Concurrent System Programming with Effect Handlers

Stephen Dolan[1], Spiros Eliopoulos[3], Daniel Hillerström[2⋆], Anil Madhavapeddy[1], KC Sivaramakrishnan[1], and Leo White[3]

[1] University of Cambridge
[2] The University of Edinburgh
[3] Jane Street Capital

Abstract. Algebraic effects and their handlers have been steadily gaining attention as a programming language feature for composably expressing user-defined computational effects. While several prototype implementations of languages incorporating algebraic effects exist, Multicore OCaml incorporates effect handlers as the primary means of expressing concurrency in the language. In this paper, we make the observation that effect handlers can elegantly express particularly difficult programs that combine system programming and concurrency without compromising performance. Our experimental results on a highly concurrent and scalable web server demonstrate that effect handlers perform on par with highly optimised monadic concurrency libraries, while retaining the simplicity of direct-style code.

Category: Research paper.

1 Introduction

Algebraic effects are a modular foundation for effectful programming, which separate the *operations* available to effectful programs from their concrete implementations as *handlers*. Effect handlers provide a modular alternative to monads [25,33] for structuring effectful computations.

The separation between operations and handlers is achieved through the use of delimited continuations, allowing effect handlers to pause, resume and switch between different computations. Effect handlers provide a structured interface for programming with delimited continuations [11], and can implement common abstractions such as state, generators, async/await, promises, non-determinism, exception handlers and backtracking search.

Algebraic effects were introduced by Plotkin and Power [27] as a mechanism to reason about computational effects in a pure setting, and extended by Plotkin and Pretnar [28]. The original work on effects equips operations with *equations* that their handlers must satisfy, although implementations to date have not

⋆ PhD student

included this feature (in this paper, we do not consider equations on effects). Though originally studied in a theoretical setting, effect handlers have gained practical interest with several prototype implementations in the form of libraries, interpreters, compilers and runtime representations [5,6,10,13,16,17,20,21].

However, the application space of effect handlers remains vastly unexplored. In this paper we explore the application space of effect handlers in system oriented programming.

2 Motivation

Multicore OCaml [9] incorporates effect handlers as the primary means of expressing concurrency in the language. The modular nature of effect handlers allows the concurrent program to abstract over different scheduling strategies [10]. Moreover, effect handlers allow concurrent programs to be written in *direct-style* retaining the simplicity of sequential code as opposed to callback-oriented style with either monadic concurrency libraries such as Lwt [32] and Async [24] for OCaml or explicit callbacks. In addition to being more readable, direct-style code tends to be easier to debug, compared to callback-oriented code; unlike callback-oriented code, direct-style code uses the stack for function calls, and hence, backtraces can be obtained for debugging. Indeed, experience from Google suggests that direct-style code not only improves performance but also makes the code more compact and easier to understand in warehouse-scale settings, where thousands of developers touch the codebase [4].

In Multicore OCaml, the user-level thread schedulers themselves are expressed as OCaml libraries, thus minimising the secret sauce that gets baked into high-performance multicore runtime systems [30]. This modular design allows the scheduling policy to be easily changed by swapping out the scheduler library for a different one with the same interface. Since the scheduler is a library, it can live outside the main compiler distribution, and this also permits tailoring it to application-specific requirements.

However, the interaction between user-level threading systems and the operating system services is difficult. For example, the Unix `write()` system call may block if the underlying buffer is full. While this would be fine in a sequential program or a program with each user-level thread mapped to a unique OS thread, a blocking system call would block the entire program where many user-level threads are multiplexed over a single OS thread. How then can we safely allow interaction between user-level threads and system services?

Concurrent Haskell [23], which is also a system with user-level threads, solves the problem with the help of specialised runtime system features such as safe FFI calls and bound threads for dealing with such interactions. However, implementing these features in the runtime system warrants that the scheduler itself be part of the runtime system, which is incompatible with our goal of writing thread schedulers in OCaml. Attempts to lift the scheduler from the runtime system to a library in the high-level language while retaining other features in the runtime system lead to further complications [30].

Our goals then are:

- Retain the simplicity of direct-style code for concurrent OCaml programs.
- Allow user-level thread schedulers for concurrent programs to be written in OCaml as libraries.
- Allow safe interaction between user-level threads and operating system services.
- Performance of the resultant code should be on par with or better than existing solutions.

We observe that algebraic effects and their handlers can meet all of these goals. In particular, we introduce *asynchronous effects* and their handlers, and show how they elegantly solve the interaction between user-level threads and operating system services. This paper makes the following contributions:

- We introduce effect handlers for Multicore OCaml and illustrate their utility by constructing a high-performance asynchronous I/O library that exposes a direct style API (Section 3).
- We show how *asynchronous effects* provide a clean interface to difficult-to-use operating system services, such as signal handling and asynchronous notification of I/O completion, and demonstrate how effect handlers enable scoped interrupt handling (Section 4).
- We evaluate the performance of effect handlers in OCaml by implementing a highly scalable web server and show that Multicore OCaml effect handlers are efficient (Section 5).

After the technical content of the paper in Sections 3, 4, and 5, we discuss related work in Section 6 and our conclusions and future work in Section 7.

3 Algebraic effects and their handlers

Since the primary motivation for adding effect handlers in Multicore OCaml is concurrency, we introduce effect handlers in constructing an asynchronous I/O library which retains the simplicity of direct-style programming [4].

