High-Level Type-Safe Effect Handlers in C++

DAN GHICA, Huawei Research, UK
SAM LINDLEY, The University of Edinburgh, UK
MARCOS MAROÑAS BRAVO, Huawei Research, UK
MACIEJ PIRÓG, Huawei Research, UK

Effect handlers allow the programmer to implement computational effects, such as custom error handling, various forms of lightweight concurrency, and dynamic binding, inside the programming language. We introduce `cpp-effects`, a type- and memory-safe C++ library for effect handlers with a high-level, object-oriented interface. We demonstrate that effect handlers can be successfully applied in imperative systems programming languages with manual memory management. Through a collection of examples, we explore how to program effectively with effect handlers in C++, discuss the intricacies and challenges of the implementation, and show that despite its limitations, `cpp-effects` performance is competitive and in some circumstances even outperforms state-of-the-art approaches such as C++20 coroutines and the `libmprompt` library for multiprompt delimited control.

1 INTRODUCTION

Effect handlers [Plotkin and Pretnar 2009, 2013] are an expressive control mechanism that allows programmers to define and manage bespoke, fit-for-purpose computational effects. Typical examples include customised error handling, mutable state, input/output, lightweight concurrency, dependency injection, and dynamic binding. Handlers also allow for transparent composition of effects — both with each other and with native effects built into the language. From an engineering methodology perspective, the advantage of handlers is the explicit separation of effect definition and their programming interface, a collection of *commands* (also known elsewhere in the literature as *operations*). This makes the abstraction ergonomic, as it does not require any exotic conventions to use an effect, unlike the case of programming with monads. It also makes the job of instrumenting existing code possible without extensive rewriting.

One application of effect handlers which is emerging as increasingly important is lightweight cooperative concurrency, that is, concurrency in which all tasks are realised on a single OS thread, without pre-emption. Effect handlers can lead to streamlined and efficient implementations of lightweight (green) threads with different kinds of schedulers, fibers, generators, async/await, message-passing actors, etc. Multiple such user-defined concurrency abstractions compose seamlessly, to obtain, for example, nested generators, or threads in which every thread has its own scheduled pool of message-passing actors. The advantage of using handlers is that the programmer can fine-tune all the details (schedulers, communication channels, cancellation policy, error handling), while being able to define them on a relatively high level in a type-safe manner, which would be hard to achieve with lower-level tools, such as a bare context-switching mechanism.

The origins of effect handlers [Plotkin and Pretnar 2009, 2013] lie in the realm of functional programming [Kammar et al. 2013], first as a tool to extend the capabilities of the formal semantics of algebraic effects [Plotkin and Power 2001, 2002, 2003], later as a distinctive feature of several experimental programming languages [Bauer and Pretnar 2015; Convent et al. 2020; Hillerström and Lindley 2016; Leijen 2017b]. Now more mainstream programming languages are beginning to adopt them as built-in features, for example Multicore OCaml [Sivaramakrishnan et al. 2021] (soon to be released as part of OCaml 5.0) and Uber's Pyro language for probabilistic programming [Bingham et al. 2019]. Effect handlers heavily influenced the design of the React GUI framework [Meta

Authors' addresses: Dan Ghica, Huawei Research, UK; Sam Lindley, The University of Edinburgh, UK; Marcos Maroñas Bravo, Huawei Research, UK; Maciej Piróg, Huawei Research, UK.

2022], for instance, inspiring the design of "React Hooks". Effect handlers are also central to the code navigation functionality of GitHub. This functionality depends on the GitHub Semantic library [GitHub 2022], which itself depends fundamentally on effect handlers in order to allow analyses to scale modularly to support multiple languages and features.

In this paper, we address the open question of whether effect handlers can be meaningfully exploited in system programming languages, such as C++, which allow for low-level programming with a focus on performance, but also include sophisticated mechanisms for high-level low-cost abstractions, such as objects and classes. We address this problem by implementing and evaluating `cpp-effects`, a reasonably performant C++ library for programming with effect handlers. It is built around the stack-switching mechanism provided by the `boost.context` library [Boost 2022]. Our handlers are one-shot, meaning that suspended computations can only be resumed once. In this style, resumptions are more akin to mutable linear resources, rather than the immutable (and copyable) continuations of functional languages, which is in accord with how stack and memory are manipulated in C++ via RAII and move semantics [Combette and Munch-Maccagnoni 2018].

The main contributions of this paper are the following:

(1) **A usable library for effect handlers in C++.** The design goal of our library is to reconcile an ergonomic API with type- and memory-safety. The library requires no additional compiler support and incurs no performance penalties for code that does not use its features. We argue that this library is not merely a proof-of-concept exercise, but can offer a decisive step towards the incorporation of effect handlers into C++ programming practice.

(2) **An evaluation of programming with handlers in an object-oriented system programming language.** Effect handlers have been previously implemented in object-oriented languages with managed (garbage-collected) memory [Brachthäuser et al. 2018; Inostroza and van der Storm 2018], and also in C [Leijen 2017a] but without providing a type-safe API. In our current implementation we show that the paraphernalia of object orientated programming plays a key role in helping overcome the challenges of low-level programming, namely the use of templates (instead of generics), value types, and direct memory management in the absence of garbage collection. We discuss in depth some further difficulties of programming with effect handlers in such a setting. For example, the style of programming cannot be ported wholesale from functional languages, since such naive implementations will quickly overflow the stack. Such problems require specific solutions, which we provide.

(3) **Higher-level library on top of a low-level core.** The core `boost.context` library suffers from limited functionality, but enjoys the benefits of small size and wide availability. Our library is built on top of it, in pure C++, so it is as portable as `boost.context` itself. In fact it should be easy to replace `boost.context` with some other backend for similar low-level stack manipulations. Our design allows for a clear separation between the architecture-specific core and the implementation of handlers using abstractions provided by the language. It can serve as a recipe for libraries in other languages, and in the implementation of compilers of languages with native effect handlers. In the particular case of a compiler, the benefit is that the compiler's backend need only provide a basic stack-switching mechanism, while the rest of the abstraction can be implemented entirely in the frontend.

The focus of this paper is twofold: the programmer's experience of using effect handlers in C++ and the implementation of the library.

Section 2 discusses effects and their handlers on some typical examples: mutable state, cooperative threads, message-passing actors, and generators. These examples also illustrate the way effect handlers can be used in the OO setting: mutable state can be implemented using native mutable state; a scheduler has the familiar shape of a while-loop that picks the process to be resumed; actors can be implemented as a composition of state and threads. We discuss how the lack of tail-call optimisation and the use of value types and templates can adversely affect the experience of using handlers, when compared to Java or functional languages, but we also show how some commonly used workarounds can be successfully applied here. We also discuss how the design of the library allows for some optimisations that can be applied by the programmer to make the code faster and more readable, e.g., a tail- and self-resumptive command clause in a handler can be explicitly marked as such, and in effect can be sped up by avoiding context switching altogether. Section 3 takes a closer look at the implementation details of effect handlers. Our approach is based on a stack of active handlers, each storing a call-stack segment (a *stacklet*), used to evaluate the handled computation. This is a known implementation strategy, used for example in the Multicore OCaml compiler [Sivaramakrishnan et al. 2021]. However, due to our particular setting, we must address a number of novel obstacles. First we show that OO abstractions and metaprogramming capabilities of C++ are enough to provide an ergonomic, type-safe API for handlers as a library. Then we tackle issues of memory-safety and performance. Section 4 presents some microbenchmarks. Though our main focus is the ergonomics of programming with `cpp-effects`, and we are sure there is plenty of scope to improve performance, these benchmarks provide evidence that `cpp-effects` performance can be competitive with the existing mainstream. In particular, we compare with generators built on top of C++20 coroutines, and mutable state built using the `libmprompt` library [Leijen and Sivamarakrishnan 2022]. Section 5 discusses related work and Section 6 concludes.

