# Effectively Composing Concurrency Libraries

DEEPALI ANDE, IIT Madras, India
SUDHA PARIMALA, Tarides, India
KC SIVARAMAKRISHNAN, Tarides and IIT Madras, India

Effect handlers have proved to be a versatile mechanism for modular programming with user-defined effects. Effect handlers permit non-local control-flow mechanisms such as generators, async/await, coroutines and lightweight threads to be composably expressed. There is increasing interest in supporting effect handlers in industrial-strength languages. The recent major release of OCaml, version 5, supports effect handlers as the primary mechanism for expressing user-level concurrency. Several concurrency libraries have already been designed around effect handlers. However, these libraries are designed monolithically, with their own notion of tasks and mechanisms for inter-task synchronisation. Under this monolithic approach, we face the risk that different concurrency libraries will be incompatible preventing a program from taking advantage of several libraries in the same application. Such is the case with OCaml today with the Lwt and Async libraries with the library ecosystem incompatibly split between the libraries building over either Lwt or Async.

In this paper, we observe that the composability of effect handlers permits the composability of concurrency libraries. The key idea is to define a uniform yet expressive interface for suspending and resuming tasks, which is implemented by different schedulers. Against this interface, we implement *scheduler-agnostic synchronisation structures* that permit tasks from different concurrency libraries to interact. We also show how to extend this interface to support composition with monadic concurrency libraries such as Lwt and Async. We show how to extend this interface to support various forms of thread cancellation. Finally, we show how this interface helps in safely sharing lazy computations between different concurrency abstractions provided by OCaml including user-level and OS threads.

**ACM Reference Format:**
Deepali Ande, Sudha Parimala, and KC Sivaramakrishnan. 2023. Effectively Composing Concurrency Libraries. In . ACM, New York, NY, USA, 25 pages. https://doi.org/XXXXXXX.XXXXXXX

## 1 INTRODUCTION

Many modern programming languages provide language-level support for non-local control-flow primitives such as async/await, generators, coroutines, lightweight threads, etc. While some concurrency primitives such as JavaScript generators [JSGenerators 2023], C# async/await [C# async/await 2023] and Kotlin coroutines [Kotlin coroutines 2023] are implemented with program transformations, there is an increasing trend towards supporting *true* concurrency native in the language. This is because of the inherent cost associated with the program transformation in order to support suspending and resuming tasks as well as the necessity to do something special for supporting features that inspect the program stack such as backtraces and exceptions. These language-level primitives introduce *function colouring* which splits the world between the asynchronous primitives which may suspend execution and synchronous primitives which won't [Function Colour 2023].

Alternatively, many languages provide concurrency primitives which are *stackful*, where the runtime system provides support for managing and switching between multiple stacks. They can

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
*Conference acronym 'XX, June 03–05, 2018, Woodstock, NY*
© 2023 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

be broadly split into two groups – one which bakes-in the thread scheduler into the runtime system and the other which permits schedulers to be written as libraries. For example, the Go programming language and GHC Haskell support lightweight language-level threads (called *goroutines* in Go and *threads* in GHC Haskell) which are managed by the runtime system. On the other hand, languages that provide support for schedulers to be written as libraries expose some form of *continuations* at the language level. Java has recently introduced support for virtual threads [Java Virtual threads 2023], which is built on top of delimited continuations [Loom 2023]. The recent release of the OCaml programming language supports effect handlers [Sivaramakrishnan et al. 2021], which offers a structured primitive for programming with delimited continuations.

There are several downsides to having the scheduler baked into the runtime system as is the case with Go and GHC Haskell. The runtime system not only supports the threads and its scheduler, but also implementations of synchronisation primitives such as locks, condition variables, timers, IO event loop, thread pools, etc. These primitives are often implemented in a low-level language (C in the case of GHC Haskell), which makes it hard to maintain the thread subsystem. This also makes it harder to evolve the language as the changes to the thread subsystem can only be released as part of the language release [KC et al. 2016].

Moreover, there is no one-size-fits-all scheduling policy for all the programs written in a language. For example, a nested parallel computation such as computing the nth Fibonacci number recursively will benefit from a work-stealing scheduler whereas a scheduler for a web server will benefit from a first-in-first-out (FIFO) scheduling of outstanding requests in order to minimise overall latency. Additionally, each style of application may need their own notion of cancellation of outstanding tasks. For example, a parallel depth first search procedure or a large graph may want to terminate all the outstanding search threads once a result has been found. On the other hand, it is preferable to have *structured concurrency* [JEP428 2023] for IO-bound tasks in order to properly clean up resources.

It is with this goal that OCaml 5 introduces effect handlers in order for thread scheduling and concurrency primitives to be implemented as libraries rather than baking it into the runtime. The key idea with effect handlers is to permit effectful operations to be declared and used in a computation without defining how the operations are handled. The meaning of the operations is given by *handlers* of the effects, akin to how exception handlers define how exceptions thrown by a computation are handled. Effect handlers permit handlers of different operations to be expressed modularly and composed together similar to how functions handling different exceptions can be composed together.

The primary motivation to extend OCaml with effect handlers is to permit *direct style* concurrency, as opposed to using monadic concurrency [Function Colour 2023] libraries such as Lwt [Vouillon 2008] and Async [Async 2023], where asynchronous functions are represented as monadic computations. Monadic concurrency libraries not only introduce function colours [Function Colour 2023] that split the API between being synchronous and asynchronous but also often end up specialising the asynchronous APIs to the specific monad, Lwt or Async, due to the lack of higher-kind of polymorphism in OCaml which makes it cumbersome to write code that remains parametric over the concurrency monad. As a result, in OCaml today, one either needs to choose the Lwt or the Async ecosystem and can only use libraries from that ecosystem.

## 1.1 Challenge

The promise of effect handlers is that such an ecosystem split need not happen while also allowing specialisation of schedulers. Indeed, several libraries have already been written utilising effect handlers, which take advantage of the ability to specialise scheduling.

Eio [Eio 2022] is a library that provides a multicore-capable, direct-style IO stack for OCaml. Eio adopts a work-sharing model where the lightweight user-level tasks, *fibers* in Eio parlance[1], may be pushed onto other cores, but the tasks remain pinned to the core that they were spawned on. Eio provides structured concurrency [JEP428 2023] through a notion of *switches* such that the tasks form a tree-structured hierarchy that ensures that resources are cleaned up deterministically. Eio switches also serve as the notion of task cancellation. When a switch is cancelled, all the tasks that are attached to this switch get cancelled and their resources such as open file descriptors and sockets are close. Structured concurrency leads to cleaner code and avoids resource leaks. Domainslib [Domainslib 2022] is a library for nested parallel computation. Unlike Eio, Domainslib uses a work-stealing scheduler that automatically schedules tasks on idle cores. Given the target application, Domainslib neither provides structured concurrency nor does it offer cancellation mechanisms.

While effect handlers permit rich concurrency libraries to be implemented, the fundamental problem is that each of these libraries end up implementing their own incompatible notion of tasks and inter-task synchronisation. Eio provides streams which are bounded queues where taking from an empty stream blocks the Eio task. Domainslib provides async/await mechanism to wait for task completion. Neither of these mechanisms are aware of any other tasks other than the tasks from their own libraries. It is conceivable that an application such as the Tezos blockchain node may want to utilise both of these libraries at the same time, using Eio for the network operations while offloading compute-intensive serialisation and cryptographic primitives to Domainslib. Alas, one cannot build such an application today that utilises both Eio and Domainslib.

This problem of concurrency library composition is not unique to the OCaml ecosystem. Any programming language that provides the ability to implement their own lightweight thread subsystem will need to handle this issue. Rust programming language provides language-level support for marking asynchronous computations using the `async` keyword. The compiler transforms the async functions into a state machine that represents suspendable computations. Rust does not have a default scheduler for asynchronous tasks and instead relies on libraries called *async runtimes* execute asynchronous applications. As a result, not only does Rust suffer from the problem of function colours, but also suffers incompatibility between asynchronous runtimes [Async Rust Book 2023].

## 1.2 Solution

We observe that modularity of effect handlers helps us abstract away from the details of schedulers. For example, in OCaml 5, we can declare the following effects for forking and yielding tasks:

```
1 type _ Effect.t += Fork : (unit -> unit) -> unit Effect.t
2                 | Yield : unit Effect.t
```

The `Fork` effect takes a thunk which is spawned as a concurrent thread, and the `Yield` effect yields control to another thread in the scheduler queue. We can define helper functions to `perform` these effects:

```
1 let fork (f : unit -> unit) : unit = perform (Fork f)
2 let yield () : unit = perform Yield
```

The type annotations are not necessary and are only included for clarity. Observe that concurrent programs may call the functions `fork` and `yield` without knowing how they are implemented. The

[1]In this paper, we use *tasks* as a common terminology to represent lightweight threads created and managed by different concurrency libraries

implementation is described by each of the concurrency libraries which implement a handler for the `Fork` and `Yield` effects.