3.1 Concurrency

We will start with an abstraction for creating asynchronous tasks and waiting on their results:

```
val async : (α → β) → α → β promise
(* [async f v] spawns a fiber to run [f v] asynchronously. *)

val await : α promise → α
```

[4] A comprehensive list of effect handler examples in Multicore OCaml is available at `https://github.com/kayceesrk/effects-examples`

```
(* Block until the result of a promise is available. Raises
    exception [e] if the promise raises [e]. *)

val yield : unit → unit
(* Yield control to other fibers. *)
```

We use the term *fiber* to indicate a user-level thread to distinguish it from kernel threads. Since `async`, `await` and `yield` are effectful operations, we declare the following effects:

```
effect Async : (α → β) * α → β promise
effect Await : α promise → α
effect Yield : unit
```

The first declaration says that `Async` is an effect which is parameterised by a pair of a thunk and a value, and returns a promise as a result when performed. `Await` is parameterised by a promise and returns the result. `Yield` is a nullary effect that returns a unit value. To be precise, these declarations are *operations* of a single built-in effect type `α eff` in Multicore OCaml. Indeed, the declarations themselves are syntactic sugar for extending the built-in effect type:

```
type _ eff +=
  | Async : (α → β) * α → β promise eff
  | Await : α promise → α eff
  | Yield : unit eff
```

We can now define the functions `async`, `await` and `yield` as:

```
let async f v = perform (Async (f,v))
let await p = perform (Await p)
let yield () = perform Yield
```

Effects are performed with the `perform : α eff → α` primitive, which performs the effect and returns the result. Observe that we have not provided an interpretation for these effects. Effects are interpreted by an effect handler, as shown in Figure 1.

A promise (lines 1–6) is either completed successfully `Done v`, failed with an exception `Error e` or still pending `Waiting l`, with a list of fibers waiting on it for completion. The function `run` (line 8) is the top-level function that runs the `main` concurrent program. `run_q` is the queue of concurrent fibers ready to run. The effect handler itself is defined in the lines 17–38. An effect handler comprises of five clauses – a value clause, an exception clause, and three clauses that handle the effects `Async`, `Await` and `Yield`.

Effect clauses are of the form `effect e k` where `e` is the effect and `k` is the continuation of the corresponding `perform` delimited by this handler. The delimited continuations are of type `(α ,β) continuation` which represents a continuation waiting for a value of type α and returns a value of type β when resumed.

In the case of an `Async (f,v)` effect (lines 28–31), we create a new promise value `p` which is initially waiting to be completed. We set up the original fibers, represented by continuation `k`, to resume with the promise using the `continue` primitive. Finally, we recursively call `fork` to run the new fiber `f v`. The handlers

```
type α _promise = Done of α | Error of exn
                | Waiting of (α, unit) continuation list

type α promise = α _promise ref

let run main v =
  let run_q = Queue.create () in
  let enqueue f = Queue.push f run_q in
  let run_next () =
    if Queue.is_empty run_q then ()
    else Queue.pop run_q ()
  in
  let rec fork : α β. α promise → (β → α) → β → unit =
    fun p f v →
      match f v with
      | v →
          let Waiting l = !p in
          List.iter (fun k →
            enqueue (fun () → continue k v)) l;
          p ≔ Done v;
          run_next ()
      | exception e →
          let Waiting l = !p in
          List.iter (fun k →
            enqueue (fun () → discontinue k e)) l;
          p ≔ Error e;
          run_next ()
      | effect (Async (f,v)) k →
          let p = ref (Waiting []) in
          enqueue (fun () → continue k p);
          fork p f v
      | effect (Await p) k →
          match !p with
          | Done v → continue k v
          | Error e → discontinue k e
          | Waiting l → p ≔ Waiting (k::l); run_next ()
      | effect Yield k →
          enqueue (fun () → continue k ());
          run_next ()
  in
  fork (ref (Waiting [])) main v
```

Fig. 1: A simple scheduler, implemented with effects

in Multicore OCaml are so-called *deep handlers* which implicitly enclose continuations, meaning that effects are handled uniformly. In the case of `Await p`, we check whether the promise is complete. If successfully completed, we immediately resume with the value. If the promise failed with an exception, the we use the `discontinue` primitive to resume the continuation by raising an exception. Otherwise, we block the current fiber on the promise and resume the next fiber from the scheduler. In the case of `Yield` effect, we enqueue the current fiber and run the next available fiber. In the case, of a fiber successfully running to completion (lines 18–23) or raising an exception (lines 24–29), we update the promise, wake up the waiting fibers and resume the next available fiber.

3.2 Implementing effect handlers

Unlike other languages that incorporate effect handlers, effects in Multicore OCaml are unchecked. That is, there is no static check for whether all the possible effects have been handled in the program. As a result, a fiber that performs an unhandled effect is discontinued with `Unhandled` exception.

There are several alternatives to implement the continuations in effect handlers including free monadic interpretations [17,18,34], CPS translations [14,20], and runtime strategies. Multicore OCaml chooses the latter and makes use of its own custom stack implementation, which are efficiently supported by the runtime system. In particular, we observe that many effect handlers do not resume the continuations more than once. Hence, we only support linear continuations by default, which can be implemented efficiently [10]. We also support explicit copying for non-linear use of continuations.

