Effect Handlers in C++

or: stack switching… and then what?

https://github.com/maciejpirog/cpp-effects

Maciej Piróg
Huawei Research, Edinburgh
maciej.pirog@huawei.com

WebAssembly Stacks Subgroup, 21 Mar 2022

What? Why?

- A C++ library for programming with one-shot effect handlers built on top of boost.context (which provides a quite portable mechanism for stack switching)
- Goal is to provide an expressive, usable, type-safe, and composable abstraction for programming with control in C++
- Research question: Are effect handlers C++-friendly in terms of idiomaticity :), expressivity, and performance?

EH in OO

- Previous work on effect handlers in (Java-like) OO, e.g.
  - P. Inostroza, T. van der Storm. **JEff: Objects for effect**. Onward! 2018 (doi: 10.1145/3276954.3276955)
  - J.I. Brachthäuser, P. Schuster, K. Ostermann. **Effect handlers for the masses**. OOPSLA 2018 (doi: 10.1145/3276481)
- C++ is a very different kind of beast: value types, no GC, no parametric polymorphism, RAII and move-semantics for resource management, often weird template hocus-pocus as programmer-facing interface, etc.

Wishlist

Goals of the project in the order of importance:

1. Elegant programmer-level interface
2. Type-safety
3. Performance

API

Let’s define simple cooperative threads:

1. Two commands (operations):

```
struct Yield : Command<void> { };

struct Fork : Command<void> {
  std::function<void()> proc;
};
```

2. Functional interface:

```
void yield()
{
  OneShot::InvokeCmd(Yield{});
}

void fork(std::function<void()> proc)
{
  OneShot::InvokeCmd(Fork{{}, proc});
}
```

3. Handler:

```
class Scheduler : public FlatHandler<void, Yield, Fork> {
public:
  static void Start(std::function<void()> f)
  {
    Run(f);
    while (!queue.empty()) { // Round-robin scheduling
      auto resumption = std::move(queue.front());
      queue.pop_front();
      OneShot::Resume(std::move(resumption));
    }
  }
private:
  static std::list<Resumption<void, void>> queue;
  static void Run(std::function<void()> f)
  {
    OneShot::Handle<Scheduler>(f);
  }
  void CommandClause(Yield, Resumption<void, void> r) override
  {
    queue.push_back(std::move(r));
  }
  void CommandClause(Fork f, Resumption<void, void> r) override
  {
    queue.push_back(std::move(r));
    queue.push_back(OneShot::MakeResumption<void>(std::bind(Run, f.proc)));
  }
};
```

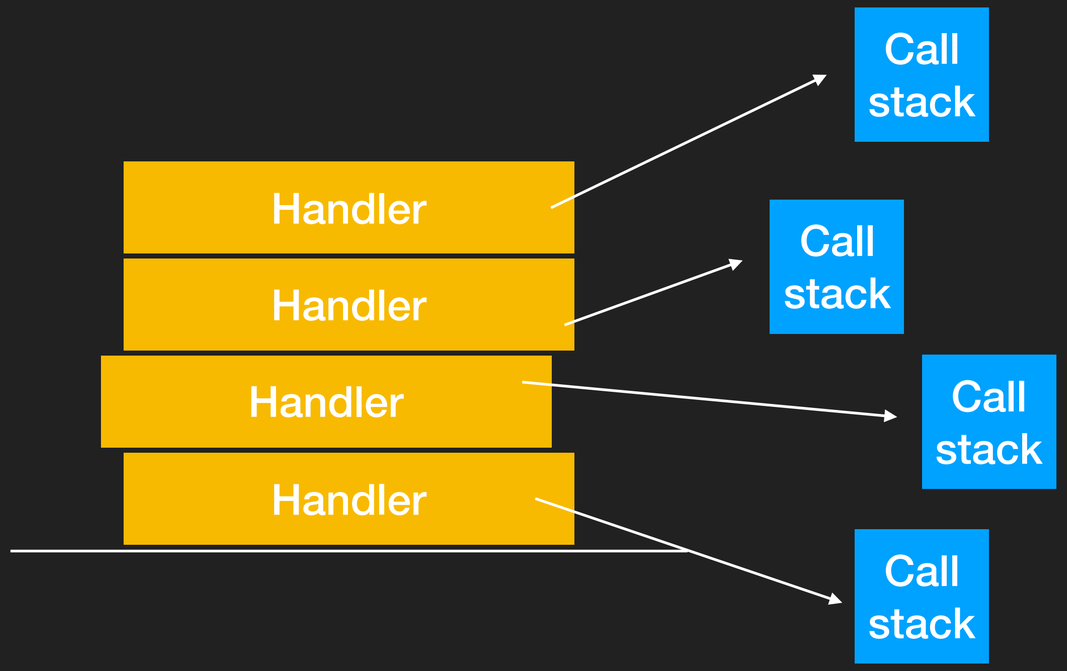
4. Example:

```
void worker(int k)
{
  for (int i = 0; i < 10; ++i) {
    std::cout << k;
    yield();
  }
}

void starter()
{
  for (int i = 0; i < 5; ++i) {
    fork(std::bind(worker, i));
  }
}

int main()
{
  Scheduler::Start(starter);

  // Output:
  // 0102103210432104321043210432104321043210432104321432434
}
```