Our library is available at:

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

2 THE PROGRAMMER'S INTERFACE

In this section, we introduce the features of `cpp-effects` through a collection of examples including mutable state, lightweight cooperative threads, actors, and generators. These examples and more (e.g. async/await) are available with the library.

2.1 Mutable state

We begin with a basic mutable state effect in order to illustrate how effects and handlers work in our library. The main purpose of this example is not to illustrate anything like the full power of effects and handlers, but rather to give a minimal overview of the syntax and semantics of the core features of our library. Having said that, it turns out that the mutable state effect is a crucial component for actors (Section 2.3). The aim is to abstract over what it means to read and write to a single state cell. The interface will be given by two *commands*: `Put` writes a value to the state cell, and `Get` reads the current. Different effect handlers provide different implementations of these commands: an implementation might implement the state cell on the local heap in the standard way, but it might alternatively store it remotely or apply more complex policies such as caching.

First let us define the two commands which constitute the effect interface.

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

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

A command is defined as a class (or a struct, which is really just a class whose members are all public) that inherits from the `Command` class. The template parameter is the return type for the command. Arguments are supplied to the command as fields of the class. The two commands are defined as templates in order to enable them to be parameterised by the underlying type of the state cell (`S`). The `Put` command takes a single argument `newState` of type `S` and does not return a value (denoted by `void`). The `Get` command takes no arguments and returns a value of type `S`.

In order to invoke a command we use the `InvokeCmd` static member function from the `OneShot` class. For convenience, we define templated wrapper functions for invoking each of our two commands.

```
template <typename S>
void put(S s) {
  OneShot::InvokeCmd(Put<S>{{}, s});
}

template <typename S>
S get() {
  return OneShot::InvokeCmd(Get<S>{});
}
```

We can now write state computations such as an increment function `inc` for integer state.

```
int inc()
{
  put(get<int>() + 1);
  return get<int>();
}
```

This function increments the current value of the state and returns the updated value of the state.

So far we have given the state interface as a pair of commands and shown how to invoke them. However, we have not yet given them a meaning. In order to do that, we must define a handler. We now define a state handler that simply represents the state cell as a private member.

```
template <typename Answer, typename S>
class Stateful : public Handler<Answer, Answer, Put<S>, Get<S>> {
public:
  Stateful(S initialState) : state(initialState) { }
private:
  S state;
  Answer CommandClause(Put<S> p, Resumption<void, Answer> r) override
  {
    state = p.newState;
    return std::move(r).TailResume();
  }
  Answer CommandClause(Get<S>, Resumption<S, Answer> r) override
  {
    return std::move(r).TailResume(state);
  }
  Answer ReturnClause(Answer a) override
  {
    return a;
  }
};
```

A handler is defined as a class that inherits from the `Handler` class. The general idea is that a handler handles some computation by interpreting the commands used by that computation and the return type of the computation appropriately. Although not the case here, the return type of the handler and the return type of the computation being handled may differ. These are determined respectively by the return and argument types of the return clause. The template parameters are: the return type of the handler, the return type of the computation being handled, and a list of all handled commands. In this case the final return type and original return type are both `Answer`. We abstract over this answer type and the type of the state cell using a template.

Each command is given a meaning by overloading the `CommandClause` method. Here we have one overload for `Put` and another one for `Get`. The second argument `r` to the `CommandClause` method is a *resumption*. It is an object that captures the rest of the computation being handled. Its first template parameter is the return type of the command and its second template parameter is the final return type. Our use of resumptions in this example is not particularly interesting (we pass the return value to the resumption once at the end of each command clause), but as we shall soon see resumptions are central for allowing effect handlers to express features such as concurrency. An important aspect of resumptions that is visible here is that they are *one-shot*. They are movable but not copyable. After resuming, the resumption object becomes invalid. When a resumption object goes out of scope the computation it captures is deleted (i.e., the corresponding stack is unwound). Being one-shot allows resumptions to be implemented efficiently using non-copyable data structures such as various kinds of system stack or fibers. Indeed, our implementation is built on the non-copyable fibers of the `boost.context` library.

The `Put` command clause updates the state member with the new value and then invokes the resumption with no argument (`void`). The `Get` command clause invokes the resumption with the current state. The `ReturnClause` method defines how to process the final result value. In this case it is simply returned as is.

Having defined a handler we can now invoke it on a computation.

```
int main()
{
  std::cout << OneShot::Handle<Stateful<int, int>>(inc, 100);  // Output: 101
}
```

A computation is handled using the static function `OneShot::Handle`. The first argument is the computation. Subsequent arguments are forwarded as arguments to the constructor of `Stateful`.

2.2 Cooperative lightweight threads

Now we move onto an example, cooperative lightweight threads, that begins to demonstrate the real power of effect handlers. We begin by defining two commands and convenient wrapper functions.

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

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

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

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

The `Yield` command yields control to another lightweight thread. The `Fork` command forks off a new lightweight thread. They can be used in the following code, where `starter` starts a number of threads, each printing out a number in a loop.

```
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([=](){ worker(i); });
  }
}
```

The intention is that calling `starter` in a scheduler will print out a stream of interleaved digits.

A handler for `Yield` and `Fork` defines a scheduler. Here we implement the round-robin strategy.

```
using Res = Resumption<void, void>;

class Scheduler : public Handler<void, void, Yield, Fork> {
public:
  static void Start(std::function<void()> f)
  {
    Run(f);
    while (!queue.empty()) {
      auto resumption = std::move(queue.front());
      queue.pop_front();
      std::move(resumption).Resume();
    }
  }
private:
  static std::list<Res> queue;
  static void Run(std::function<void()> f)
  {
    OneShot::Handle<Scheduler>(f);
  }
  void CommandClause(Yield, Res r) override
  {
    queue.push_back(std::move(r));              // push the captured resumption on the queue
  }
  void CommandClause(Fork f, Res r) override
  {
    queue.push_back(std::move(r));              // push the captured resumption on the queue
    queue.push_back({std::bind(Run, f.proc)});  // lift a function to resumption
  }
  void ReturnClause() override { }
};
```

The scheduler maintains a queue of lightweight threads represented as resumptions. Execution is initiated by the static `Start` method, which is passed a computation as a function argument. The `Run` method forks off a new thread. The scheduling loop executes each thread in the queue in turn. The interesting control flow is provided by the command clauses. The `Yield` clause simply pushes the current resumption onto the queue, causing control to return to the scheduling loop. The `Fork` clause again pushes the current resumption onto the queue, but then also places the new forked thread onto the queue too.

In functional programming languages with tail-call optimisation and effect handlers, one typically specifies this example using tail-recursion. But here we see that it works perfectly well using a loop instead. One message we would like to convey is that though effect handlers do require a way of capturing resumptions, they do not depend on functional programming and work perfectly well when programming in a primarily imperative style.

We can run the scheduler as follows.

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

This will print out `010210321043210432104321043210432104321043210432104321432434`.

2.3 Actors

One of the key strengths of effect handlers is that they are composable. We now show how to compose our state and lightweight thread handlers in order to implement message-passing actors similar to those of Erlang [Armstrong et al. 1996]. For simplicity, we here give a fixed implementation of actors, but in Section 2.5 we observe that we can treat actors themselves as an effect whose implementation can itself be given by a handler (or indeed different handlers corresponding to different implementation strategies).

Following Erlang, we refer to actors as *processes*. Each process has its own mailbox. Any process can send messages to the mailbox of any other process, but only the owner of a mailbox can read messages from it. We begin by representing a process id as a pointer to a queue of messages.

```
using Pid = std::shared_ptr<std::queue<std::any>>;
```

We use the `std::any` type, because mailboxes are heterogeneous, which means that messages can be of different types.