3.3 Adding I/O

Next let us add support for the following I/O operations:

```
val accept : file_descr → file_descr * sockaddr
val recv : file_descr → bytes → int → int
        → msg_flag list → int
val send : file_descr → bytes → int → int
        → msg_flag list → int
```

These functions have the *same* signature as their counterparts in the `Unix` module. However, it is important to note that invoking any of these functions may block the kernel thread until the I/O operation is complete. In a user-level threaded system such as the one we are considering this would block the scheduler, preventing other fibers from running.

The standard solution to this problem is to use an event loop, suspending each task performing a blocking I/O operation, and then multiplexing the outstanding I/O operations through an OS-provided blocking mechanism such as `select`, `epoll`, `kqueue`, `IOCP`, etc. Such asynchronous, non-blocking code typically warrants callback-oriented programming, making the continuations of I/O operations explicit through explicit callbacks (à la JavaScript) or concurrency monad

(Lwt and Async libraries for OCaml). This warrants a wholesale departure from synchronous direct-style code, and the resultant code is arguably messier and more difficult to understand.

Effect handlers lets us retain the direct-style while still allowing the use of event loops. For the sake of clarity, we shall just consider `accept`. The other functions are similar. As earlier, we start by declaring an effect for an `accept` function: `effect Accept : file_descr → (file_descr * sockaddr)`. The handler for `Accept` is:

```
| effect (Accept fd) k →
    if poll_rd fd then
      try continue k (Unix.accept fd)
      with e → discontinue k e
      else (block_accept fd k; run_next ())
```

We first poll the file descriptor `fd` to see whether it is available to read. If so, we immediately perform the blocking call (which is expected to succeed[5]), and resume the fiber with the result. However, if the call would block, then we record that the fiber is waiting to accept on `fd` and switch to the next thread from the scheduler queue. The `send` and `recv` operations have similar handler implementations.

```
let run_next () =
  if Queue.is_empty run_q then
    if io_is_pending () then begin
      wait_until_io_ready ();
      do_io ();
      run_next ()
    end else () (* done *)
  else Queue.pop run_q ()
```

Correspondingly, the `run_next` function is updated such that it first runs all the available threads, and then if any I/O is pending it waits until at least one of the I/O operations are ready, and then tries to perform the I/O and continue. If the scheduler queue is empty, and there are no pending I/O, then the scheduler returns.

Using this API, we can write a simple server that echoes client messages until client goes away as follows:

```
let rec echo_server sock =
  let sent = ref 0 in
  let msg_len = (* receive message *)
    try recv sock buffer 0 buf_size [] with
    | _ → 0 (* Treat exceptions as 0 length message *)
  in
  if msg_len > 0 then begin
    (* echo message *)
```

[5] In reality, the call might not succeed due to a variety of exceptional cases that must be handled for correctness [2]. But importantly, the client-facing API remains in a direct-style.

```
    (try while !sent < msg_len do
      let write_count =
        send sock buffer !sent (msg_len - !sent) [] in
      sent := write_count + !sent
    done with _ → ()); (* ignore send failures *)
    echo_server sock
  end else close sock (* client left, close connection *)
```

The details of the code are not important, but observe that the code is in direct-style and moreover is the *same* code for the synchronous, blocking echo server. Since the following code is asynchronous:

```
run (fun () →
  async echo_server sock1;
  async echo_server sock2) ()
```

it concurrently runs two echo servers on sockets `sock1` and `sock2` where neither blocks the other server.

3.4 Default handlers

For an invocation of an effectful operation to be meaningful it must happen in the scope of an appropriate handler. A default handler is a convenient mechanism for ensuring that an operation invocation is always meaningful even when not in scope of a handler. A default handler provides a *default* interpretation of an operation. This interpretation is chosen if no other appropriate handler encloses the invocation context. In other words, a default handler can operationally be understood as a top level handler which encloses the entire program context. As a concrete example we can give a default synchronous semantics for `Accept`

```
effect Accept : file_descr → (file_descr * sockaddr)
  with function Accept fd → Unix.accept fd
```

In Multicore OCaml a default handler is declared along with the effectful operation it is handling using the familiar `function` construct. In contrast to a regular effect handler, a default handler does not expose the continuation of the operation to the programmer, rather, the continuation is implicitly applied to the body clause(s). This particular design admits an efficient implementation, since every continuation invocation in a default handler is guaranteed to be in tail position. Thus the runtime does not need to allocate a continuation, it can simply return the value produced by the default handler clause. As a consequence an invocation of a default handler amounts to a function call. This makes it possible for effectful libraries to remain *performance backwards compatible* with programs that do not use regular effect handlers.

Continuing, we can also obtain the synchronous counterparts to `Await`, `Async`, and `Yield` by giving them all a default synchronous semantics, i.e.

```
effect Async : (α → β) * α → β promise
  with function Async (f, v) →
    match f v with
```

```
    | v → ref (Done v)
    | exception e → ref (Error e)

effect Await : α promise → α
  with function Await (ref (Done v)) → v
              | Await (ref (Error e)) → raise e

effect Yield : unit with function Yield → ()
```

If a default handler raises an exception, then the fiber is discontinued with that exception. Furthermore, if a default handler perform an effect then the default handler of that effect is invoked. If we define the default implementations of `Send` and `Recv` in a similar way then by using default handlers the program

```
async echo_server sock1; async echo_server sock2
```

behaves exactly like its synchronous counterpart.