- Traditionally, objects encapsulate **state** and **behaviour**.
- Here, they encapsulate **state**, **behaviour**, and **control**.
- EH are known to be very expressive, but safety is also important:
  - Type-safety of commands and resumptions,
  - Effects encapsulated by handlers: bare resumptions rarely leave handlers and wander around programs,
  - No worries about lifetimes of handlers/resumptions (unless the programmer wants to manage them manually),
  - The API plays nicely with the usual C++ idioms: RTTI (or tags) to select handler, overloading to select command clause, move semantics for managing resources (resumptions).

Implementation

The implementation uses a global metastack (a stack of handlers)



Call stack - boost.context-allocated call stack

Handler - A “metaframe”

But what is this metaframe? (new Facebook product?)

Metaframes

- Metaframe is simply a (pointer to an) object of a class derived from Handler:

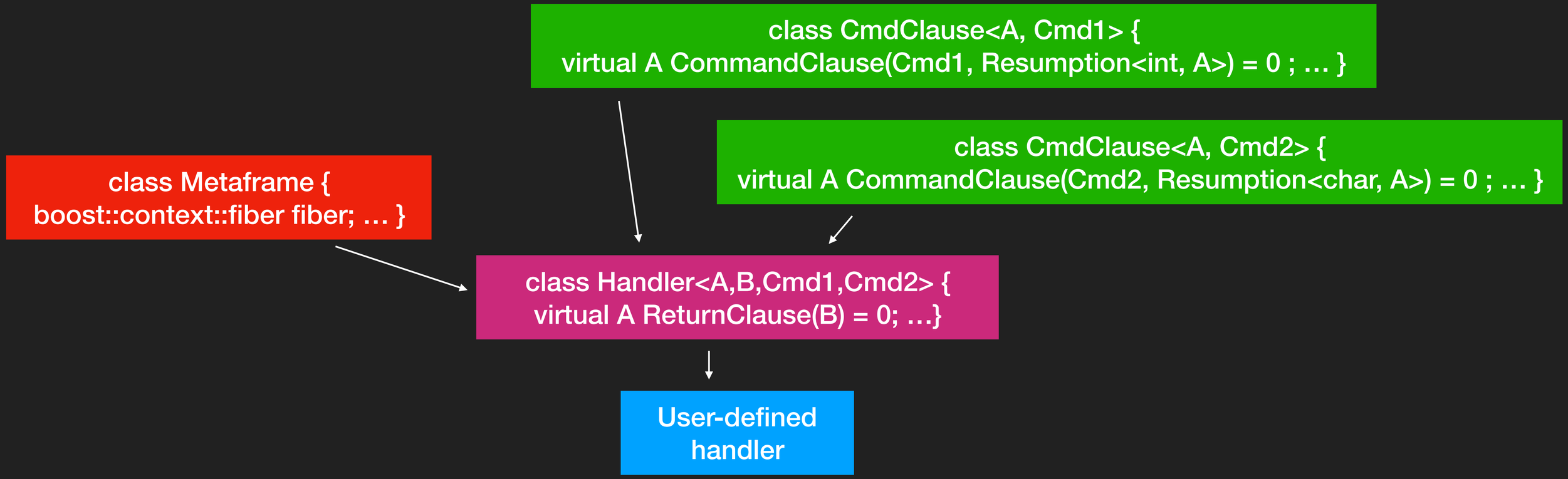
```
class Metaframe {
  ...
  boost::context::fiber fiber;
};

template <typename A, typename B, ...>
class Handler : Metaframe {
  ...
protected:
  virtual A ReturnClause(B b) = 0;
};

// almost, but not exactly:
std::list<Metaframe*> metastack;
```