In this work, we propose a solution for the concurrency library composition problem by describing a single `Suspend` effect that captures the core details of the threading subsystem. The interface captures the effect of suspending and resuming tasks without appealing to the details of the scheduler implementation. The different concurrency libraries will have to implement a handler only for this effect in order to make them composable with other concurrency libraries.

While our core proposal is astonishingly simple, we show that this one effect is able to capture the complexity of full-fledged concurrent programming support in OCaml. On top of the `Suspend` effect, we show how to implement thread-safe, scheduler-agnostic synchronisation structures such as promises, MVars [Peyton Jones et al. 1996], mutexes, condition variables, channels, etc. These synchronisation structures may be used to communicate between tasks that belong to different libraries. We show how to extend the `Suspend` effect to capture different, concurrency library specific implementation of task cancellation.

Apart from new libraries such as Eio and Domainslib that take advantage of effect handlers available in OCaml 5, the OCaml ecosystem has millions of lines of legacy code written using monadic concurrency libraries such as Lwt and Async. Hence, it is conceivable that these monadic libraries will continue to be used into the future even when the users switch to OCaml 5. We show how to compose newer effect-based libraries with monadic concurrency libraries using the `Suspend` effect, and thereby enabling an incremental transition of code using monadic concurrency to direct-style. Our solution also helps reconcile the conflict between concurrency and lazy evaluation in OCaml. While OCaml has primitive support for lazy evaluation [OCaml Lazy 2023], it is not concurrency-safe.We show how effect handlers enable a graceful solution for concurrency-safe lazy without appealing to a particular concurrency library.

### 1.3 Contributions

Our contributions are as follows.

- The design of a single `Suspend` effect that succinctly captures the core details of threading subsystem. Each concurrency library implementing a handler for this effect makes them composable with other concurrency libraries.
- We show how to implement scheduler-agnostic thread-safe synchronisation structures such as IVars, MVars, rendezvous channels, mutex and condition variables on top of this effect without appealing to the details of individual schedulers. These synchronisation structures can be concurrently utilised by tasks from different concurrency libraries.
- We show how to extend the `Suspend` effect in order to enable variety of task cancellation strategies to co-exist.
- We show how the `Suspend` effect enables the composition of legacy monadic concurrency libraries such as Lwt and Async with effect-based concurrency libraries.
- We illustrate that the `Suspend` effect enables a graceful, backwards-compatible solution for making OCaml lazy values concurrency-safe.
- Extensive experimental evaluation shows that the composition of concurrency libraries using our solution incurs negligible overheads compared to non-composable alternatives and offers impressive performance improvements in applications where composition is necessary.

The rest of the paper is organised as follows. The next section 2 motivates the need for composition of concurrency libraries, followed by our solution in Section 3. We then extend our solution to support task cancellation in Section 4. Section 5 shows how to compose legacy monadic-concurrency

```
module T = Domainslib.Task

(* set up a pool of [num_domains] domains for parallel computation *)
let pool = T.setup_pool ~num_domains ()

(* Parallel Fibonacci computation *)
let rec fib_par n =
  let rec fib n =
    if n < 2 then 1
    else fib (n - 1) + fib (n - 2)
  in
  if n > 20 then begin
    let a = T.async pool (fun _ -> fib_par (n-1)) in
    let b = T.async pool (fun _ -> fib_par (n-2)) in
    T.await pool a + T.await pool b
  end else
    fib n

let main () =
  let sock = Eio.Net.listen ... in
  (* Runs once per request in an Eio task *)
  let request_handler n =
    T.run pool (fun _ -> fib_par n)
  in
  while true do
    (* spawn an Eio task to run [request_handler] per request *)
    Eio.Net.accept_fork sock ... request_handler ...
  done

let () = Eio_main.run main
```

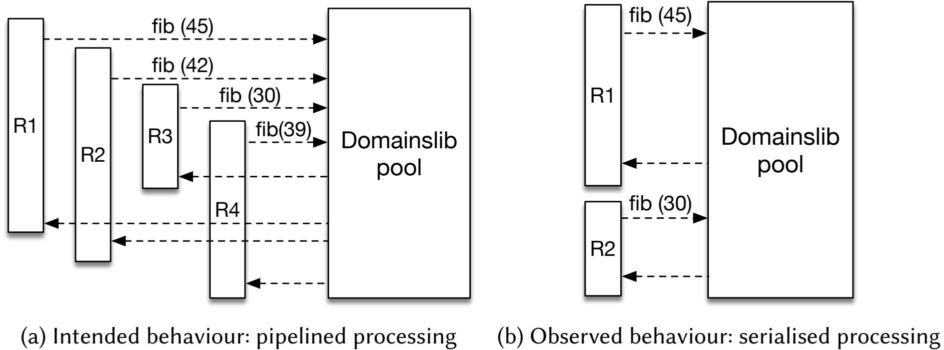
Fig. 1. Failed composition of Eio and Domainslib to implement a Fibonacci server.

libraries with effect-based ones. Section 6 discusses the challenges with lazy evaluation and concurrency and our solution. We evaluate the performance of our solution in Section 7. Finally, we discuss the related work in Section 8 and offer concluding discussion in Section 9.

## 2 MOTIVATION

In this section, we will describe a simplified example that illustrates the need to combine multiple concurrency libraries in the same application. Recall that OCaml 5 brings support for direct-style concurrency using effect handlers and shared-memory parallelism. Suppose the developer wants to implement a highly scalable web server that performs a compute-intensive but parallelisable computation for each request. For edification purposes, let us consider that for each request server gets a natural number input from the client and the server returns the nth Fibonacci number computed *recursively* back to the client. While the example itself is artificial, it captures a pattern OCaml users have encountered in practice.

The OCaml ecosystem provides appropriate libraries for implementing different parts of this application. The highly-scalable web server can be implemented with Eio. The nested parallel




(a) Intended behaviour: pipelined processing (b) Observed behaviour: serialised processing

Fig. 2. Composition of Eio and Domainslib in the Fibonacci server

computation may be parallelised with Domainslib. One may attempt to build the application as shown in Figure 1.

Let us walk through the code snippet. The unit of parallelism in OCaml is *domains*. Domains are heavy weight entities. Each domain maps directly to an OS thread and it is recommended that only as many domains as the number of available cores is created by the user. The library Domainslib permits better management of domains by creating a pool of domains, and submitting tasks to them. In our example, we create a pool of `num_domains` domains. This pool is shared between all of the requests that arrive at the server.

The function `fib_par` computes nth Fibonacci number in parallel using the sequential implementation for small inputs. The `main` function initialises the web server which listens on the socket `sock`. For each connection request, it creates an Eio task that runs the `request_handler` function. All of these Eio tasks are multiplexed on the same domain, but their execution overlaps with the execution of other concurrent requests. The `request_handler` uses `T.run` function (the same as `Domainslib.Task.run` function) to offload the expensive computation to Domainslib pool to perform the computation in parallel. The expected behaviour is shown in Figure 2a where the concurrent requests are processed in a pipelined fashion.

Unfortunately, the observed behaviour is that the requests are processed in a serialised fashion, one after the other as shown in Figure 2b. This is because the function `T.run` is a blocking function that blocks the entire calling domain and not just the Eio task that is making the call. Hence, none of the other tasks from the Eio scheduler can run until `T.run` returns. The fundamental problem is that Domainslib does not have a conception of Eio tasks, and cannot block and unblock them since their semantics is defined by the Eio scheduler. While we can define a point-wise synchronisation solution that works for the composition of Domainslib and Eio, such a pair-wise solution is unsatisfactory as it cannot accommodate other concurrency libraries. What we need is a generic way for tasks from different concurrency libraries to be suspended and resumed without appealing to the specific implementation details of a particular library.

## 3 EFFECTIVE COMPOSITION

In this section, we shall introduce our solution that enables applications such as the Fibonacci server to combine several concurrency libraries developed independently and have them work in the intended fashion. We shall first introduce a simple concurrency-safe scheduler that schedules threads in a FIFO fashion. We shall also use this example as a way to explain the semantics of

effect handlers in OCaml 5. We refer interested readers to the OCaml manual page on effect handlers [OCaml Effects 2023] for a more detailed explanation of their semantics.