4 Programming with resources and effects

Systems programming generally involves the manipulation of scarce resources such as file handles, connections and locks. Such resources are inherently linear, stateful values: once a file handle is closed, it cannot be used again.

Ordinary straight-line imperative code is not enough to use resources correctly in the presence of exceptions (let alone algebraic effects). For instance, the following code will leak an unclosed file handle if `do_stuff_with f` raises an exception:

```
let f = open_in "data.csv" in
do_stuff_with f;
close_in f
```

It is necessary to ensure that the filehandle is closed, even if an exception is raised:

```
let f = open_in "data.csv" in
match do_stuff_with f with
| () → close_in f
| exception e → close_in f; raise e
```

Note that the initial `open_in` occurs outside the exception handler - if opening the file fails with an exception, it is not necessary to close it. This idiom or something equivalent to it is standard in many languages, often with syntactic support as `try-finally`.

However, note an implicit assumption in this code, that if `do_stuff_with f` terminates then it does so only once. If the computation `do_stuff_with f` were somehow to return twice, then the cleanup code (`close_in f` in this example) would incorrectly run multiple times. If the computation `do_stuff_with_f` were to continue execution after the cleanup code had run, its operations would have unexpected effects.

As well as the performance advantages mentioned above, this is the other major reason that our continuations are linear. By preserving the linearity of computations (operations that are begun once do not complete twice), we allow resource-manipulating code to work correctly in the presence of effects.

Some interesting examples of programming with effects and handlers (such as backtracking) are incompatible with this approach, since they rely on continuations to be usable more than once. To support experimenting with such examples, we do provide a primitive to allow re-use of continuations, with the proviso that it is not safe in general when used with code that handles resources.

The linearity of computations is implicit in OCaml without effect handlers, but once continuations appear as first-class values the possibility of using them twice arises. OCaml does not have the linear types necessary to prevent this statically (and we are not proposing to add them), so we must enforce linearity dynamically. Ensuring that a continuation is not used twice is easy enough, by keeping a bit of state in the continuation, updated by `continue` and `discontinue` so that subsequent resumptions fail. Ensuring that a continuation is not simply discarded is harder: the system must detect when a continuation is being garbage-collected, and `discontinue` it with a special exception so that resource cleanup code runs.

4.1 Asynchronous exceptions

Correct use of resources is much more difficult in the presence of *asynchronous exceptions*. For example, on Unix-like systems when the user of a command-line program presses `Ctrl-C` the `SIGINT` signal is sent to the running program. By default, this terminates the program. However, programs may indicate that they can handle this signal, for instance by cancelling a long-running unresponsive task and accepting user input again.

In OCaml, programs indicate willingness to handle `SIGINT` by calling `Sys.catch_break true`. From that point onwards, the special exception `Sys.Break` may be raised at essentially any point in the program, if the user presses `Ctrl-C`. Unfortunately, the usual try-finally idiom is not enough to correctly clean up resources in this case:

```
let f = open_in "data.csv" in
match do_stuff_with f with
| () → close_in f
| exception e → close_in f; raise e
```

If `Sys.Break` is raised just after `open_in` returns but before the `match` statement is entered, then the filehandle will never be closed. What is needed is a means of temporarily disabling asynchronous exceptions, to eliminate this possibility. Suppose we introduce two functions `set_mask` and `clear_mask`, to disable (*mask*) and re-enable asynchronous exceptions. Our second attempt at resource handling looks like:

```
set_mask ();
let f = open_in "data.csv" in
```

```
match clear_mask (); do_stuff_with f; set_mask () with
| () → close_in f; clear_mask ()
| exception e → set_mask (); close_in f; clear_mask ();
    raise e
```

Correctly placing calls to `set_mask` and `clear_mask` is a very tricky business. Indeed, the above code has a serious bug: if `open_in` fails with an ordinary synchronous exception (because e.g. the file is not found), then asynchronous exceptions will never be unmasked.

Instead, we follow the good advice of Marlow et al. in the design of Haskell's asynchronous exceptions [22], and prefer instead *scoped combinators*:

```
mask (fun () →
  let f = open_in "data.csv" in
  match unmask (fun () → do_stuff_with f) with
  | () → close_in f
  | exception e → close_in f; raise e)
```

The changes to the masking state made by `mask` and `unmask` apply only to one scope, and are automatically undone, making it impossible to accidentally leave asynchronous exceptions masked.

4.2 Signal handling and asynchronous effects

Even with the scoped masking combinators, it is difficult and unpleasant to write code that correctly manipulates resources in the presence of asynchronous exceptions. For this reason, many systems choose instead to poll for cancellation requests, instead of being interrupted with one. That is, instead of a `Sys.Break` exception being raised at an arbitrary point, the programmer manually and regularly checks a mutable boolean `cancellation_flag`, which is asynchronously set to `true` when the user presses `Ctrl-C` (This check may be combined with other system facilities: one common choice is that cancellation is checked at all I/O operations, since the program must handle failures there anyway).