Invoking a command

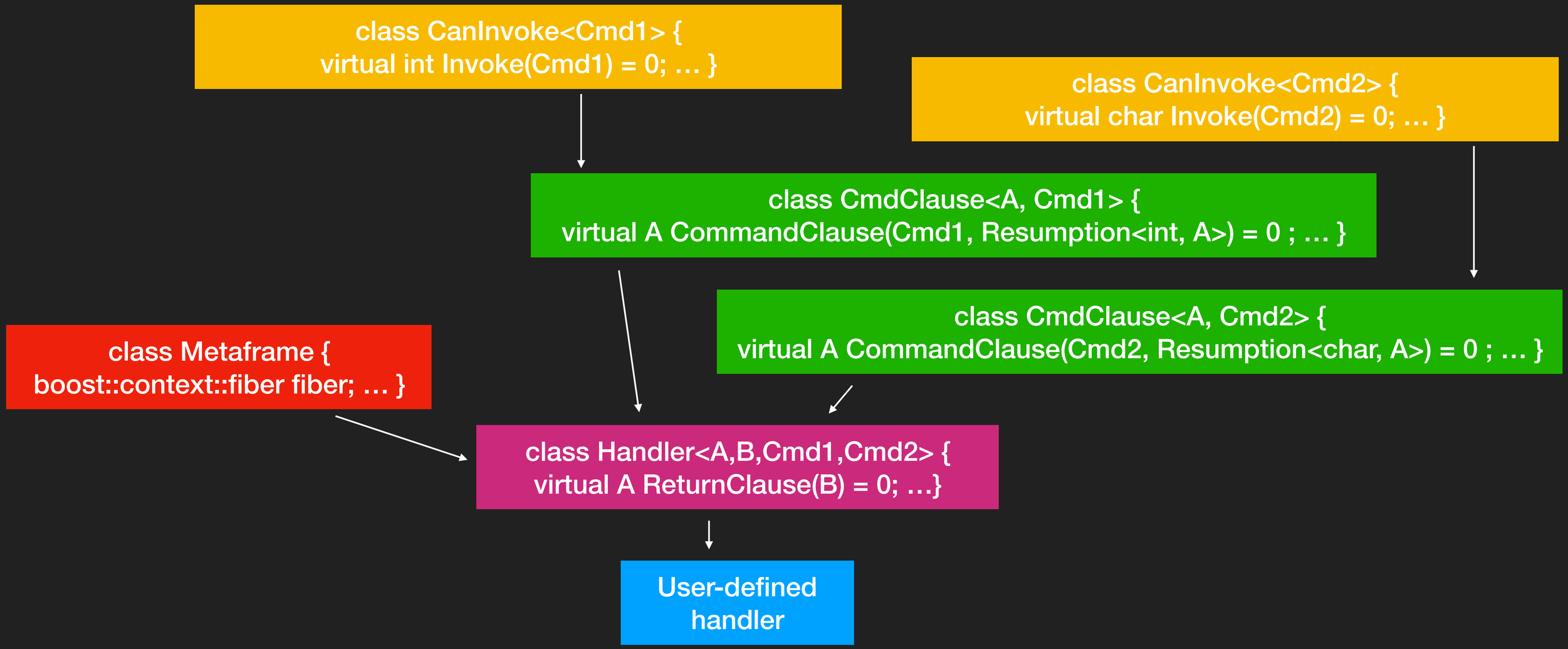
Here's a problem: When we invoke a command, we cannot know what the **type** of the handler is (in particular, we cannot know its answer type). We use dynamic dispatch. For example, consider the following Handler class:

```
struct Cmd1 : Command<int> {};
struct Cmd2 : Command<char> {};

class Handler<A, B, Cmd1, Cmd2>
{
  virtual A ReturnClause(B a) = 0;
  virtual A CommandClause(Cmd1, Resumption<int, A>) = 0;
  virtual A CommandClause(Cmd2, Resumption<char, A>) = 0;
};
```