The actor interface is given by the following four functions.

```
Pid spawn(std::function<void()> body);         // spawn a new process and return its process id
Pid self();                                    // return the process id of the current process
template <typename T> void send(Pid p, T msg)  // send msg to p
template <typename T> T receive();             // receive a message
```

Here is a simple ping-pong example that uses the interface to spawn a process. The main process then sends a sequence of messages to the child process which just sends the messages back again.

```
void pong()
{
  while (true) {
    auto [pid, n] = receive<std::tuple<Pid, int>>();
    if (n == 0) { return; }
    send<int>(pid, n);
  }
}

void ping()
{
  auto pongPid = spawn(pong);
  for (int i = 1; i <= 10; i++) {
    send<std::tuple<Pid, int>>(pongPid, {self(), i});
    std::cout << receive<int>() << std::endl;
  }
  send<std::tuple<Pid, int>>(pongPid, {self(), 0});
}
```

Now we give the implementation of the actor interface, which uses the effects defined in previous sections: state and threads.

```
Pid spawn(std::function<void()> body)
{
  auto mailbox = std::make_shared<std::queue<std::any>>();
  fork([=]() {
    OneShot::Handle<Stateful<void, Pid>>(body, mailbox);
  });
  return mailbox;
}

Pid self()
{
  return get<Pid>();
}

template <typename T>
void send(Pid p, T msg)
{
  p->push(msg);
}

template <typename T>
T receive()
{
  auto mailbox = get<Pid>();
  while (mailbox->empty()) { yield(); }
  auto msg = mailbox->front();
  mailbox->pop();
  return std::any_cast<T>(msg);
}
```

The `spawn` function makes use of the `Fork` command to implement the new process as a lightweight thread. It also uses the `Stateful` handler to manage the mailbox in the body of the process. The `self` and `receive` functions access the mailbox using the `Get` command. (It turns out that for this example we only really need a read-only state effect — we don't use `Put` — as the mailbox itself is accessed through a further indirection.) We make use of templates and `std::any_cast` in order to support different message types. If we try to receive a message when the mailbox is empty, then the current process will yield until the mailbox is no longer empty.

We can now run our ping example using the `Scheduler` handler for lightweight threads, with the `Stateful` handler used internally.

```
int main()
{
  Scheduler::Start(std::bind(spawn, ping));
}
```

This outputs the sequence of integers as expected.

A notable aspect of our implementation is that the state is bound dynamically as a result of being implemented using handlers. Whenever we call `receive` or `self`, we dynamically bind to the state, so we do not have to carry any information about the current process id around, and the body of the actor looks natural, even if it uses features such as recursion or higher-order functions. Crucially, we could not use a global variable to maintain the state, because the state is not global, but rather local to a process.

2.4 Ergonomics

Sometimes the full power of effect handlers is overkill. For instance, in the `Stateful` handler we always invoke the resumption argument in tail position at the end of each command clause. Moreover, the return clause just returns the value it is passed. It is possible to avoid writing some of this boilerplate. Here is a more ergonomic version of `Stateful`.

```
template <typename Answer, typename S>
class Stateful : public FlatHandler<Answer, Plain<Put<S>>, Plain<Get<S>>> {
public:
  Stateful(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;
  }
};
```

The `FlatHandler` class automatically inserts an identity return clause — hence it only takes a single `Answer` template argument before the commands. Flat handlers make it easier to define handlers that are truly parametric in the final result type. The problem is due to the C++ template system. For example, in our original definition of the `Stateful` handler, it is not possible to instantiate `Answer` with `void`, as it is used as the argument type of the `ReturnClause` method.

The *clause modifier* `Plain` in the type arguments of `Handler` indicates that a command is a *plain* command, meaning that the resumption given as the argument of `CommandClause` must be invoked in tail position at the end of the command clause (we say that it is *tail- and self-resumptive*). As a consequence there is no need to expose the resumption at all to the programmer. The corresponding `CommandClause` no longer includes a resumption argument, and the return type is now that of the command instead of the final return type of the handler. As well as improving readability, plain commands also improve performance as there is no need to perform any kind of context switch. So where possible, it is worth marking a command as plain.

Plain clauses are not to be confused with two ways in which we can resume: `Resume` (used in the handler for threads) and `TailResume` (used in the handler for state in Section 2.1). The reason for the latter is the lack of general tail-call optimisation in C++. If we used `return std::move(r).Resume()` in the state handler, the `CommandClause` frames created when we invoke a command would stay on the call stack until the very end of the handled computation. Since we need to be able to perform any number of commands in a computation, `TailResume` allows us to avoid stack overflow. It can be used for any resumption as the last handler-related statement in a command clause (or a function called by a command clause), and it can be used to replace the top-most `CommandClause` call-frame with the resumed computation. However, from a code-engineering perspective, if the tail-resumed resumption is the one we get as an argument, it makes more sense to use the `Plain` modifier, while if the effect juggles a number of resumptions, it is, in our experience, more convenient to trampoline them in a loop as in the lightweight threads example.

The library provides a further clause modifier `NoResume`. This is used to indicate a command that never invokes its resumption at all, in other words an exception. As with plain command clauses, the resumption argument is omitted from a `NoResume` command clause. For instance, we might want to extend the lightweight threading interface to support a command to kill the current thread.

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

We could then adapt the `Scheduler` handler as follows:

```
class Scheduler : public Handler<void, void, Yield, Fork, NoResume<Kill>> {
  ...
  void CommandClause(Kill) override { }
  ...
};
```

The main benefit here is clarity. Because of the techniques we use to optimise creation of resumptions, the performance benefit is minimal.

2.5 Actors revisited

In Section 2.3, we gave a fixed implementation of actors. Now we decouple that implementation into an effect interface and a handler, opening up the possibility of easily plugging in alternative implementations. For instance, we might want to swap out the underlying implementation of lightweight threads to use a different scheduler, or we may wish to give a completely different implementation that does not factor through the lightweight threads implementation at all.

The commands and wrapper functions are as follows.

```
using Pid = std::shared_ptr<std::queue<std::any>>;

struct Spawn : Command<void> {
  std::function<void()> body;
};

struct Self : Command<Pid> { };

struct Send : Command<void> {
  Pid p;
  std::any msg;
};

struct Receive : Command<std::any> { };

Pid spawn(std::function<void()> body)
{
  return OneShot::InvokeCmd(Spawn({}, body));
}

Pid self()
{
  return OneShot::InvokeCmd(Self{});
}

template <typename T>
void send(Pid p, T msg)
{
  OneShot::InvokeCmd(Send({}, p, msg));
}
```

```
template <typename T>
T receive()
{
  return std::any_cast<T>(OneShot::InvokeCmd(Receive{}));
}
```

The commands themselves make use of `std::any`, but the wrapper functions for sending and receiving are template functions and in particular, the `receive` wrapper performs the type cast.

Now we can move the actual implementation of the commands into a handler.

```
template <typename Answer>
class Act : public FlatHandler<Answer, Plain<Spawn>, Plain<Self>, Plain<Send>, Plain<Receive>> {
  Pid CommandClause(Self) override
  {
    return get<Pid>();
  }
  Pid CommandClause(Spawn s) override
  {
    auto mailbox = std::make_shared<std::queue<std::any>>();
    fork([=]() {
      OneShot::Handle<Stateful<void, Pid>>(s.body, mailbox);
    });
    return mailbox;
  }
  void CommandClause(Send s) override
  {
      s.p->push(s.msg);
  }
  std::any CommandClause(Receive) override
  {
    auto mailbox = get<Pid>();
    while (mailbox->empty()) { yield(); }
    auto msg = mailbox->front();
    mailbox->pop();
    return msg;
  }
}
```

All of the commands are plain and the handler is a parametric flat handler. Now if we want to change the implementation, we can just define a different handler to use in place of `Act`.