On Unix-like systems, it is possible to implement this behaviour using a *signal handler*, which is a callback invoked when a signal is raised (e.g. by the user pressing `Ctrl-C`). In OCaml, these can be installed using the function `Sys.set_signal`. In fact, the behaviour of the previously-mentioned `Sys.catch_break` is implemented by installing a signal handler that raises `Sys.Break`:

```
set_signal sigint (Signal_handle(fun _ → raise Break))
```

Synchronous cancellation can be implemented using a signal handler that sets a cancellation flag:

```
let cancellation_flag = ref false
let is_cancelled () = !cancellation_flag
let () =
  set_signal sigint (Signal_handle(fun _ →
    cancellation_flag := true))
```

By removing the possibly of cancellation except at designated points, the imperative parts of the system become safer and easier to write. However, as Marlow et al. [22] note, for the purely functional parts of the system asynchronous cancellation is both necessary and just as safe as synchronous: necessary, because inspecting the mutable cancellation state breaks referential transparency, and safe, because purely functional code holds no resources and pure computations can be abandoned without issue.

In order to call a pure function from imperative code while maintaining prompt cancellation, we need to switch from synchronous (polling) cancellation to asynchronous cancellation and back, by providing a combinator:

```
async_cancellable : (unit → α) → (unit → α option)
```

Normally, `async_cancellable f` returns `Some (f ())`. However, the computation `f` may be cancelled asynchronously, causing `async_cancellable f` to return `None`, ensuring that asynchronous cancellation does not affect the caller.

Our first attempt at such a mechanism looks like:

```
let sync_handler = Signal_handle (fun _ →
  cancellation_flag := true)
let async_handler = Signal_handle (fun _ →
  raise Break)
let async_cancellable f =
  mask (fun () →
    match
      set_signal sigint async_handler;
      let result = unmask f in
      set_signal sigint sync_handler;
      result
    with
    | x → Some x
    | exception Break → None)
```

Unfortunately, getting this code right is tricky, due to the delicate mutation of the global state representing the "current signal handler". In this respect, it is very similar to the code we saw earlier using `set_mask` and `clear_mask` (and even has the same bug: this code leaves the wrong signal handler in place if `f` raises an exception).

As before, scoped combinators make such code easier to get right (or, more accurately, harder to get wrong). To this end, we introduce *asynchronous effects*, which are effects that can be performed asynchronously, just as asynchronous exceptions can be raised asynchronously. By treating `Break` as an asynchronous effect, we can mix synchronous and asynchronous cancellation reliably.

As before, the imperative code using synchronous cancellation handles the `Break` effect by setting the cancellation flag:

```
let cancellation_flag = ref false
let sync_cancellable f =
  mask (fun () →
    match unmask f with
```

```
    | result →
      result
    | effect Break k →
      cancellation_flag ≔ true; continue k ())
```

Asynchronously-cancellable code can be implemented by handling the `Break` effect and discarding the continuation. Since effect handlers delimited continuation, the asynchrony is limited to the specified function.

```
let async_cancellable f =
  mask (fun () →
    match unmask f with
    | result → Some result
    | effect Break k → None)
```

Instead of having a single global callback as the current signal handler, asynchronous effects allow handlers to delimit their scope and nest correctly.

4.3 Managing multiple computations with asynchronous effects

Unlike signal handlers, asynchronous effects get an explicit representation of the computation they interrupted by way of the continuation `k`. While signal handlers can only resume the computation or abandon it (by raising an exception), effect handlers have other options available. For instance, a scheduler which maintains a collection of tasks can switch to another task upon receipt of an asynchronous effect, just as the scheduler in Fig. 1 does upon receipt of the synchronous `Yield` effect. That is, using asynchronous effects the cooperative scheduler of Fig. 1 can be made preemptive, by asking the operating system to provide a periodic timer signal (using e.g. the Unix `timer_create` API), and adding a new effect clause to the scheduler:

```
| effect TimerTick k →
  enqueue (fun () → continue k ());
  run_next()
```

4.4 Asynchronous I/O notifications

Operating systems provide a number of different interfaces with which to perform I/O. The simplest is the direct-style *blocking I/O*, in which the program calls I/O functions provided by the operating system, which do not return until the operation completes. This allows a straightforward style of programming in which the sequence of I/O operations matches the flow of the code. We aim to preserve this style of programming, but implement it using alternative operating system interfaces that allow multiple I/O operations to be overlapped.

In Section 3.3, we saw one way of accomplishing this with effects, by using operating system multiplexing mechanisms like `select`, `poll`, etc., which block until one of several file descriptors is ready. An alternative interface is *asynchronous I/O*, in which multiple operations are submitted to the operating system, which

overlaps their execution. However, applications written using asynchronous I/O tend to have complex control flow which does not clearly explain the logic being implemented, due to the complexity of handling the operating system's asynchronous notifications of I/O completion.

We propose effects and handlers as a means of writing direct-style I/O code, but using the asynchronous operating system interfaces. We introduce two new effect operations: `Delayed`, which describes an operation that has begun and will complete later, and `Completed`, which describes its eventual completion. Both of these take an integer parameter, which is an ID number identifying the particular operation.

Potentially long-running operations like `read` perform the `Delayed` effect, indicating that the operation has been submitted to the operating system but has not yet completed. Later, upon receipt of an operating-system completion notification, the asynchronous effect `Completed` is performed.

Using this mechanism, support for asynchronous completions can be added to the scheduler of Fig. 1 by adding clauses for the `Delayed` and `Completed` effects, where `ongoing_io` is an initially empty hash table:

```
| effect (Delayed id) k →
  Hashtbl.add ongoing_io id k
| effect (Completed id) k →
  let k' = Hashtbl.find ongoing_io id in
  Hashtbl.remove ongoing_io id;
  enqueue (fun () → continue k ());
  continue k' ()
```

In this sample, the continuation `k` of the `Delayed` effect is the continuation of the code performing the I/O operation, which instead of being immediately invoked is stored in a hash table until it can be invoked without blocking.