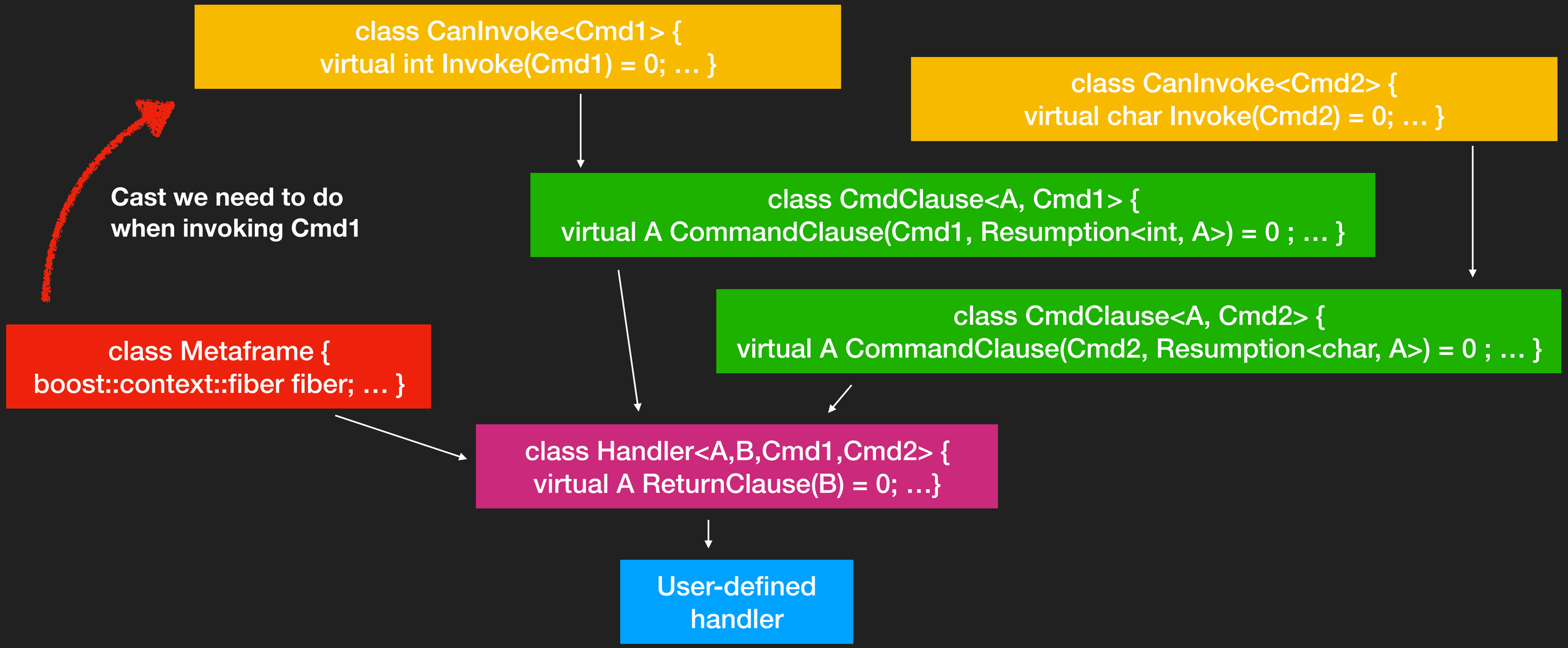
Invoking a command



Invoking a command



Invoking a command



Memory-management of handlers

A new handler is created when we handle a computation. For example:

```
static void Run(std::function<void()> f)
{
  OneShot::Handle<Scheduler>(f);
}
```

A new Scheduler object is created when we call OneShot::Handle.

But when should it die?

First thought: when it is removed from the meta stack (on calling ReturnClause)

Memory-management of handlers

…First thought: when it is removed from the meta stack (on calling ReturnClause)

But:

```
// Reveals the number of times SomeCmd is invoked
class H : public Handler<int, void, SomeCmd> {
  int state = 0;
  int CommandClause(SomeCmd, Resumption<void, int> r) override
  {
    OneShot::Resume(std::move(r));
    return state++; // this happens after return clause!!
  }
  int ReturnClause() override
  {
    return 0;
  }
};
```

Memory-management of handlers

Handler can be deleted after **both** of the following happen:

- OneShot::Handle returns (no more stack frames with pointer to the handler)
- An (always unique!) resumption that holds a pointer to the handler is deleted

There is no static way which will happen first, so we use shared pointers:

```
std::list<std::shared_ptr<Metaframe>> Metastack;
```

Things skipped over

- Mechanism for returning a value from a handler (which is also nontrivial because there's no natural place to store the answer).
- Pool of resumptions (it is possible pre-allocate all resumptions, because they are one-shot). This leads to a nasty pointer/stack cycle.
- No semantic TCO in C++ — naive handling quickly overflows the stack.
- Programming with handlers and value types (not that easy to define polymorphic operations needed for exceptions or async-await).

