:

- Depend on stable interfaces.
- Dig into the problem and coax out its fundamental concepts.
- Take the fundamentals and make a declarative interface.

When do we optimize?

- Performance requirements should guide our <u>design</u>.
- Optimize the <u>implementation</u> when you have proof it is necessary.
- Decouple interfaces from performance.

Thank you!