### 3.1 A concurrency-safe FIFO scheduler

```
type task = Task : ('a,unit) continuation * 'a -> task

let run main =
  (* Lock-free queue *)
  let run_q = Queue.create () in
  (* Number of running tasks *)
  let nt = ref 0 in
  (* Mutex & Condition pair for parking the domain *)
  let m,c = Mutex.create (), Condition.create () in

  let enqueue k v = Queue.push (Task (k,v)) run_q in
  let rec dequeue () =
    match Queue.pop run_q with
    | Some (Task (k,v))-> continue k v (* resume the next task *)
    | None when !nt = 0 -> () (* No more threads. We're done. *)
    | _ -> (* Some task is blocked elsewhere *)
        Mutex.lock m;
        if Queue.is_empty run_q (* check again with lock *)
        then (Condition.wait c m; Mutex.unlock m; dequeue ())
        else (Mutex.unlock m; dequeue ())
  in

  let rec spawn f =
    incr nt;
    match_with f ()
    { retc = (fun () -> decr nt; dequeue ());
      exnc = begin fun exn ->
        decr nt;
        print_string (Printexc.to_string exn);
        dequeue ()
      end;
      effc = fun (type a) (e : a t) ->
        match e with
        | Yield -> Some (fun (k: (a,_) continuation) ->
            enqueue k (); dequeue ())
        | Fork f -> Some (fun (k: (a,_) continuation) ->
            enqueue k (); spawn f)
        | _ -> None
   }
  in
  spawn main
```

Fig. 3. Concurrency-safe FIFO scheduler using effect handlers

As mentioned in Section 1.2, the effects `Fork` and `Yield` have been declared but their implementation was not defined. The implementation of these effects are given by the effect handlers in each

of the concurrency libraries, which describe how to interpret Fork and Yield. A computation may perform the Fork and Yield effects without knowing about their implementations.

Figure 3 describes the core primitives of the concurrency-safe FIFO scheduler. The scheduler maintains a queue of tasks in a lock-free queue (line 5). A tasks is a pair of a delimited continuation and the value to resume the continuation with (line 1). We also maintain the number of live tasks in the integer counter nt (line 7). We use a mutex and condition variable pair to park the execution of the domain (line 9). It should be noted that mutex and condition from the OCaml standard library operate at the level of domains and not just on the tasks from a concurrency library. Hence, they block and unblock the entire domain and not just the current task from the scheduler.

The enqueue function (line 11) takes a delimited continuation and the value to resume the continuation with and pushes a task into the queue. The dequeue function is a bit more involved than the enqueue function. If the queue is not empty, we resume the next task from the scheduler (line 14). The continue primitive resumes a delimited continuation with the given value. Recall that our goal is to build synchronisation structures such as channels and MVars that can be utilised by tasks from different concurrency libraries. As a result, we may encounter the case where the tasks from this scheduler are blocked elsewhere and the queue is empty. In the case when the queue is empty and the nt counter is 0, we know that all the tasks created by this scheduler have run to completion. And hence, we are done (line 15). Otherwise, the queue is empty and the nt counter is not 0, which indicates that some task created by this scheduler is blocked elsewhere. Hence, we use the mutex and condition variable to park this domain. Care has to be taken to read the queue again with the lock to ensure proper synchronisation.

The spawn function (lines 23 to 41) implements the handler for the effects Fork and Yield. When a task is spawned, we increment the atomic counter nt, and evaluate the computation f in the context of the effect handler using the match_with function (line 25). The computation may return with a unit value (case retc on line 26), in which case, we decrement the nt counter and run the next tasks from the scheduler. If the task raises an exception (case exnc on line 27), then we decrement the nt counter, print the exception to standard output and resume with the next task. The case effc on line 32 describes how effects are handled. Observe that for each handled effect, we get a delimited continuation k that represents the suspended computation from the point of corresponding perform, delimited by the current handler. The handler for Yield enqueues the current task and resumes the next one from the scheduler. The handler for Fork suspends the current task and recursively calls spawn to run the given computation f as a new task.

### 3.2 The Suspend effect

How do we allow the tasks from the scheduler that we have defined to wait on tasks from other schedulers? For this, we need a common way to describe how to suspend and resume tasks. The solution is remarkably simple. We expect concurrency libraries implementing their own schedulers to implement a handler for the following effect.

```
1 type 'a resumer = 'a -> unit
2 type _ Effect.t += Suspend:('a resumer -> 'a option) -> 'a Effect.t
```

In order to suspend the current task, the computation performs the effect Suspend block, where the function block is applied to the resumer function that encapsulates the functionality to enqueue the task to the scheduler that it belongs to. In particular, the resumer closure will have the delimited continuation of the suspended task in its environment. The synchronisation structures such as channels and MVars can define the function block to block the current task. The function block is expected to return None when the task was successfully blocked. In the presence of multiple

domains, it may be the case that the original condition for which the task was about to be blocked already occurred. In which case, the task need not be blocked, and the function `block` should return `Some v`, where `v` of type `'a` is the value necessary to resume the task.

### 3.3 Promises

The use of `Suspend` effect is best understood by looking at how a synchronisation structure might utilise it. To that end, let us implement a concurrency-safe *promise* synchronisation structure that permits tasks from different schedulers to wait for a value. The promise interface is shown below:

```
module type Promise = sig
  type 'a t
  (** The type of promises *)
  val create : unit -> 'a t
  (** Create an unfilled promise *)
  exception Already_filled
  val fill : 'a t -> 'a -> unit
  (** Fill the promise with a value. Raises [Already_filled] exception
      if the promise is already filled. *)
  val await : 'a t -> 'a
  (** If the promise is filled, returns the value in the promise.
      Otherwise, blocks the calling task until the promise is filled
      and returns the filled value. *)
end
```

We can represent the state of the promise as:

```
type 'a state = Full of 'a | Empty of 'a resumer list
type 'a t = 'a state Atomic.t
```

A state of the promise is either full with a value or empty with a list of waiting tasks, represented by a list of resumers. The promise itself is a atomic reference to the state. The promise is created empty:

```
let create () = Atomic.make (Empty [])
```

The `fill` function:

```
exception Already_filled

let rec fill p v =
  let old = Atomic.get p in
  match old with
  | Full _ -> raise Already_filled
  | Empty l ->
      if Atomic.compare_and_set p old (Full v)
      then List.iter (fun r -> r v) l
      else fill p v
```

raises the `Already_filled` exception if the promise is already filled. Otherwise, it tries to update the atomic reference to the full state. If successful, then the blocked tasks are all resumed using the resumer. It may be the case that the tasks belong to different schedulers. All of the necessary information to resume the task in the right scheduler is encapsulated in the closure.

The `await` function is the most interesting one.

```
1 let await p =
2   let rec block r =
3     let old = Atomic.get p in
4     match old with
5     | Full v -> Some v
6     | Empty l ->
7       if Atomic.compare_and_set p old (Empty (r::l))
8       then None else block r
9   in
10  let old = Atomic.get p in
11  match old with
12  | Full v -> v
13  | _ -> perform (Suspend block)
```

If the promise is full, then the await function returns the filled value. Otherwise, the await function performs the Suspend block effect. The scheduler for the current task handles the Suspend block effect and applies block to the resumer r.

The block function reads the atomic location again. If the promise has since been filled, then block returns Some v to the scheduler. The scheduler immediately resumes the same task with the value v. Otherwise, block atomically tries to add the resumer r to the promise state. If successful, the block function returns None to the scheduler. At this point the scheduler switches to the next task from the scheduler queue or parks the domain if there are no other tasks. Otherwise, the block function is retried.

Observe that the promise implementation does not appeal to the specifics of any particular scheduler, but it allows tasks from different schedulers to interact using the promise, only blocking calling task from the corresponding scheduler. We call such implementations *scheduler agnostic*. We have implemented many common synchronisation structures such as channels, MVar, mutex, condition variables, etc., using a similar strategy. These implementations are compatible with any concurrency library that handles the Suspend effect.

### 3.4 Handling suspend effect

Let us now extend our scheduler from Section 3.1 in order to handle the Suspend effect.

```
1 | Suspend block -> Some (fun (k: (a,_) continuation) ->
2     let resumer v =
3       let wakeup = Queue.is_empty run_q in
4       enqueue k v;
5       if wakeup then begin
6         Mutex.lock m; Condition.signal c; Mutex.unlock m
7       end
8     in
9     match block resumer with
10    | None -> dequeue ()
11    | Some v -> continue k v)
```

The snippet above is added as one more case in our effect handler in Figure 3. The resumer function first checks whether the scheduler queue is empty. In this case, domain running the scheduler is parked. We must signal the condition c to wake up the domain. We then enqueue the continuation k to be resumed with the value v to the scheduler queue. Finally, if the domain running the scheduler needs to be woken up, then we signal the condition variable c. Note that if

multiple domain race to enqueue into the empty queue, at least one of the domains will see the queue to be empty and will wake up the parked domain.