Performance

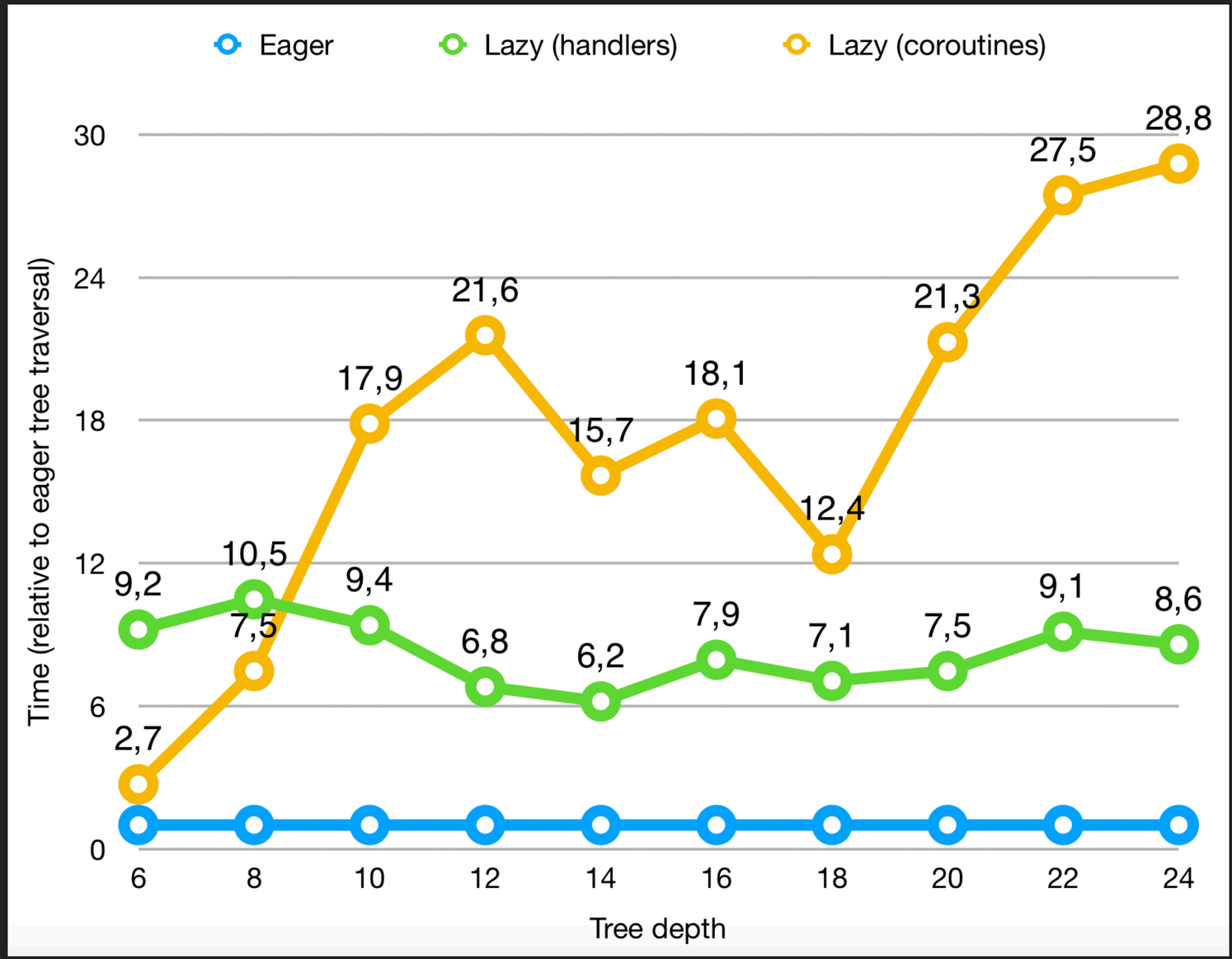
- We don’t hope for the best possible performance without direct support from the compiler (which we don’t have, because it’s just a library) or low-level hacking (except for boost.context, the library is pure C++17).
- The stack of abstraction is quite tall anyway: boost.context -> handlers -> user-defined effects. In a critical hot code, we would probably trade abstraction for performance and use bare boost.context or try to avoid effects. Otherwise, performance seems not to be a deal-breaker.
- We could trade elegant API for possible performance gains, but we don’t want to.

Some optimisations that could be performed by the compiler can be explicitly announced by the programmer. E.g. for self- and tail-presumptive clauses:

```
template <typename S>
struct Put : Command<void> {
  S newState;
};

template <typename S>
struct Get : Command<S> { };
```

```
template <typename Answer, typename S>
class HStateful : public FlatHandler<Answer,
                        Plain<Put<S>>, Plain<Get<S>>> {
public:
  HStateful(S initialState) : state(initialState) { }
private:
  S state;
  void CommandClause(Put<S> p) final override
  {
    state = p.newState;
  }
  S CommandClause(Get<S>) final override
  {
    return state;
  }
};
```

A very crude benchmark: traversing a binary tree (using a fold and two types of generators).



A more crude benchmark: a repeated simple mathematical formula (one addition, one multiplication, one mod) repeated for a variable. This formula can live in a loop body, function, or a handler.

```
--- plain handler ---
loop:              11353007ns
native:            13107581ns
native-inline:     10164356ns
lambda:            17320162ns
dynamic_cast:      136198780ns
handlers:          613352095ns
plain-handlers:    363418662ns
s-handlers:        369128845ns
s-plain-handlers: 91883497ns
```

Future Work:

- Sensible benchmarks and use-cases,
- Better error messages?
- Port to C++20 (because of concepts)
- Try a different backend, e.g. Dean Leijen's **libhandler** library.

https://github.com/maciejpirog/cpp-effects

Maciej Piróg
Huawei Research, Edinburgh
maciej.pirog@huawei.com