The continuation `k` of the `Completed` effect is the continuation of whichever fiber was running when the I/O completed. This scheduler chooses to preempt that fiber in favour of the fiber that performed the I/O, by retrieving the continuation `k'` from the hashtable and continuing it. Equally, the scheduler's policy could be to give priority to the already running fiber, by swapping `k` and `k'` in the last two lines.

5 Results

So far we have presented what we believe are compelling applications of effect handlers for elegant system programming. However, none of that would matter if the resultant programs were unacceptably slower compared to extant solutions. Hence, in this section, we evaluate the performance of a web server built with effect handlers against existing production-quality web servers.

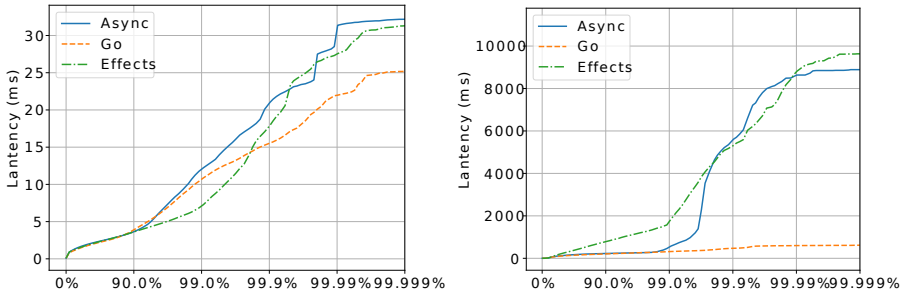
We have implemented an effect-based asynchronous I/O library, `aeio` [2], that exposes a direct-style API to the clients. At its core, `aeio` uses the main loop engine from Lwt library using the libev[6] event loop (using *epoll* in our experi-

[6] `http://software.schmorp.de/pkg/libev.html`

ments). For the OCaml web server, we use `httpaf`, which is a high performance, memory efficient, and scalable web server that uses the Async library [24] (also using *epoll* as its I/O system call). We then extended `httpaf` and implemented an effect handler based backend using `aeio`. The configurations we use for the evaluation are:

- **Effect:** Effect-based version which uses `httpaf` with `aeio` on the Multicore OCaml compiler. Though the compiler is multicore capable, the benchmark only utilises a single core. The Multicore OCaml compiler was forked off vanilla OCaml version 4.02.2.
- **Async:** This configuration is `httpaf` + Async 113.33.03 on vanilla OCaml version 4.03.0. Since vanilla OCaml has a global runtime lock, this also uses one core.
- **Go:** Go 1.6.3 version of the benchmark using `net/http` package. The `GOMAXPROCS` variable was used to constrain this to run on one core only, to be comparable with the others above.

The evaluations were done on a 3 GHz Intel Core i7 with 16 GB of main memory running 64-bit Ubuntu 16.10. The client workload was generated by the `wrk2`[7] program. Each `wrk2` run establishes a fixed number of client connections and then issue requests at a constant rate, and measures the request latency and throughput.



(a) Medium contention: 1k connections, 10k requests/sec

(b) High contention: 10k connections, 30k requests/sec

Fig. 2: Latency profile of client requests

Figure 2 shows the latency profiles for 1 minute runs under two different configurations. At 1k connections and 10k requests per second, the effect implementation performs marginally better than Async. Go performs the best with all requests satisfied within 27 ms. The average request latency of effect configuration is 2.127 ms over 587969 requests. Under this configuration, the observed

[7] `https://github.com/giltene/wrk2`

throughput is between 9780 and 9800 requests per second in all of the configurations.

At high loads, the performance degrades substantially in every configuration, but it is worse in the OCaml implementations. The average latency for satisfying a client request increases to 333.40 ms in the effect case, while it is 139 ms in async and 107.25 ms in Go. While Go achieved 17389 requests per second, Async and effect implementations achieved only 16761 and 15440 requests per second, respectively. This indicates that there is room for optimizations. Multicore OCaml has a new garbage collector, which has not been tuned to the extent of vanilla OCaml and Go. We strongly suspect that garbage collector optimisation and tuning would lead to performance improvements.

Importantly, the tail latencies of both OCaml implementations (the vanilla Async and our effect-based server) were comparable in both configurations, indicating that there is no significant performance degradation from our switch to using the effects model presented in this paper.