This example illustrates a more general problem that we encounter with defining an API for effect handlers as a library: commands cannot be polymorphic. This is because a polymorphic command in the form of a template, say

```
template <typename T> struct Receive<T> { };
```

would make its corresponding command clauses both virtual and templates, which is illegal in C++.

The solution, which is folklore and applicable in many different scenarios, is to split the command into the monomorphic "core" and a polymorphic wrapper. This, however, often requires other types to be factorised in this way. For example, here is a snippet from our implementation of async/await:

```
struct GenericFuture {
  std::vector<Resumption<void, void>> awaiting;
};
```

```
template <typename T>
class Future : public GenericFuture {
  std::optional<T> value;
  ...
};

struct Await : Command<void> {
  GenericFuture* future;
};

template <typename T>
T await(Future<T>* future)
{
  if (*future) { return *(future->value); }
  OneShot::InvokeCmd(Await{{}, future});  // Suspend until the value is ready
  return future->Value();
}
```

2.6 Generators

Generators provide a convenient interface for producing a stream of results. They can be implemented using effect handlers in such a way that the user only sees the generator interface and is not exposed to any underlying commands or handlers. Because the type of the handler used internally is statically known, this provides an opportunity for performance gains.

The interface is given by a single `Yield` command which yields a value to the caller (not to be confused with the `Yield` command used by our earlier lightweight threads examples).

```
template <typename T>
struct Yield : Command<void> {
  T value;
};
```

Our wrapper function takes an additional `label` argument. This argument is passed to an overloaded variant of `OneShot::InvokeCmd`. It identifies the handler that will be used to handle the command.

```
template <typename T>
void yield(int64_t label, T x)
{
  OneShot::StaticInvokeCmd(label, Yield<T>{{}, x});
}
```

By default, and for all the examples we have seem up to now, the handler is chosen as the inner-most one that supports the invoked command, which requires dynamically checking runtime type information (RTTI). By supplying a label, we can reduce the need for such expensive RTTI checks. By using `StaticInvokeCmd` we are asserting that we know the exact type of the handler associated with `label` and in particular that it handles the `Yield` command with the correct type. This is a potentially dangerous assumption, but offers significant performance benefits, and can be done relatively safely if, as in this case, we encapsulate the command and handler inside a library, so that the user of the library will never need to interact with either.

We now give an implementation of a handler for generators.

```
template <typename T>
struct GenState;

template <typename T>
using Result = std::optional<GenState<T>>;
```

```
template <typename T>
struct GenState {
  T value;
  Resumption<void, Result<T>> resumption;
};
```

```
template <typename T>
class GeneratorHandler : public Handler<Result<T>, void, Yield<T>> {
  Result<T> CommandClause(Yield<T> y, Resumption<void, Result<T>> r) override
  {
    return GenState<T>{y.value, std::move(r)};
  }
  Result<T> ReturnClause() override
  {
    return {};
  }
};
```

A generator state comprises a value and a current resumption. The return type of the handler is `Result` object, which is an optional generator state.

The more interesting part of the implementation is in a separate `Generator` class.

```
template <typename T>
class Generator {
public:
  Generator(std::function<void(std::function<void(T)>)> body)
  {
    auto label = OneShot::FreshLabel();
    result = OneShot::Handle<GeneratorHandler<T>>(label, [body, label](){
      body([label](T x) { yield<T>(label, x); });
    });
  }
  Generator() { }  // Create a dummy generator that generates nothing
  T Value() const
  {
    if (!result) { throw std::out_of_range("Generator::Value"); }
    return result.value().value;
  }
  bool Next()
  {
    if (!result) { throw std::out_of_range("Generator::Next"); }
    result = std::move(result->resumption).Resume();
    return result.has_value();
  }
  explicit operator bool() const
  {
    return result.has_value();
  }
private:
  Result<T> result = {};
};
```

The idea is that when a generator is created its body is executed until the first value is yielded. The constructor takes the body as a higher-order function parameterised by a yield function whose implementation is supplied by the constructor itself. This implementation invokes the `Yield` command using a fresh label that identifies the generator handler. (The library maintains a global counter of labels and the user can create a fresh label using `OneShot::FreshLabel`.) This label is also passed to an overloaded version of `OneShot::Handle` in order to associate it with the handler.

Notice that generators are not copyable, as `GenState` is not copyable, as `Resumption` is not copyable.

The `Value` method returns the current value and the `Next` method moves onto the next value by invoking the resumption, returning true if the stream of values has not been exhausted. The following example illustrates how to use a generator to output a stream of 100 natural numbers.

```
int main()
{
  Generator<int> naturals([](auto yield) {
    int i = 1;
    while (true) { yield(i++); }
  });
  for (int i = 0; i < 100; i++) {
    std::cout << naturals.Value() << std::endl;
    naturals.Next();
  }
}
```

Again notice that this user code makes no reference to any commands or effect handlers.

3 IMPLEMENTATION

In this section, we give an overview of the implementation, and detail a few aspects specific to our setting. In general, our approach is based on a stack of handlers, similar to how effect handlers are implemented, for example, in Multicore OCaml [Sivaramakrishnan et al. 2021]. We discuss the inheritance structure of the `Handler` class, and memory management of handlers and resumptions.

3.1 Metastack (the stack of handlers)

We begin by explaining the semantics of effect handlers via manipulation of the call stack, and show how an optimised version of this semantics can directly inform the implementation. Since handlers are a form of generalised resumable exceptions, we first draw a parallel between effect handlers and exception handlers.

The familiar semantics (but not necessarily implementation) of exceptions can be described as follows. The `try comp catch(const E& e) catchClause` statement pushes a handler frame (which, broadly speaking, corresponds to `catch(const E& e) catchClause`) on the stack, and proceeds with `comp`. The `throw e` statement, assuming `e` is of type `E`, unwinds the stack until it finds a frame of the shape that corresponds to some `catch(const E& f) catchClause`, and then continues with `catchClause` with the value `e` bound to the reference `f`. Figure 1 depicts the process of throwing an exception.

In the case of handlers, the function `OneShot::Handle<H>(f)` pushes a new frame, which corresponds to a new object of type `H`, and proceeds with `f()`. A call to `OneShot::InvokeCmd<Cmd>(c)` finds the first frame on the stack that corresponds to a handler that can interpret `Cmd`, and then continues with its `CommandClause(Cmd, Resumption<...>)`, which gets as its arguments the value `c` and a new resumption that stores the stack segment above and including the handler frame. Importantly, we do not unwind the stack. The stack stored in a resumption is unwound only when the resumption is discontinued (that is, goes out of scope). Figure 2 depicts the invocation of a command.

The stack used in this semantics of effect handlers consists of segments of regular call frames, separated by handler frames. In practice, we can keep each segment of the stack (a *stacklet*) in a separate chunk of memory, and use a *metastack*: a stack of handlers, each with a pointer to