The argument to the `Suspend` effect, `block`, is applied to this `resumer`. If `block` returns `None`, then the continuation `k` (captured in the `resumer`) is successfully blocked on the synchronisation structure, and the scheduler resumes the next thread from the scheduler. If `Block` returns `Some v`, then we immediately resume the current task with the value `v`.

The takeaway is that, by handling this one effect `Suspend` in the concurrency library, the concurrency library becomes compatible with the synchronisation structures such as the promise that we have defined earlier.

### 3.5 Fixing the Fibonacci server

We can use our promise implementation to fix the erroneous behaviour in our Fibonacci server from Section 2. Recall that the problem in Figure 1 was that the call to `T.run` in the `request_handler` function blocks the entire domain and not just the current task. In order to get the pipelined behaviour as shown in Figure 2a, we replace the `request_handler` with the one that follows:

```
1 let request_handler n =
2   let p = Promise.create () in
3   ignore (T.async pool (fun _ -> Promise.fill p (fib_par n)));
4   Promise.await p
```

Here, we create a promise `p` per request. The function to compute the nth Fibonacci number is sent to the Domainslib pool using the `T.async` function. Importantly, `T.async` does not block the caller. The promise `p` is filled with the result of the execution of `fib_par n` function. The `request_handler` awaits on the promise `p`, which only blocks the current Eio task. Hence, other Eio tasks are free to run and we get the pipelined behaviour.

## 4 CANCELLATION

Languages that support user-level concurrency make it easy to create millions of tasks with ease. Unlike concurrency through heavy-weight OS threads, lightweight tasks are also cancelled frequently. For example, in a parallel depth-first search in a graph, a large number of search tasks may be created. Once the element that we are looking for is found, all the other search threads will need to be cancelled. Similarly, concurrency libraries such as Eio that allow high-performance I/O prefer structured concurrency [JEP428 2023] where the tasks are arranged in a tree-structured hierarchy and cancelling a task ensures that all the tasks as well as their resources are cleaned up. The cancellation mechanisms in different concurrency libraries are bespoke, diverse and are expected to be efficient.

### 4.1 Cancellation challenges

Given that tasks that are blocked on synchronisation structures may be cancelled, cancellation needs coordination between the concurrency libraries and the synchronisation structures. Without taking cancellation into account, we will have observe unintended behaviours. To illustrate this, consider that we extend our scheduler from Section 3 with the ability to cancel tasks with the following API:

```
1 type handle
2 val fork : (unit -> unit) -> handle
3 val cancel : handle -> unit
```

We introduce a type of task handlers `handle`. The `fork` function returns a `handle` rather than `unit`. The function `cancel` marks the task represented by the `handle` to be cancelled. The task may either be currently running, ready to run in the scheduler queue or be blocked on some synchronisation structure. If the task is currently not running, then a cancelled task is guaranteed not to run.

We can implement the functionality be extending the scheduler from Figure 3 as follows. We only highlight the important changes here and the full code for the scheduler that supports cancellation is found in the supplementary material.

```
1 type handle = {mutable cancelled : bool}
2 let cancel task = task.cancelled <- true
3 type _ Effect.t += Fork : (unit -> unit) -> handle Effect.t
4 type task = Task: handle * ('a,unit) continuation * 'a -> task
```

The `handle` is represented with a boolean mutable field in a record. `cancel` just sets this field to `true`. The `Fork` effect now returns the `handle` to the newly created task and the suspended task now carries its handle. We use this `handle` to decide whether to resume a suspended task:

```
1 let rec dequeue () =
2   match Queue.pop run_q with
3   | Some (Task (handle, k, v)) -> (* resume the next task *)
4       if handle.cancelled then discontinue k Exit else continue k v
5   ...
```

Instead of unconditionally resuming the task, we now examine whether the `handle` has been cancelled. If so, we resume the continuation by raising the `Exit` exception using the `discontinue` primitive. Discontinuing the continuation on cancellation is essential to ensure that the task stack is unwound freeing any resources such as open file descriptors. If the `handle` is not cancelled, then we resume the continuation `k` with the value `v` as before.

Suppose we use this scheduler with a scheduler-agnostic task-level mutex library `TaskMutex`. Similar to the standard library `Mutex` module, `TaskMutex` allows creation, lock and unlock of the mutex, except that the latter blocks only the calling task and not the calling domain. Without paying attention to cancellation, the combination of a scheduler that supports cancellation with the `TaskMutex` breaks. For example, consider the following code:

```
1 module M = TaskMutex;;
2
3 run (fun () -> (* main task *)
4   let m = M.create () in
5   let lu () = M.lock (); M.unlock () in
6   M.lock m;
7   let t1 = fork lu in (* control switches to t1 *)
8   cancel t1;
9   let t2 = fork lu in (* [t2] waiting behind [t1] to lock the mutex *)
10  M.unlock m; (* lock gets transferred to [t1] which was cancelled *)
11  (* [t2] does not get the mutex, and [run] gets stuck *)
12 )
```

Here, we create a mutex `m` to synchronise access between concurrent tasks. The function `lu` simply locks and unlocks the mutex `m`. The main task locks the mutex, and spawns a task `t1` which calls `lu`. By looking at the handler for `Fork` effect 3, one can see that the `fork` call suspends the current task and switches control to `t1`, which tries to lock the mutex. Given that the mutex is currently held by the main task, `t1` blocks on the mutex and the control switches back to the main

task. The main task now cancels t1 marking its handle as cancelled. Now, the main task creates t2, which also runs lu. Given that the mutex is still held by the main task, t2 blocks behind t1 waiting to lock the mutex.

Finally, the main task unlocks the mutex and runs to completion. The problem occurs here. Without the knowledge that t1 has been cancelled, the unlock call transfers the lock over m over to t1 and enqueues t1 to the scheduler queue. However, t1 is immediately terminated by **discontinue** when the dequeue function examines it at the end of the execution of the main task. Since the mutex is never unlocked by t1, t2 remains blocked forever. Recall that the run function will only return when all the tasks run to completion. Hence, the call to run function never returns and the execution deadlocks.

One may argue that the problem here is that the call to lock should be aware of cancellation and should be protected by an exception handler that handles the Exit exception. But observe that the Exit exception comes from the scheduler that was independently developed from the TaskMutex module. It is not immediately apparent whether documenting that blocking functions may through exceptions is the appropriate approach. Either way, the libraries should be built to avoid deadlocks even in the presence of buggy code.

## 4.2 Cancellation awareness

We fix this issue by modifying the signature of the Suspend effect.

```
1 type 'a resumer = 'a -> bool (* instead of [unit] *)
2 type _ Effect.t += Suspend: ('a resumer -> 'a option) -> 'a Effect.t
```

The only change that we introduce is to make the resumer return bool instead of unit. The resumer is expected to return true if the task was not cancelled and successfully resumed. Otherwise, it returns false.

As before, the use of this interface is split between the concurrency library and the synchronisation structure. In the concurrency library, we modify the Suspend handler as follows:

```
1  | Suspend block -> Some (fun (k: (a,_) continuation) ->
2      let resumer v =
3        let wakeup = Queue.is_empty run_q in
4        enqueue k v;
5        if wakeup then begin
6          Mutex.lock m; Condition.signal c; Mutex.unlock m
7        end;
8        not handle.cancelled
9      in
10     match block resumer with
11     | None -> dequeue ()
12     | Some v -> if handle.cancelled then continue k v
13                 else discontinue k Exit )
```

The handler for the Suspend effect here is the cancellation-aware version of the Suspend handler in Section 3.4. There are two changes. First, the resume function returns true when the task is not cancelled, and false otherwise. It might seem strange that we enqueue the continuation k to be resumed with the value v. But recall that the dequeue function first checks whether the handle was cancelled, and if so, discontinues the continuation k with the Exit exception. The second change is that we also check whether the task has been cancelled in the case when block resumer returns

```
module Mutex : Mutex = struct
  type state = Unlocked | Locked of unit resumer list
  type t = state Atomic.t

  let create () = Atomic.make Unlocked

  let lock m =
    let rec block r =
      let old = Atomic.get m in
      match old with
      | Unlocked -> if Atomic.compare_and_set m old (Locked [])
          then (Some ()) else block r (* failed CAS; retry *)

      | Locked l -> if Atomic.compare_and_set m old (Locked (r::l))
          then None else block r (* failed CAS; retry *)
    in
    perform (Suspend block)

  let rec unlock m =
    let old = Atomic.get m in
    match old with
    | Unlocked -> failwith "impossible"
    | Locked [] -> if Atomic.compare_and_set m old Unlocked
        then () else unlock m (* failed CAS; retry *)
    | Locked (r::rs) -> if Atomic.compare_and_set m old (Locked rs)
        then begin
          if r () then () (* successfully transferred control *)
          else unlock m (* cancelled; wake up next task *)
        end else unlock m (* failed CAS; retry *)
end
```

Fig. 4. Task-level mutex implementation that is aware of cancellation.