6 Related work

Implementations of effect handlers Since their inception, several implementations of algebraic effect handlers have appeared, many of which are implemented as libraries in existing programming languages [6,16,17,18,19,29,34]. There are several other implementations that like Multicore OCaml provide language level support for effect handlers:

- Eff [5] is the first programming language designed with effect handlers in mind. It is a strict language with Hindley-Milner type inference similar in spirit to ML. It includes a novel feature for supporting fresh generation of effects in order to support effects such as ML-style higher-order state. The language has an OCaml "look-and-feel". Incidentally, it is compiled to a free monadic embedding in OCaml.
- Frank [21] is a programming language with effect handlers but no separate notion of function: a function is but a special case of a handler. Frank has a bidirectional type and effect system with a novel form of effect polymorphism. Furthermore, the handlers in Frank are so-called *shallow handlers*. In contrast to deep handlers, shallow handlers do not implicitly wrap themselves around the continuation, thereby allowing nonuniform interpretations of operations.
- Koka is a functional web-oriented programming language which has recently been enriched with effect handlers [20]. It has a type-and-effect system which is based on row polymorphism. Koka uses a novel type-and-effect driven selective CPS compilation scheme for implementing handlers on managed platforms such as .NET and JavaScript.
- Links [8] is a single source, statically typed language with effect tracking for multi-tier web programming with three backends: a JavaScript compiler for client side code, an interpreter for the server side, and an SQL generator

for the database. Links supports effect handlers on both the client and the server. The server side implementation is based on a generalised abstract CEK machine [13], while the client side implementation is based on a CPS translation [14]. Links also has a prototype compiler for the server side with effect handlers based on the Multicore OCaml compiler [15].

A common theme for the above implementations is that their handlers are *multi-shot* handlers which permit multiple invocations of continuations.

Asynchronous IO Many systems seek to combine the simplicity of direct-style, blocking I/O with the performance gains of allowing independent operations to complete in parallel. Indeed, the blocking I/O interfaces of most operating systems are designed in this way, by descheduling a process that requests a slow operation and running another process until the operation completes.

However, operating system mechanisms rely on hardware context switching. The high overheads of such mechanisms lead to a desire for lightweight concurrent tasks integrated into programming languages.

The Erlang system [3] is a good example, capable of managing large numbers of lightweight processes with an integrated scheduler, and multiplexing their I/O onto operating system interfaces like `select`, `poll`, etc. More recently, the work by Syme et al. [31] adding `async/await` to F# allows the programmer to specify which operations should be completed asynchronously, implemented by compiling functions which use `async` differently from those that do not. The work by Marlow et al. on Concurrent Haskell [23] also supports large numbers of concurrent threads with multiplexed I/O, while allowing possibly-blocking operating system services to be used without blocking the entire system via the mechanism of *safe foreign calls*.

Resource handling with control operators Programming languages supporting systems programming and exceptions generally support some variant of the `try-finally` idiom, often with special syntactic support (e.g. `try{...}finally{...}` in Java, `using` statements in C#, destructors and RAII in C++, or `defer` in Go).

Languages with more powerful control operators require correspondingly more powerful constructs for safe resource handling. The Common LISP condition system allows conditions (similar to effects) to be handled by abandoning the computation with an error, restarting it, ignoring the error and continuing, but does not allow the continuation to be captured as a value. It supports the `unwind-protect` form to ensure that cleanup code is run no matter how a block is exited. (See Pitman [26] for an analysis of the condition system's design).

Scheme supports general nonlinear continuations [1], which present difficulties when handling inherently linear resources. Many Scheme implementations provide a primitive `dynamic-wind` [12], which generalises the try-finally idiom by taking some setup and cleanup code to be run not just once but every time control passes in and out of a specified block of code. However, this comes with its own caveats: the naive approach of using `dynamic-wind` to open and close a file will close and reopen the file every time a computation is paused and resumed,

which is not safe in general as there is no guarantee that the file still exists at the second opening. One-shot (linear) continuations have also been proposed for Scheme [7].

Support for truly asynchronous interrupts is more rare, partially due to the difficulty of programming in their presence. The Unix signalling mechanism is an important example, but its reliance on global mutable state makes programming difficult (see Section. 4.2). Marlow et al. [22] present a more composable design for asynchronous interrupts in their work on asynchronous exceptions for Haskell. (Our approach is heavily based on theirs, and can be viewed as the extension of the Haskell approach to effects as well as exceptions).

7 Conclusions and future work

Multicore OCaml provides effect handlers as a means to abstract concurrency. In this paper, we have described and demonstrated the utility of effect handlers in concurrent system oriented programming. We have developed a direct-style asynchronous I/O with effect handlers [2]. Using this library, we built a highly concurrent and scalable web server. Our evaluation shows that this implementation retains a comparative performance with the current state of the art in vanilla OCaml, but that OCaml has some room for improvement *vs* direct-style multicore concurrency in Go.

Rather than providing the full generality of effect handlers with nonlinear continuations, our design provides effect handlers with linear continuations. This design admits a particularly efficient implementation. Furthermore, linear continuations interplay more smoothly with system resources. Our implementation of asynchronous effects also provides an elegant solution to handling problematic corner cases in typical operating system interfaces, such as reliable signal handling and efficiently implementing quirky system call interfaces while exposing a simple, portable interface to the developer.

Acknowledgements

The third author was supported by EPSRC grant CDT in Pervasive Parallelism (EP/L01503X/1).