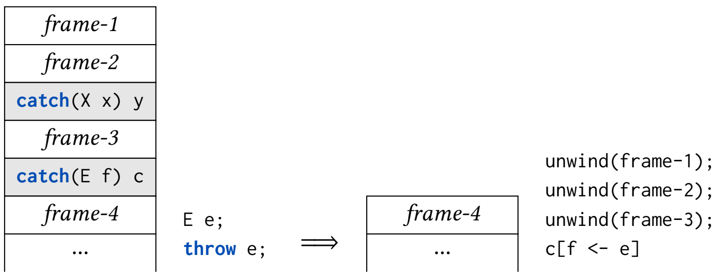


Fig. 1. Unwinding the call-stack on throwing an exception

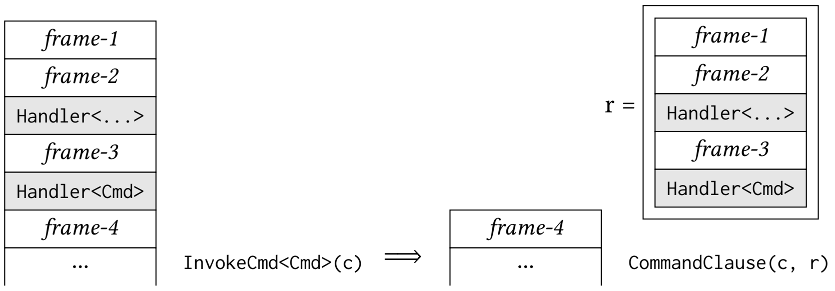
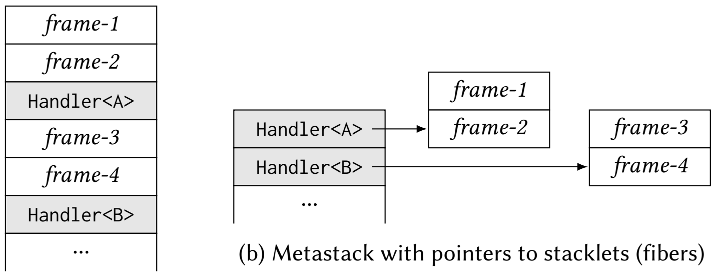


Fig. 2. Invoking a command



(b) Metastack with pointers to stacklets (fibers)

(a) Single stack

Fig. 3. Different representations of stack

the segment of regular frames above. It is useful for two reasons. First, it is faster to search the metastack for the right handler, as we do not have to go through every frame on the stack. Second, we don't have to physically move the memory that contains regular frames when we create or resume a resumption, we only manipulate pointers to stacklets.

3.2 Commands and handlers

The informal semantics given above is the basis of the implementation of the library. It does not require any additional support from the compiler, since the metastack is simply a global data structure (a linked list of pointers to handler objects), while the `boost.context` library provides the mechanism for stacklet allocation and switching.

There are three main class templates in the library: `Command`, `Handler`, and `Resumption`. The `Command` template, used as a parent class for user-defined commands, does not have any functionality of its own. It is just a wrapper around the type of the template's argument:

```
template <typename Out>
struct Command {
  using OutType = Out;
};
```

Handlers are defined via multiple inheritance to provide functionality that allows them to serve as frames on the metastack and provide interpretations to commands. First, every handler inherits from the `Metaframe` class, which groups a pointer to the stacklet (a *fiber* in `boost.context`'s terminology) and a label that can be used to select a handler.

```
class Metaframe {
  boost::context::fiber fiber;
  int64_t label;
  ...
};
```

For each command listed in the template arguments pack, `Handler` inherits from the `CmdClause` class. It provides the virtual member function `CommandClause` that particular implementations of handlers need to override.

```
template <typename Answer, typename Cmd>
class CmdClause {
protected:
  virtual Answer CommandClause(Cmd, Resumption<typename Cmd::OutType, Answer>) = 0;
  ...
};
```

Then, the handler is defined as:

```
template <typename Answer, typename Body, typename... Cmds>
class Handler : public Metaframe, public CmdClause<Answer, Cmds>... {
  using CmdClause<Answer, Cmds>::CommandClause...;
protected:
  virtual Answer ReturnClause(Body b) = 0;
  ...
};
```

Note that the `using` declaration in the definition of `Handler` exposes `CommandClause` from every `CmdClause` base, in a sense combining them together into one overloaded function.

The metastack is a list of pointers to objects of the common superclass of all handlers, `Metaframe`. This allows us, for example, to access the label of a handler on the metastack without knowing its actual type. Note that `Handler` is a template, so it would be cumbersome, if not impossible, to make the metastack a list of (well-typed pointers to) `Handler` objects.

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

The fact that the metastack is implemented as a linked list means that we can easily move parts of the metastack to resumptions when invoking a command (as in Figure 2) and back when resuming. Alternatively, we could use a vector, which provides faster handler lookup, but requires allocation every time we create a resumption. In practice, a metastack usually contains no more than a handful of frames, and our experiments suggest that performance of the two possible implementations is similar. We detail why we need reference-counting shared pointers in Section 3.5.

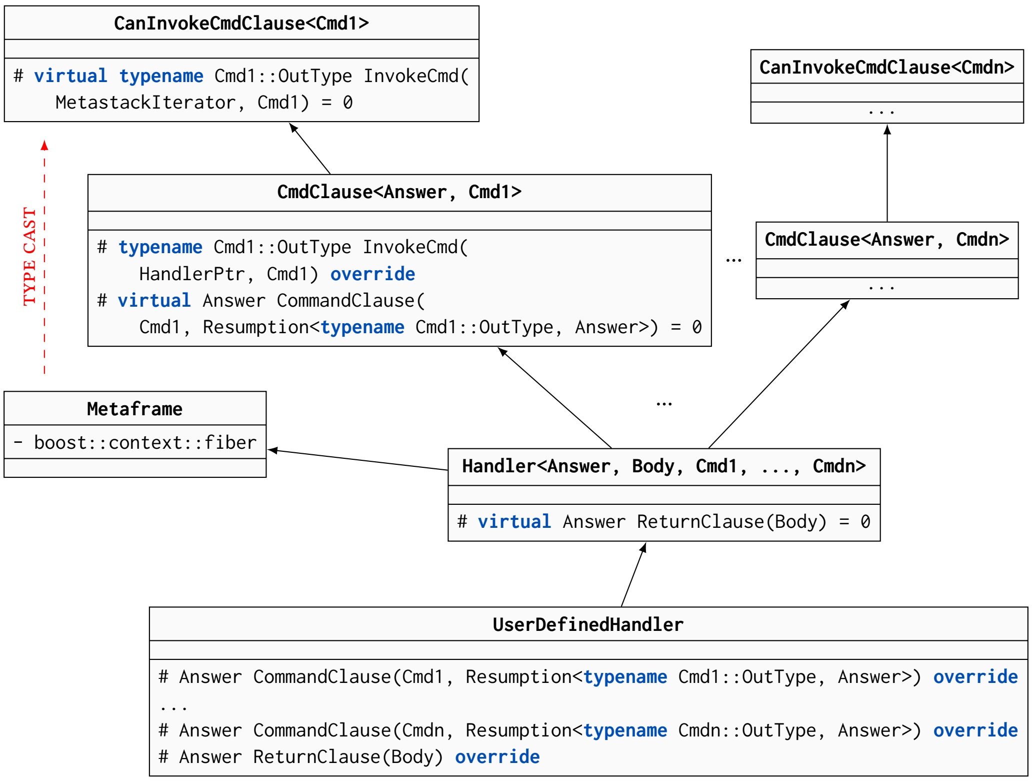

CanInvokeCmdClause<Cmd1>
# virtual typename Cmd1::OutType InvokeCmd(
MetastackIterator, Cmd1) = 0
CanInvokeCmdClause<Cmdn>
...
TYPE CAST
CmdClause<Answer, Cmd1>
# typename Cmd1::OutType InvokeCmd(
HandlerPtr, Cmd1) override
# virtual Answer CommandClause(
Cmd1, Resumption<typename Cmd1::OutType, Answer>) = 0
...
CmdClause<Answer, Cmdn>
...
Metaframe
- boost::context::fiber
...
Handler<Answer, Body, Cmd1, ..., Cmdn>
# virtual Answer ReturnClause(Body) = 0
UserDefinedHandler
# Answer CommandClause(Cmd1, Resumption<typename Cmd1::OutType, Answer>) override
...
# Answer CommandClause(Cmdn, Resumption<typename Cmdn::OutType, Answer>) override
# Answer ReturnClause(Body) override

Fig. 4. Inheritance diagram for handlers

The call to `OneShot::Handle<H>(comp)` first creates a new object of class `H` together with a stack, pushes a pointer to it on the metastack, and on the new stack calls a function that first calls `comp()`, and then uses its result to call the `ReturnClause` of the handler.