`Some v`. Before resuming the continuation `k` with `v`, we confirm that the task has not been cancelled. If cancelled, we immediately **`discontinue`** the continuation.

Figure 4 shows the `TaskMutex` implementation that has been made aware of cancellation. The implementation follows ideas similar to the promise implementation from Section 3.3. The only change necessary to make the implementation aware of cancellation is in the lines 26 and 27. When we unlock the mutex, we check whether there are pending tasks waiting to lock the mutex. If so, we try to resume them by invoking the resumer `r`. If the resumer `r` returns `true`, then the task associated with this resumer is not cancelled and successfully resumed. Otherwise, if `r` returns `false`, then the task was cancelled and we retry `unlock` to wake up other blocked tasks.

### 4.3 Eager and lazy cancellation

The cancellation semantics that we have prototyped here may be termed as *lazy* cancellation. When a task is cancelled, we simply mark its handle as cancelled and we wait until it is the tasks turn to run in the scheduler in order to terminate it with the **`discontinue`** primitive. In particular, if the cancelled task was blocked on a synchronisation structure, we require that a matching operation is done on the synchronisation structure that unblocks the task and pushes it into the scheduler

queue. An alternative would be *eager* cancellation, where, at the point of cancellation, if the task were blocked on a synchronisation structure, it is removed from the structure eagerly and pushed into the scheduler queue. This is orthogonal to the concerns in this paper and is a concern of the concurrency library and the synchronisation structure. The **Suspend** effect only expects the `resumer` to return `false` irrespective of whether the task cancellation was eager or lazy.

## 5 COMPOSING MONADIC LIBRARIES

So far we have focussed on the composition of concurrency libraries written using effect handlers such as Eio and Domainslib. However, given that effect handlers is a fairly recent addition to OCaml, most of the concurrent code in OCaml today are written in monadic concurrency libraries such as Lwt and Async. These library ecosystems are fairly mature and millions of lines of monadic concurrency code are used everyday. It is likely that monadic concurrency will survive many years into the future.

Given the impossibility of a whole-sale migration of the code written in monadic concurrency to direct-style effect based concurrency, there is the need to ensure that the monadic code can be incrementally migrate to effect-based concurrency. Even if the aim is not to migrate code, legacy applications may want to take advantage of newer libraries. For example, an Lwt application may way to offload compute-intensive computation to a Domainslib pool to take advantage of parallelism. Similarly, an Async application may want to utilise Eio to take advantage of newer OS features for efficient IO such as `io_uring` [io_uring 2023]. While solutions such as `Lwt_domain` [Lwt_domain 2023] and Eio bridges to Lwt and Async exist, such point-wise solutions are unsatisfactory. For example, Eio-Lwt bridge cannot take advantage of parallelism since Lwt is not parallelism-safe. Ideally, we would like to run Lwt on one domain and Eio on multiple domains to get the best performance. In this section, we show that our solution enables monadic concurrency libraries such as Lwt and Async to be composed with effect-based concurrency libraries.

### 5.1 Monadic API for synchronisation structures

There are several challenges to enable such a composition. The notion of a continuation is different between direct-style concurrency libraries based on effect handlers and monadic concurrency libraries. With effect handlers, the continuation is represented by a segment of the call stack [Sivaramakrishnan et al. 2021] managed by the runtime whereas Lwt and Async essentially utilise callback functions as continuations. This leads to the situation where we will have two orthogonal continuations in the program. For example, consider that the Lwt program uses the promise from Section 3.3. The program state in this case is shown in the Figure 5. Since the promise API is in direct-style, the call to the `await` function may include several intermediate function calls `f0`, `f1` and `f2` from the Lwt scheduler. The continuation captured by performing the **Suspend** effect is the *vertical* one that includes the relevant segment of the stack segment of the stack. On the other hand, the continuation in Lwt is the callback function `g1 >>= g2 >>= ...`. It is unclear how to capture and reconcile both of these continuations.

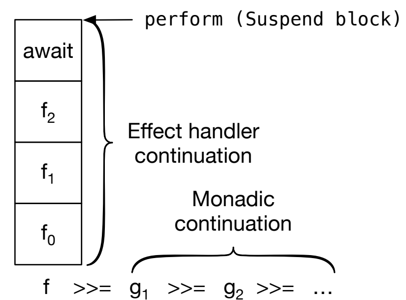



Fig. 5. Two continuations in a monadic concurrency library.

Instead, we obviate the need to capture the vertical stack by wrapping the API of the synchronisation structures in a monadic interface as follows:

```
module Lwt_promise : sig
  type 'a t
  val create : unit -> 'a t
  val fill : 'a t -> 'a -> unit
  val await : 'a t -> 'a Lwt.t
end = struct
  type 'a t = 'a Promise.t
  let create = Promise.create
  let fill = Promise.fill
  let await p = Lwt.return (Promise.await p) (* WIP *)
end
```

The only way to use `Lwt_promise.await` is to bind it with the rest of the Lwt computation using >>=. Hence, `Lwt_promise.await` cannot appear in the nested position as in Figure 5. Now, we only need to capture the Lwt continuation. Note that `fill` does not block and hence does not need the Lwt wrapper.

### 5.2 Integrating Suspend effect with Lwt

The `Lwt_promise.await` that we have defined still needs some work. Recall that `await` performs the `Suspend block` effect which needs to be handled. Since we have introduced a monadic wrapper around `await`, the delimited continuation `k` from the effect handler is no longer necessary. But something has to be done to intercept the `Suspend block` effect and apply the `block` function to a suitably prepared `resumer` for Lwt. For this, we exploit the fact that in OCaml, if there are no handlers for an effect `e`, then the exception `Unhandled e` is raised at the point of `perform`. Our final `await` function is as follows.

```
let suspend_monad block =
  let promise, resolver = Lwt.wait () in
  let resumer v = Lwt.wakeup resolver v; true in
  match block resumer with
  | Some v -> Lwt.return v
  | None   -> promise

let await p =
  try Lwt.return (Promise.await p) with
  | Unhandled (Suspend block) -> suspend_monad block
```

The `Lwt_promise.await` function handles the `Unhandled (Suspend block)` effect and applies the `suspend_monad` function to the `block` function. The `suspend_monad` function performs the task similar to the handler for the `Suspend` effect in effect handler based scheduler from Section 3.4. It uses `Lwt.wait` to get a pair of an Lwt's own internal `promise` and a `resolver`. The `resolver` is the other end of the Lwt `promise`. When the resumer is invoked, the promise is filled using `Lwt.wakeup` on the `resolver` with value `v`, which enables the Lwt task to continue. As in the effect handler based scheduler, the `block` function is applied to the `resumer` with the results appropriately handled. For simplicity, the `resumer` function shown here does not handle task cancellation.

Using this solution, we have implemented Lwt-based versions of synchronisation structures that enables Lwt to seamlessly interact with effect handler based concurrency libraries such as Eio.

## 6 CONCURRENCY-SAFE LAZY VALUES

So far we have discussed a number of scheduler-agnostic synchronisation structures such as promises and task-level mutexes. These synchronisation structures were implemented as libraries. It turns out that the OCaml language has a primitive feature that can take advantage of the `Suspend` effect to be parametric over the concurrency library. This feature is *lazy values*.

OCaml has built-in support for deferred computations through lazy values. The special syntax `lazy (expr)` returns a lazy value for computing `expr`. Forcing this lazy value computes `expr` and returns its result. The compiler performs certain optimisations that keeps the cost of accessing the result of the already forced lazy value to a minimum. Lazy values are especially useful to model the case where there are many expensive computations but only a few of whose results will be needed, potentially many times.