References

1. Abelson, H., Dybvig, R., Haynes, C., Rozas, G., Adams, N., Friedman, D., Kohlbecker, E., Steele, G., Bartley, D., Halstead, R., Oxley, D., Sussman, G., Brooks, G., Hanson, C., Pitman, K., Wand, M.: Revised5 report on the algorithmic language scheme. Higher-Order and Symbolic Computation 11(1), 7–105 (1998)
2. Aeio: An asynchronous, effect-based I/O library (2017), `https://github.com/kayceesrk/ocaml-aeio`, accessed: 2017-05-05 09:21:00
3. Armstrong, J., Virding, R., Wikström, C., Williams, M.: Concurrent programming in erlang (1993)

4. Barroso, L., Marty, M., Patterson, D., Ranganathan, P.: Attack of the killer microseconds. Commun. ACM 60(4), 48–54 (Mar 2017), http://doi.acm.org/10.1145/3015146
5. Bauer, A., Pretnar, M.: Programming with algebraic effects and handlers. J. Log. Algebr. Meth. Program. 84(1), 108–123 (2015)
6. Brady, E.: Programming and reasoning with algebraic effects and dependent types. In: ICFP. pp. 133–144. ACM (2013)
7. Bruggeman, C., Waddell, O., Dybvig, R.K.: Representing control in the presence of one-shot continuations. In: Fischer, C.N. (ed.) Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation (PLDI), Philadephia, Pennsylvania, May 21-24, 1996. pp. 99–107. ACM (1996)
8. Cooper, E., Lindley, S., Wadler, P., Yallop, J.: Links: Web programming without tiers. In: FMCO. LNCS, vol. 4709, pp. 266–296. Springer (2006)
9. Dolan, S., White, L., Madhavapeddy, A.: Multicore OCaml. OCaml Workshop (2014)
10. Dolan, S., White, L., Sivaramakrishnan, K., Yallop, J., Madhavapeddy, A.: Effective concurrency through algebraic effects. OCaml Workshop (2015)
11. Forster, Y., Kammar, O., Lindley, S., Pretnar, M.: On the expressive power of user-defined effects: Effect handlers, monadic reflection, delimited control (2017), to appear in ICFP'17
12. Friedman, D.P., Haynes, C.T.: Constraining control. In: Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. pp. 245–254. POPL '85, ACM (1985)
13. Hillerström, D., Lindley, S.: Liberating effects with rows and handlers. In: TyDe@ICFP. pp. 15–27. ACM (2016)
14. Hillerström, D., Lindley, S., Atkey, R., Sivaramakrishnan, K.: Continuation passing style for effect handlers. http://homepages.inf.ed.ac.uk/slindley/papers/handlers-cps-draft-april2017.pdf (Apr 2017), draft
15. Hillerström, D., Lindley, S., Sivaramakrishnan, K.: Compiling Links effect handlers to the OCaml backend. ML Workshop (2016)
16. Kammar, O., Lindley, S., Oury, N.: Handlers in action. In: ICFP. pp. 145–158. ACM (2013)
17. Kiselyov, O., Ishii, H.: Freer monads, more extensible effects. In: Haskell. pp. 94–105. ACM (2015)
18. Kiselyov, O., Sabry, A., Swords, C.: Extensible effects: an alternative to monad transformers. In: Haskell. pp. 59–70. ACM (2013)
19. Kiselyov, O., Sivaramakrishnan, K.: Eff directly in OCaml. ML Workshop (2016)
20. Leijen, D.: Type directed compilation of row-typed algebraic effects. In: POPL. pp. 486–499. ACM (2017)
21. Lindley, S., McBride, C., McLaughlin, C.: Do be do be do. In: POPL. pp. 500–514. ACM (2017)
22. Marlow, S., Jones, S.P., Moran, A., Reppy, J.: Asynchronous exceptions in Haskell. In: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation. pp. 274–285. PLDI '01, ACM, New York, NY, USA (2001), http://doi.acm.org/10.1145/378795.378858
23. Marlow, S., Jones, S.P., Thaller, W.: Extending the Haskell foreign function interface with concurrency. In: Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell. pp. 22–32. Haskell '04, ACM, New York, NY, USA (2004), http://doi.acm.org/10.1145/1017472.1017479

24. Minsky, Y., Madhavapeddy, A., Hickey, J.: Real World OCaml - Functional Programming for the Masses. O'Reilly (2013), http://shop.oreilly.com/product/0636920024743.do#tab_04_2
25. Moggi, E.: Notions of computation and monads. Inf. Comput. 93(1), 55–92 (1991)
26. Pitman, K.M.: Condition Handling in the Lisp Language Family, pp. 39–59. Springer Berlin Heidelberg, Berlin, Heidelberg (2001)
27. Plotkin, G.D., Power, J.: Adequacy for algebraic effects. In: FoSSaCS. LNCS, vol. 2030, pp. 1–24. Springer (2001)
28. Plotkin, G.D., Pretnar, M.: Handling algebraic effects. Logical Methods in Computer Science 9(4) (2013)
29. Saleh, A.H., Schrijvers, T.: Efficient algebraic effect handlers for Prolog. TPLP (2016), proceedings of ICLP
30. Sivaramakrishnan, K., Harris, T., Marlow, S., Peyton Jones, S.: Composable scheduler activations for Haskell. Journal of Functional Programming 26 (2016)
31. Syme, D., Petricek, T., Lomov, D.: The f# asynchronous programming model. In: International Symposium on Practical Aspects of Declarative Languages. pp. 175–189. Springer (2011)
32. Vouillon, J.: Lwt: A cooperative thread library. In: Proceedings of the 2008 ACM SIGPLAN Workshop on ML. pp. 3–12. ML '08, ACM, New York, NY, USA (2008), http://doi.acm.org/10.1145/1411304.1411307
33. Wadler, P.: The essence of functional programming. In: POPL. pp. 1–14. ACM (1992)
34. Wu, N., Schrijvers, T., Hinze, R.: Effect handlers in scope. In: Haskell. pp. 1–12. ACM (2014)