3.3 Invoking and handling a command

When the user invokes a command, we look for the right `Metaframe` on the metastack, down-cast from `Metaframe` to `Handler`, create the resumption, switch to the stacklet below the handler, and call `CommandClause` with the command and the resumption as its arguments. However, if we try to implement this scenario, we encounter a problem, which stems from the fact that we have to deal with value types in C++: in general, we cannot know the type of the handler, so we cannot simply down-cast to it. We cannot even do a sideways cast from `Metaframe` to `CmdClause<Answer, Cmd>` of the command, because in general we cannot know the `Answer` template parameter of `CmdClause` when invoking the command.

Fortunately, we never need to provide a value of the type `Answer` to implement the invocation. The code that is responsible for creating the resumption, switching stacks, and calling the virtual function `CommandClause` can be provided by the class `CmdClause<Answer, Cmd>` as an implementation of an interface that does not depend on the `Answer` type. In particular, the class `CmdClause` is defined as follows (the complete diagram of inheritance for a user-defined handler is shown in Figure 4).

```
template <typename Cmd>
class CanInvokeCmdClause {
```

```
protected:
  virtual typename Cmd::OutType InvokeCmd(std::list<MetaframePtr>::reverse_iterator, const Cmd&)
     = 0;
};

template <typename Answer, typename Cmd>
class CmdClause : public CanInvokeCmdClause<Cmd> {
  virtual typename Cmd::OutType InvokeCmd(std::list<MetaframePtr>::reverse_iterator, const Cmd&)
     final override;
protected:
  virtual Answer CommandClause(Cmd, Resumption<typename Cmd::OutType, Answer>) = 0;
  ...
};
```

Thus, the process of invoking a command is as follows.

(1) The user calls `OneShot::InvokeCmd<Cmd>`, which is responsible for finding the right handler on the metastack. Depending on the overload, it does so using runtime type information (the first handler for which the dynamic cast to `CanInvokeCmdClause<Cmd>` succeeds) or the handler's label (in which case we use the dynamic cast only on the first handler with the given label). In Figure 4, this cast from `Metaframe` to `CanInvokeCmdClause<Cmd1>` is indicated by the dashed arrow. The user can also call `OneShot::StaticInvokeCmd<H, Cmd>`, in which case we find the handler by the label, and statically cast it to `H`.

(2) The function `OneShot::(Static)InvokeCmd` calls the found `CanInvokeCmdClause<Cmd>`'s `InvokeCmd` function, providing it with the command and the metaframe *below* the found handler, since the found handler's command clause is run on the stacklet of the previous handler (see Figure 2). We create the resumption by moving the top segment of the metastack, switch the stack, and run the result's `CommandClause`. If the stack is ever switched back to this place, we return the value with which the resumption was resumed.

The virtual function call to `CanInvokeCmdClause<Cmd>::InvokeCmd` (implemented in `CmdClause<Answer, Cmd>`) solves the problem of the unknown `Answer` type of the handler, but it is also useful for implementing the optimisations provided by clause modifiers. Each clause modifier is implemented as a specialisation of `CmdClause`. For example, the implementation of `InvokeCmd` in `CmdClause<Answer, Plain<Cmd>>` simply does not create a resumption, and does not switch stacks, but only calls `CommandClause`, (temporarily removing the top segment of the metastack in case `CommandClause` itself uses effects).

3.4 Resumptions

Resumptions are implemented as smart pointers to the `ResumptionData` class, which stores a segment of the metastack and some additional members used for transferring data when resuming (which we omit in this description).

```
template <typename Out, typename Answer>
class ResumptionData {
  std::list<MetaframePtr> storedMetastack;
  ...
};

template <typename Out, typename Answer>
class Resumption {
  ResumptionData<Out, Answer>* data;
  ...
};
```

When the user invokes a command, the computation is suspended, a resumption is created by moving an appropriate segment of the metastack to `ResumptionData::storedMetastack`, and the appropriate command clause of a handler is called. In our design goals, we assumed that this process should be as fast as possible. As allocation is relatively expensive, we want to avoid allocating a new resumption `ResumptionData` object every time a command is called, and deleting it when it is resumed or goes out of scope.

Fortunately, we observe that since our resumptions are one-shot, the user cannot copy them, and so there can exist at most one resumption per suspended computation at a given time. Hence, we can pre-allocate all resumptions that will ever be needed for a particular handler object: one for each supported command. Hence, all `ResumptionData` objects can be kept as data members of the `Handler` class. In particular, each `CmdClause<Answer, Cmd>` base provides a "buffer" for a resumption that "hangs" on `Cmd`:

```
template <typename Answer, typename Cmd>
class CmdClause : public CanInvokeCmdClause<Cmd> {
  ResumptionData<typename Cmd::OutType, Answer> resumptionBuffer;
  ...
};
```

A local `Resumption` object is created as a pointer to the `resumptionBuffer` and given as argument to the command clause, which leads to an interesting circular dependency. A resumption contains a stored metastack (as a list, so via pointers), first metaframe of which is a pointer to a handler, which contains, as its member, the resumption. This cycle is problematic when a resumption goes out of scope, as its destructor needs to break this cycle first, which means that it is not enough to use `std::unique_ptr` to implement the `Resumption` class.

3.5 Lifetime of handlers

Prior implementations of effect handlers in object-oriented settings take advantage of automatic memory management: the object representing a handler is created when calling the equivalent of `OneShot::Handle`, and is deleted automatically by the garbage collector. In C++, there is no default garbage collector, and our library manages its own memory, relieving the user from deleting handlers manually.

When should a handler be deleted? One obvious guess would be: when it is popped from the metastack (that is, after the return clause returns) or when a resumption that holds a pointer to it is discontinued (the handlers are one-shot, so there can be only one such resumption). However, consider the following simple example of an effect that logs messages:

```
struct Log : Command<void> { std::string msg; };

class Logger : public Handler<std::string, void, Log> {
public:
  Logger(std::string separator) : separator(separator) { }
private:
  const std::string separator;
  std::string ReturnClause() override
  {
    return "";
  }
  std::string CommandClause(Log l, Resumption<void, std::string> r) override
  {
    return l.msg + this->separator + std::move(r).Resume();
  }
};
```

Since the order of evaluation of operator arguments is unspecified in C++, the body of the command clause for the command `Log` might very well be treated by the compiler as equivalent to:

```
auto&& temp = std::move(r).Resume();
return l.msg + this->separator + temp;
```

Now observe that `std::move(r).Resume()` resumes the entire computation and returns *after* the return clause returns. This means that the second line (`return l.msg + ...`) happens after the object is deleted, and so the expression `this->separator` tries to access a member of a deleted object! This means that we need to keep the handler alive as long as it is on the metastack (or the metastack stored in a resumption) or there are live stack frames that can refer to it. Since we have no way to statically determine where and when such frames might live, we manage the handler's lifetime using a shared pointer. It is shared pointers that we actually keep on the metastack, and we create a copy of the pointer for the duration of each command clause.

Bumping up the counter on each entry to a command clause and running the destructor of a shared pointer on exit has a performance penalty (over 10% of the time of the invoke–resume cycle in the `generator` example). It is especially unfortunate, because in most cases, there is no need for this, as most examples meet at least one of the following conditions:

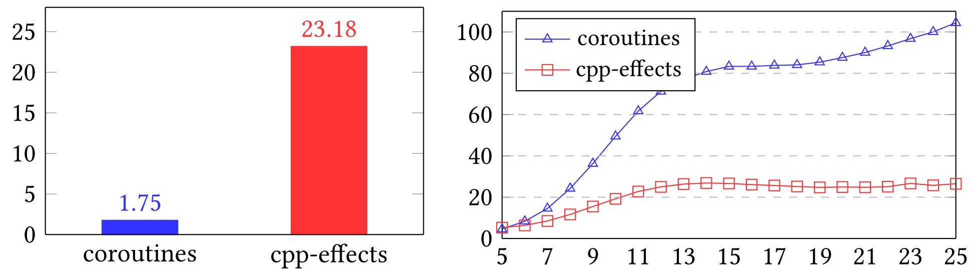
(a) The handler does not expose the resumption to the outside. In such a case, we know that all command clauses are run on top of the `OneShot::Handle` frame, and so when the frame is removed from the stack (either the function returns or the frame is unwound), we can safely delete the handler, because we know there are no more frames for command clauses anywhere on the stack or stored metastacks.