In OCaml, forcing a lazy value is not concurrency-safe. When a lazy value is concurrently forced, its behaviour is unspecified but OCaml guarantees that there will be no crashes and that the lazy computation is only every forced by one of the callers. The recommendation is that any use of lazy should be protected by a mutex. However, the downside of this solution is that even when the lazy computation has been computed, the program will still have to pay for the cost of locking and unlocking the mutex around the lazy value.

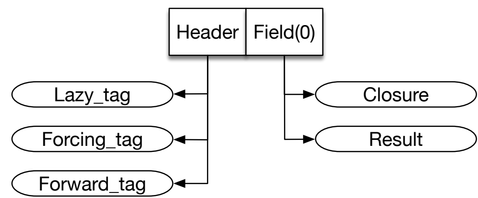



Fig. 6. Lazy object layout: current

What we need is concurrency-safe lazy that can be accessed concurrently without having to resort to the use of a mutex. The idea here is similar to blackholing of thunks in the GHC runtime [Harris et al. 2005; Marlow et al. 2009]. Whenever multiple Haskell (lightweight) threads race to evaluate a thunk, the threads that lose the race are blocked on the thunk. When the thunk evaluation completes, the blocked threads are resumed with the result of the thunk evaluation. The difference between GHC and OCaml is that, unlike GHC, OCaml does not have a built-in thread scheduler in the runtime system. Instead, we use the `Suspend` effect to make the lazy implementation parametric over the scheduler.

### 6.1 Lazy objects in OCaml today

In order to enable lazy values to take advantage of `Suspend` effect, we need to modify the layout of lazy values in OCaml. Let us first look at the implementation of lazy values in OCaml today. Figure 6 shows the layout of lazy objects in OCaml 5. A lazy value has an object header and one field, each one word in size. Initially, the lazy value has take `Lazy_tag` and the value of the first field is the closure representing the deferred computation. When the lazy value is forced, the tag is first atomically updated to `Forcing_tag`. When the lazy computation either recursively forces itself or another domain concurrency forces the lazy, it will find the tag to be `Forcing_tag`. In this case, OCaml raises the `Lazy.Undefined` exception. When the computation successfully completes execution, the tag is updated to `Forward_tag` and the first field of the object now points to the result. On the other hand, if the lazy computation raises an exception `exn`, then the first field is updated to a thunk that raises

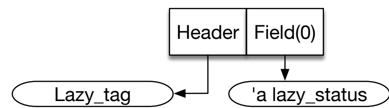



Fig. 7. Lazy object layout: concurrency-safe

exn and the tag is reset from Forcing_tag to Lazy_tag. This causes subsequent forcing of this lazy value to immediately raise the exception exn.

The OCaml garbage collector (GC) performs short-cutting optimisation on lazy value. If the GC observes an object with the Forward_tag, it makes the reference to this object directly point to the result. Short-cutting avoids one hop when the program access the result of the already evaluated lazy value. When all the reference to the Forward_tag object have been short-circuited, the object itself is GCed.

### 6.2 Suspending on lazy

We modify the layout of the lazy object as shown in Figure 7 in order to accommodate concurrency. We only use the Lazy_tag now and not the other tags. The first field now stores a value of type 'a lazy_state defined below:

```
type 'a lazy_state =
| Unforced of (unit -> 'a)
| Forcing of int (* unique id *) * 'a resumer list
| Forwarded of 'a
```

The lazy value initially stores Unforced comp where the comp is the deferred computation. When the lazy is forced, the state is atomically updated to Forcing (id,[]). The unique id is utilised to distinguish between recursive forcing of the lazy value by comp and concurrent forcing of the same lazy by a different task. Recursive forcing is an error and indicates non-termination. This case is similar to GHC Haskell's recursive evaluation of a thunk, where GHC raises the NonTermination exception at runtime. In this case, we raise the Lazy.Undefined exception. Note that this unique id should be unique not just among the tasks from the current scheduler, but also unique between tasks from different schedulers. There are several solutions to this problem, but the problem itself is orthogonal to the focus of this work. Hence, we do not elaborate on this further.

When the lazy is concurrently forced, we use the Suspend effect to capture the resumer and atomically add it to the resumer list as in the case of promises. When the computation finishes execution with the result v, the state is atomically updated to Forwarded v and any tasks that were blocked on this lazy are resumed with the help of the resumer. If the computation throws an exception exn, the state of the lazy is updated to Unforced (fun () -> raise exn). At this point, we will need to resume the blocked tasks with the exn. However, observe that the type of the resumer is a 'a -> bool function (Section 4.2) and can only be resumed with a value.

### 6.3 Resuming with an exception

We modify the signature of the Suspend effect to be

```
type 'a resumer = ('a,exn) Result.t (* instead of ['a] *) -> bool
type _ Effect.t += Suspend: ('a resumer -> 'a option) -> 'a Effect.t
```

where the Result.t type defined in the OCaml standard library is:

```
(* module Result *)
type ('a,'e) t = Ok of 'a | Error of 'e
```

This is our final type of Suspend effect that we use in our development. The expectation on the resumers is that if the argument is Ok v then the task is resumed with the value v. If the argument is Error exn, then the task is set up such that it continues with the exception exn. Observe that the resumption must be handled in a concurrency-library-specific way. For effect handler based concurrency libraries, we can use the continue and discontinue primitives for these two cases,

respectively. In the case of Lwt, the resumer will use `Lwt.wakeup` and `Lwt.wakeup_exn`, respectively. Note that `Lwt.wakeup_exn exn` resolves a promise with the exception `exn`.

### 6.4 Advantages of the new lazy design

Our lazy implementation has a number of nice advantages. The OCaml GC can still short-circuit the lazy values by examining whether the value in the first field was constructed with the `Forwarded` constructor. Unlike the existing OCaml 5 design, tags in the object header are no longer modified. This eliminates the need for the concurrent GC thread marking the lazy object and the OCaml program modifying the lazy object header [Sivaramakrishnan et al. 2020].

Thanks to the use of `Suspend` effect, any concurrency library that handles the `Suspend` effect can safely share the lazy values. Hence, lazy values can be used by multiple tasks from Eio, Domainslib and Lwt. But what about sharing lazy values between multiple domains (OS threads that run in parallel) and systhreads (OS threads that time-share a domain) [OCaml threads library 2023] directly when there is no user-level scheduler hosted on them? In this case, we will not have a handler for the `Suspend` effect. We handle this case similar to the case of handling the `Unhandled (Suspend block)` in Lwt (Section 5.2) and parking the domain or the systhread on a condition variable. As a result, **our lazy values can be used concurrently by all concurrency abstractions in OCaml** – domains, systhreads, effect handler based concurrency libraries such as Eio and Domainslib and monadic concurrency libraries such as Lwt.

## 7 EVALUATION

In this work, we propose to compose concurrency libraries through the single `Suspend` effect. This allows concurrency libraries and synchronisation structures to be implemented independently. In this section, our goal is to show that this approach is pragmatic and does not incur additional overheads compared to implementing bespoke concurrency libraries with their own synchronisation structures.

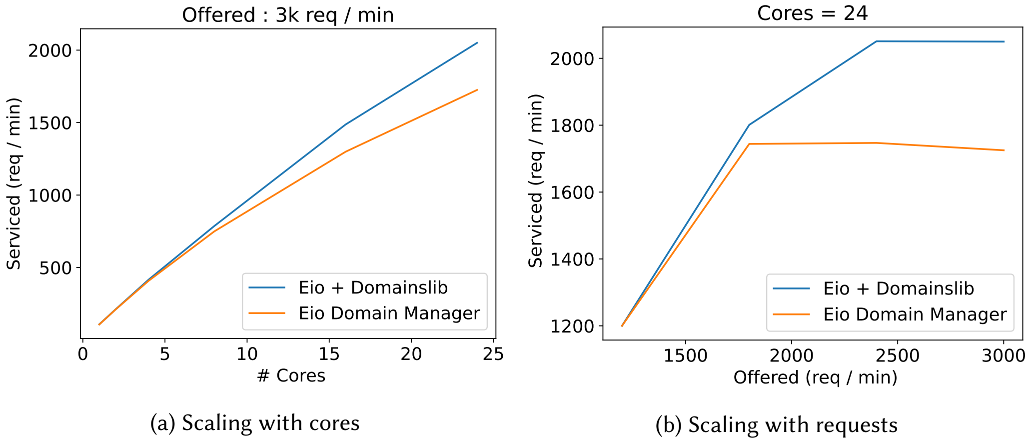
We have implemented scheduler-agnostic MVars [Peyton Jones et al. 1996], a general-purpose and expressive synchronisation structure. MVar is a blocking bounded queue with a bound of one. Taking a value from an empty MVar and putting a value into a full MVar blocks the caller. In our experiments, we use these MVars to compose together different libraries. Our experiments are run on a Intel(R) Xeon(R) 5120 CPU x86-64 server with 2 sockets and 28 physical cores. It has 14 cores on each socket and 2 hardware threads per core. Each core runs at a clock speed of 2.20 GHz. The server has 64 GB of main memory. It runs on Ubuntu 20.04. We use OCaml compiler version 5.0.0, which supports effect handlers and shared memory parallelism.