(b) The command clauses do not use the internal state of the handler after `Resume`. In this case, we can safely delete the handler even if there are still command clauses running on the current stack or any of the stacklets.

In these cases, the library allows the programmer to avoid paying the performance penalty needed in the general case. It is possible via another clause modifier, `NoManage`, that allows the programmer to trade guaranteed memory-safety for performance. It is used to indicate that at least one of the conditions above is true, and the command clause need not memory-manage the handler. A clause that is marked as `NoManage` will not care to contribute to the reference count of the handler. If all command clauses in the handler are marked as such, it means that there exist at most two references to the handler object: the pointer on the metastack, and a local variable in `OneShot::Handle`. If (a) happens, the metastack pointer is removed first, but the handler is kept alive by the pointer in `OneShot::Handle`. If (b) happens, it might be the case that the pointer in `OneShot::Handle` is removed first (for example, when a command clause stores the resumption in a global variable, and returns an unrelated value), and the handler is deleted after being popped from the metastack, but it will not cause any problems, as we know that the handled computation has ended (so there will not be new command clauses called) and all live command clause frames have already called `Resume`.

4 PERFORMANCE

While our primary goal is to provide a type- and memory-safe, usable programming interface for effect handlers via *just* a library – so without any specific optimisations provided by the compiler – performance is an important issue. As this is, as far as we are aware, the first high-level library for effect handlers in C++, there are no perfect candidates against which to measure the performance of our library. Nevertheless, we believe we can still draw some meaningful conclusions about the feasibility of using our library in programming practice.



(a) Generating numbers in a loop

(b) Recursive tree traversal

Fig. 5. Effect handlers vs native coroutines (lower is better)

First, we compare user-defined effects with those built into the language, such as exceptions and C++20 coroutines. Built-in effects are optimised for specific tasks, so unsurprisingly their performance is markedly better than for user-defined effects. However, for features beyond the scope of those provided by the language (for example, resumable exceptions, or stackful coroutines), such a comparison can help to assess the performance penalty of the additional expressiveness.

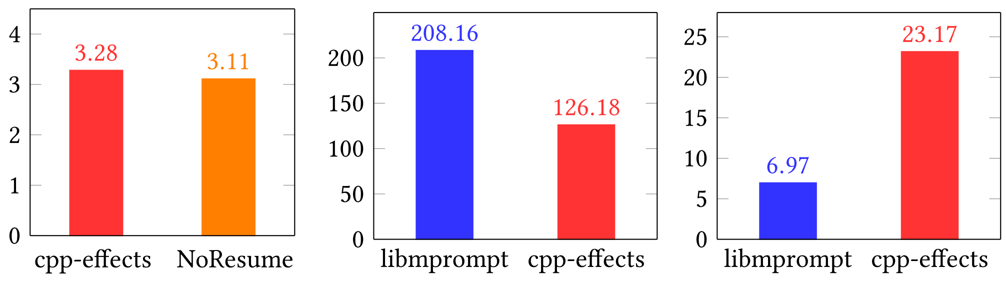
We also compare `cpp-effects` with `libmprompt` [Leijen and Sivamarakrishnan 2022] library, an existing C implementation of effect handlers. Differences in functionality include that `libmprompt` does not provide a type-safe high-level interface, features functional-style parameterised handlers instead of OO-style stateful handlers, and supports for multi-shot resumptions.

4.1 C++20 coroutines

First, we compare generators implemented via effect handlers with coroutines, which were introduced in C++20. The C++ coroutines are *stackless*, which means that they do not run on a separate call-stack, but are compiled via program transformation. In particular, they are compiled to objects with a member function representing the body of the coroutine, and data members representing local variables. Then, suspending a computation is compiled to returning from the function, and resuming to jumping to a particular place in the function body.

Such a jumping in and out of the function body can be much faster than stack switching, which we test on a program that generates consecutive natural numbers in a loop. Then, this generator is resumed a number times, adding up the generated numbers. The results are shown in Figure 5a, relative to an implementation that adds numbers using a loop with no concurrency, and indeed native coroutines are over 10x faster than our implementation using effect handlers.

However, the situation is different if we benchmark programs that cannot be compiled to a pair of `goto`-s, for example, when the body of the generator is a recursive function. In the case of coroutines, every recursive call corresponds to creating a new coroutine, in effect leading to a heap-allocated stack represented as a linked list of coroutines. Hence, instead of just jumping in and out of a function, the coroutine-based generator needs to perform a lot of allocations, which leads to worse performance. We test it on generators that recursively traverse a full binary tree with values in the leaves. When a generator reaches a leaf, it yields its value. Then, we add up all the values in a tree repetitively resuming the generator. The results are shown in Figure 5b, with y-axis showing execution time relative to recursive tree traversal with no concurrency, and x-axis is the depth of the tree. The version based on effect handlers is faster already for trees of height 6 and more, being at least twice as fast for trees of height 8 and more.



(a) Exceptions (b) General handlers (c) Plain handlers

Fig. 6. Effect handlers vs native exceptions and `libmprompt` (lower is better)

4.2 Exceptions

Exceptions in idiomatic C++ are used for handling proper errors. This means that most compilers favour the zero-cost implementation, in which installing an exception handler is cheap, while handling an exception is allowed to be expensive, since `throw` is assumed never to be on the hot path of execution. It is exactly the opposite of the usual scenario for effect handlers, in which we install an effect handler once (and allow this to be more expensive), and then invoke commands in large numbers, which should be as cheap as possible.

Nevertheless, it is still interesting to compare native exceptions with exceptions implemented using effect handlers. In a loop, we run a program that in each iteration installs a handler and throws an exception. The results are shown in Figure 6a, relative to the native version. As expected, if we modify the program not to throw the exception, the native version becomes 98% faster (because installing an exception handler is cheap), while the performance of the effect-based version does not change in any meaningful way (because invoking a command is cheap).

4.3 The libmprompt library

Finally, we compare with the `libmprompt` library [Leijen and Sivamarakrishnan 2022] and its frontend for effect handlers, `libmpeff`. Both `cpp-effects` and `libmprompt` offer a different set of features; for example, we provide a high-level API, while `libmprompt` offers multi-shot resumptions. Interestingly, both implement an optimisation for plain command clauses. We compare the two libraries on an example that modifies a mutable memory cell in a loop using the state effect. Figures 6b and 6c show the results relative to modifying the value stored in a variable. The first benchmark uses handlers implemented using the full power of handlers (as in Section 2.1), while the second one uses plain command clauses (as in Section 2.4).

5 RELATED WORK

Effect handlers in C. Leijen [2017a] describes how to implement effect handlers in C on top of `setjmp` and `longjmp`, exposing them via a rather low-level interface using C preprocessor macros. The `libhandler` library [Leijen 2019] is an implementation of this idea. The implementation has to be used with some care, but concrete features such as async/await can be implemented in such a way as to expose a relatively safe interface to the programmer. The `libmprompt` library [Leijen and Sivamarakrishnan 2022] is an alternative to the original libhandler library. Instead of implementing effect handlers directly it implements multiprompt delimited continuations and then a separate library `libmpeff` builds effect handlers on top. The `libmprompt` library uses virtual memory as a way of allowing stacks to grow without ever having to move them.