### 7.1 Producer-consumer benchmark

How does the scheduler-agnostic MVar fare against synchronisation bespoke structures on communication-intensive workloads? In order to answer this, we implemented a single-producer, single-consumer benchmark in Eio (version 0.6) where the Eio tasks exchange messages. Eio provides streams as an efficient way to synchronise between multiple tasks. Streams are blocking bounded queues that can only be used between Eio tasks. We use the bound of 1 in order to match the behaviour of an MVar. Both streams and MVars are concurrency-safe and can be used across domains. We run two experiments where the producer and consumer tasks reside on the same domain (serial) and on different domains (parallel).

Table 1. Time (in $\mu$s) for sending a single message between tasks.

| Structure | Serial | Parallel |
|---|---|---|
| Eio stream | 1.63 | 4.39 |
| MVar | 1.51 | 4.15 |




(a) Scaling with cores

(b) Scaling with requests

Fig. 8. Throughput of Fibonacci server.

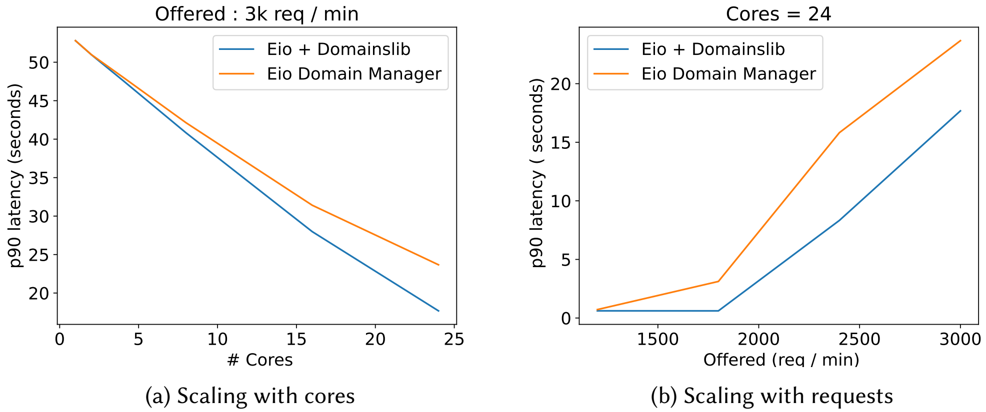
Table 1 reports the time in microseconds ($\mu s$) to send a single message between the producer and the consumer. Sending a message between domains is more expensive due to the potential cost parking and unparking domains and failed `compare_and_set` operations. The results show that our scheduler-agnostic MVar performs on par with Eio streams.

### 7.2 Fibonacci Server

We measure the performance of the Fibonacci server described in Section 2. Our server is a full-fledged HTTP server that can handle a large number of concurrent connections and requests. We compare two variants here: (1) **Eio + Domainslib** described in Section 3.5 but uses MVars instead of promises and (2) **Eio Domain Manager** that uses the Eio's built-in support for domains. Note that unlike Eio + Domainslib, the Eio Domain Manager variant does not parallelise a single request to compute the Fibonacci number, but exploits the fact that multiple requests are independent and hence can be run in parallel. Eio Domain Manager variant uses Eio streams for communicating between Eio tasks that may potentially run across different domains. In both cases, the client workload is generated using *wrk2* [Wrk2 2020], a high-performance workload generator for testing the performance of HTTP servers. For simplicity, every request computes the 45th Fibonacci number. While our server can accept different inputs, performing the same work in each request allows us to better interpret the experimental results.

*7.2.1 Throughput.* We perform two experiments and report the results. First we compare the throughput of the different variants. In the first experiment, we maintain a constant load of 3000 requests per minute and vary the number of cores. We measure the throughput in terms of the requests serviced per minute. The results are presented in Figure 8a. In the second experiment, we maintain the core count to be a constant 24 and increase the number of offered requests and measure the serviced request rate. The results are presented in Figure 8b. We can see that Eio + Domainslib variant scales better in both experiments due to better parallelisation of available requests with increasing number of core.

*7.2.2 Latency.* We also measure the 90th percentile (p90) latency on the two experiments. As we increase the number of cores (Figure 9a), the p90 latency comes down as the requests can be parallelised across the available cores. We see that the latency comes down faster with Eio + Domainslib compared to the Eio Domain Manager. This is because with Eio + Domainslib, each of the requests can itself be parallelised and hence the p90 latency for each request comes down faster.




(a) Scaling with cores

(b) Scaling with requests

Fig. 9. Latency of the Fibonacci server.

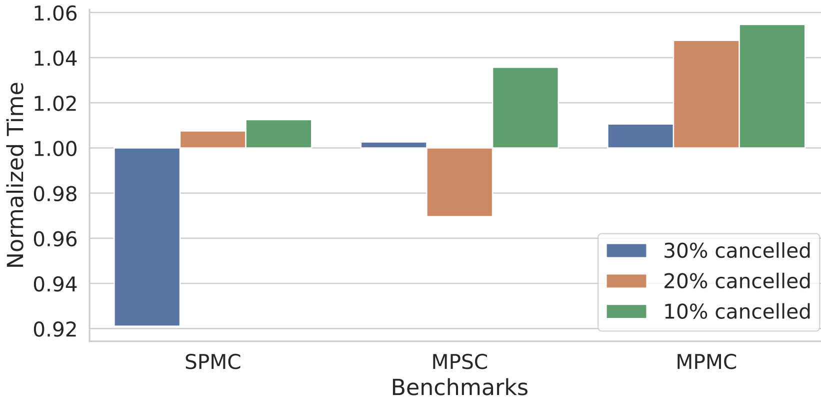



Fig. 10. Normalized time for Cancellation cost benchmark. Baseline is application without any cancelled tasks

In the second experiment, we fix the number of cores and increase the offered request rate. With increasing number of requests, we expect each of the requests to take longer to complete. Hence, we expect the p90 latency to go up. The results in Figure 9b shows that the increase is slower in Eio + Domainslib compared to Eio Domain Manager. Thus, in all of our experiments Eio + Domainslib performs better than Eio Domain Manager variant.

### 7.3 Cancellation cost

In this section, we measure the cancellation cost using the scheduler that we built in Section 4. We set up the experiment as follows. The benchmark is a producer-consumer benchmark where the producers share 50k items with the consumer. All the producers are run on a domain and all the consumers are run on a different domain. We have three variants of the benchmark: (1) 1 producer and 10k consumers (SPMC), (2) 10k producers and 1 consumer (MPSC) and (3) 5k producers and 5k consumers (MPMC). In each of the variants, we cancel a fixed percentage (10%, 20% and 30%) of tasks. Despite the cancellation, the experiment is set up such that the producers will all together still send 50k items to the consumer. Hence, the total amount of work done by the benchmark is

still the same despite cancellation. Given the work done remains the same, the expectation is that, if the cancellation were zero cost, then the total running time variant with and without cancellation will be the same.

The results are presented in Figure 10. The graph presents the normalised running time for each of the variants where the baseline is no cancellation. If the cancellation cost were zero, we would expect not to see any bars. The graphs show that the maximum slowdown with cancellation is 5%. But we also see that with cancellations, the programs tend to be faster, and sometimes even faster than the baseline with no cancellations. This is because with cancellation we will have fewer tasks, which can get through the fixed amount of work faster thanks to lower task switching overheads. Overall, the graph shows that cancellation is efficient.

## 8 RELATED WORK

Algebraic effect handlers have been an active discipline of theoretical research [Plotkin and Pretnar 2009; Plotkin and Power 2001]. Many research programming languages and libraries implemented effect handlers. Notable ones are Koka [Leijen 2017a] and Eff [Bauer and Pretnar 2015] which pioneered the scalable programming with effect handlers. OCaml 5 is the first-industrial strength language that offers effect handlers. OCaml uses effect handlers as the primary means of achieving concurrency in direct-style [Dolan et al. 2018; Sivaramakrishnan et al. 2021].