Effect handlers in OO. JEff [Inostroza and van der Storm 2018] is an object-oriented programming language with support for effect handlers. The Java Effekt library [Brachthäuser et al. 2018] is an implementation of effect handlers for Java. Like `libmprompt` it builds effect handlers on top of delimited continuations, which are implemented in Java using a form of CPS translation. The Scala Effekt library [Brachthäuser et al. 2020] is a similar library for Scala, which takes advantage of Scala's rich type system to incorporate a full-featured effect type system based on capabilities.

Stack switching in WebAssembly. WebAssembly [Rossberg et al. 2018] is a portable low-level bytecode for the web supported by all of the main browser vendors. Work is underway to extend web WebAssembly to support switching between stacks [WebAssembly Community Group 2022] in order to support exactly the kind of features that effect handlers are well-suited for (e.g. async/await, lightweight threads, and generators). In particular there is a concrete "Typed Continuations" proposal [Hillerström et al. 2022] along with an implementation in the WebAssembly reference interpreter, which amounts to an extension of WebAssembly with effect handlers. Up to now WebAssembly has largely been used for compiling C and C++; as such the Typed Continuations proposal supports an imperative style of effect handling, not dissimilar to `cpp-effects`, in which schedulers may be implemented as loops rather than with recursive functions and tail-calls.

Clause modifiers. Clause modifiers for handling commands tail- and self-resumptively (`Plain`) or as exceptions (`NoResume`), have been used previously, for instance in the Racket library associated with Kammar et al.'s early work on libraries for effect handlers [Kammar et al. 2013] and in Koka.

Commands and handlers as objects. The idea of representing commands and handlers as objects was introduced by Kammar et al. [2013] in their Haskell library for effect handlers. Like us, they use objects to maintain whatever state is necessary in commands and handlers. However, in their case, unlike ours, these objects are immutable. Similar ideas arise in later work on designs and implementations of effect handlers that make use of capability-passing [Brachthäuser et al. 2020; Zhang and Myers 2019] and evidence-passing [Xie et al. 2020; Xie and Leijen 2021].

6 CONCLUSION

We are not the first to implement effect handlers in an imperative language or the first to implement effect handlers in an object-oriented programming language. However, as far as we know ours is the first implementation of effect handlers specifically for C++. This presented a particular challenge due to the lack of garbage collection in C++. However, we were successfully able to exploit a broad range of C++ features in order to relatively smoothly integrate effect handlers with C++.

The experience of programming with `cpp-effects` is an extension of the regular experience of programming in C++. Commands and handlers are defined as classes, which can be combined with templates to provide a form of parametric polymorphism in the usual manner. Often a library can be implemented using effect handlers in such a way that the user of the library need not know anything at all about effect handlers (as illustrated by our implementation of generators). Though we have not invested huge effort into trying to optimise `cpp-effects`, it seems to offer adequate performance for realistic use-cases, and in some cases it outperforms existing approaches such as the `libmprompt` library and C++20 coroutines.

In future we would like to explore plugging in alternative backends with a view to improving performance. We would also like to explore means for providing some form of effect type system in order to further tame the complexity of programming in the large with effect handlers. Another direction which it would be interesting to explore is support for multishot effect handlers. This would require an alternative implementation mechanism as it would depend on being able to copy resumptions, but it opens up a range of other applications such as backtracking and probabilistic programming.

REFERENCES

Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. 1996. *Concurrent Programming in Erlang, Second Edition.* Prentice Hall International, Hertfordshire, UK.

Andrej Bauer and Matija Pretnar. 2015. Programming with algebraic effects and handlers. *J. Log. Algebr. Meth. Program.* 84, 1 (2015), 108–123.

Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul A. Szerlip, Paul Horsfall, and Noah D. Goodman. 2019. Pyro: Deep Universal Probabilistic Programming. *J. Mach. Learn. Res.* 20 (2019), 28:1–6.

Boost. 2022. Boost.context library. https://github.com/boostorg/context.

Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2018. Effect handlers for the masses. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 111:1–111:27.

Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020. Effekt: Capability-passing style for type- and effect-safe, extensible effect handlers in Scala. *J. Funct. Program.* 30 (2020), e8.

Guillaume Combette and Guillaume Munch-Maccagnoni. 2018. A resource modality for RAII. In *LOLA 2018: Workshop on Syntax and Semantics of Low-Level Languages.* 1–4.

Lukas Convent, Sam Lindley, Conor McBride, and Craig McLaughlin. 2020. Doo bee doo bee doo. *J. Funct. Program.* 30 (2020), e9.

GitHub. 2022. Semantic library. https://github.com/github/semantic.

Daniel Hillerström and Sam Lindley. 2016. Liberating effects with rows and handlers. In *TyDe.*

Daniel Hillerström, Sam Lindley, and Andreas Rossberg. 2022. WebAssembly Typed Continuations Proposal. https://github.com/effect-handlers/wasm-spec/proposals/continuations/Explainer.md.

Pablo Inostroza and Tijs van der Storm. 2018. JEff: objects for effect. In *Onward!* ACM, 111–124.

Ohad Kammar, Sam Lindley, and Nicolas Oury. 2013. Handlers in action. In *ICFP.* ACM, 145–158.

Daan Leijen. 2017a. Implementing Algebraic Effects in C — "Monads for Free in C". In *APLAS (Lecture Notes in Computer Science, Vol. 10695).* Springer, 339–363.

Daan Leijen. 2017b. Type directed compilation of row-typed algebraic effects. In *POPL.* ACM, 486–499.

Daan Leijen. 2019. libhandler. https://github.com/koka-lang/libhandler.

Daan Leijen and KC Sivamarakrishnan. 2022. libmprompt. https://github.com/koka-lang/libmprompt.

Meta. 2022. React library. https://reactjs.org/.

Gordon D. Plotkin and John Power. 2001. Semantics for Algebraic Operations. *Electr. Notes Theor. Comput. Sci.* 45 (2001), 332–345.

Gordon D. Plotkin and John Power. 2002. Notions of Computation Determine Monads. In *FoSSaCS (Lecture Notes in Computer Science, Vol. 2303).* Springer, 342–356.

Gordon D. Plotkin and John Power. 2003. Algebraic Operations and Generic Effects. *Applied Categorical Structures* 11, 1 (2003), 69–94.

Gordon D. Plotkin and Matija Pretnar. 2009. Handlers of Algebraic Effects. In *ESOP (Lecture Notes in Computer Science, Vol. 5502).* Springer, 80–94.

Gordon D. Plotkin and Matija Pretnar. 2013. Handling Algebraic Effects. *Logical Methods in Computer Science* 9, 4 (2013).

Andreas Rossberg, Ben L. Titzer, Andreas Haas, Derek L. Schuff, Dan Gohman, Luke Wagner, Alon Zakai, J. F. Bastien, and Michael Holman. 2018. Bringing the web up to speed with WebAssembly. *Commun. ACM* 61, 12 (2018), 107–115.

KC Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. 2021. Retrofitting effect handlers onto OCaml. In *PLDI.* ACM, 206–221.

WebAssembly Community Group. 2022. WebAssembly Stack Switching Extension. https://github.com/WebAssembly/stack-switching.

Ningning Xie, Jonathan Immanuel Brachthäuser, Daniel Hillerström, Philipp Schuster, and Daan Leijen. 2020. Effect handlers, evidently. *Proc. ACM Program. Lang.* 4, ICFP (2020), 99:1–99:29.

Ningning Xie and Daan Leijen. 2021. Generalized evidence passing for effect handlers: efficient compilation of effect handlers to C. *Proc. ACM Program. Lang.* 5, ICFP (2021), 1–30.

Yizhou Zhang and Andrew C. Myers. 2019. Abstraction-safe effect handlers via tunneling. *Proc. ACM Program. Lang.* 3, POPL (2019), 5:1–5:29.