Unlike Koka or Eff, OCaml does not offer *effect safety* – no static guarantee that all the effects performed are handled in the program. Effect safety is an active area of research [Biernacki et al. 2019a,b; Hillerstrom et al. 2020; Leijen 2017b]. In OCaml, when there are no handlers for an effect, the `Unhandled` exception is raised at the point of perform. Interestingly, we utilise this behaviour to compose monadic concurrency libraries with direct-style ones.

OCaml 5 introduces language-level support for programming with delimited continuations through effect handlers. Libraries like Eio and Domainslib offer direct-style concurrency using effect handlers as opposed to monadic concurrency libraries such as Lwt and Async. Implementing concurrency libraries over first-class continuations, delimited or otherwise, is a well-studied problem. Several languages in the Lisp family and Standard ML compilers, like SML/NJ and MLton provide support for first-class continuations on top of which concurrency libraries such as Concurrent ML (CML) [Reppy 1999] are implemented. CML implements a preemptive FIFO scheduler, provides unbounded blocking channels and mailboxes (MVars) and implements an expressive framework for building communication protocol over events. The novelty in this work is that, unlike the idea of "one concurrency library to rule them all", the `Suspend` effect permits composition of different concurrency libraries.

Many modern languages provide support for built-in lightweight concurrency such as the Go language and GHC Haskell. Implementing the concurrency support entirely in the runtime system makes the runtime system bloat and monolithic. While Go language is excellent for IO intensive programs, nested parallelism can neither be expressed naturally using goroutines nor is the Go scheduler optimised for nested parallel programming. Sivaramakrishnan et al. [KC et al. 2016] attempted to relieve GHC Haskell of this problem by implementing support for continuations in GHC and reimplementing the threading subsystem of GHC Haskell completely in Haskell. Similar to OCaml users may implement their own schedulers for tasks with the help of *scheduler activations*. The threads in GHC Haskell interact non-trivially with other parts of the runtime system including the software-transactional memory, foreign calls, IO manager, lazy evaluation and timers. The runtime system performs *upcalls* to the scheduler in order to service these requests similar to the way the synchronisation structures perform `Suspend` effect interact with the scheduler, but in the opposite direction. However, compared to our work, Sivaramakrishnan et al. associate a single scheduler with every Haskell execution context (HECs), which are equivalent to domains in OCaml.

This prevents richer scheduling policies such as hierarchical scheduling and structured concurrency libraries such as Eio.

Adding support for asynchronous IO for highly scalable concurrent applications to a programming language often creates a split between synchronous and asynchronous code [Function Colour 2023]. The Rust programming language is no exception. We cannot directly invoke async functions from sync functions in Rust. We cannot therefore arbitrarily combine sync and asynchronous code. The execution of async code requires an async runtime. Rust does not have a built-in async runtime, unlike Go or GHC Haskell. It has a number of community-maintained crates such as Tokio [Tokio 2022], Async-std [Async-std 2023], Rayon [Rayon 2022], Smol, that offer various async runtimes for concurrency. Similar to OCaml, Rust faces an interoperability problem across different runtimes due to the presence of multiple async runtimes [Async Rust Book 2023]. Two async runtimes cannot be freely combined. Nonetheless, efforts have been made to establish compatibility layers between Tokio and other runtimes [Async Rust Book 2023]. It is possible to merge the Tokio and Rayon libraries via a bespoke one-shot channel [Rayon Tokio Crate 2021]. However, these are not universal solutions for composing runtimes.

Stephen et al. [Muller et al. 2017] developed a language and graph-based cost model to combine competitive and cooperative threading models. The idea here is to implement a scheduling policy that can handle both competitive and cooperative threads in the same scheduler that guarantees that the theoretical cost bounds developed in the paper are respected by the implementation. Unlike this, our goal is to allow composition of different schedulers developed independently. Each scheduler maintains its own tasks and the tasks from different concurrency libraries only interact through the synchronisation structures.

## 9 DISCUSSION

One of OCaml community's strengths is that there are a variety of libraries for each problem. Hence, it is anti-thetical that for concurrent programming, the OCaml programmer had to make a choice between the mutually incompatible Lwt or Async ecosystems. While the arrival of OCaml 5 features, we have another cambrian explosion of concurrency libraries implemented using effect handlers. It is important that the community finds ways for the libraries to coexist so as to prevent futher split in the concurrent programming ecosystem. We believe that our unified interface for expressing concurrency using the `Suspend` effect:

```
1 type 'a resumer = ('a,exn) Result.t -> bool
2 type _ Effect.t += Suspend: ('a resumer -> 'a option) -> 'a Effect.t
```

offers a simple, effective and performant solution for concurrency library composition. We plan to propose the `Suspend` effect to be included in the OCaml standard library so that different libraries may agree on this effect and develop their composable solutions against this interface. In this work, we also show how to fix the tricky problem of making lazy values compatible with disparate forms of concurrency in the OCaml language using the above interface.

## REFERENCES

Async 2023. *Typeful concurrent programming.* https://opensource.janestreet.com/async/

Async Rust Book 2023. *Asynchronous programming in Rust.* https://rust-lang.github.io/async-book/01_getting_started/03_state_of_async_rust.html#compatibility-considerations

Async-std 2023. *Async version of the Rust standard library.* https://docs.rs/async-std/latest/async_std/

Andrej Bauer and Matija Pretnar. 2015. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming* 84, 1 (2015), 108–123. https://doi.org/10.1016/j.jlamp.2014.02.001 Special Issue: The 23rd Nordic Workshop on Programming Theory (NWPT 2011) Special Issue: Domains X, International workshop on Domain Theory and applications, Swansea, 5-7 September, 2011.

Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2019a. Abstracting Algebraic Effects. *Proc. ACM Program. Lang.* 3, POPL, Article 6 (jan 2019), 28 pages. https://doi.org/10.1145/3290319

Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2019b. Binders by Day, Labels by Night: Effect Instances via Lexically Scoped Handlers. *Proc. ACM Program. Lang.* 4, POPL, Article 48 (dec 2019), 29 pages. https://doi.org/10.1145/3371116

C# async/await 2023. *Asynchronous programming with async and await.* https://learn.microsoft.com/en-us/dotnet/csharp/asynchronous-programming/

Stephen Dolan, Spiros Eliopoulos, Daniel Hillerström, Anil Madhavapeddy, K. C. Sivaramakrishnan, and Leo White. 2018. Concurrent System Programming with Effect Handlers. In *Trends in Functional Programming*, Meng Wang and Scott Owens (Eds.). Springer International Publishing, Cham, 98–117.

Domainslib 2022. *A library for nested parallel programming.* https://github.com/ocaml-multicore/domainslib

Eio 2022. *Effects-based direct-style IO for multicore OCaml.* https://github.com/ocaml-multicore/eio

Function Colour 2023. *What Color is Your Function?* http://journal.stuffwithstuff.com/2015/02/01/what-color-is-your-function/

Tim Harris, Simon Marlow, and Simon Peyton Jones. 2005. Haskell on a Shared-Memory Multiprocessor. In *Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell* (Tallinn, Estonia) *(Haskell '05)*. Association for Computing Machinery, New York, NY, USA, 49–61. https://doi.org/10.1145/1088348.1088354

Daniel Hillerstrom, Lindley Sam, and Robert Atkey. 2020. Effect handlers via generalised continuations. *Journal of Functional Programming* 30 (2020), e5. https://doi.org/10.1017/S0956796820000040

io_uring 2023. *Efficient IO with io_uring.* https://kernel.dk/io_uring.pdf

Java Virtual threads 2023. *JEP 444: Virtual Threads.* https://openjdk.org/jeps/444

JEP428 2023. *JEP 428: Structured Concurrency (Incubator).* https://openjdk.org/jeps/428

JSGenerators 2023. *function*.* https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/function*#description

Sivaramkrishnan KC, Tim Harris, Simon Marlow, and Simon Peyton Jones. 2016. Composable scheduler activations for Haskell. *Journal of Functional Programming* 26 (2016), e9. https://doi.org/10.1017/S0956796816000071

Kotlin coroutines 2023. *Coroutines.* https://kotlinlang.org/docs/coroutines-overview.html

Daan Leijen. 2017a. Type Directed Compilation of Row-Typed Algebraic Effects. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17)*. Association for Computing Machinery, New York, NY, USA, 486–499. https://doi.org/10.1145/3009837.3009872

Daan Leijen. 2017b. Type Directed Compilation of Row-Typed Algebraic Effects. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17)*. Association for Computing Machinery, New York, NY, USA, 486–499. https://doi.org/10.1145/3009837.3009872

Loom 2023. *Project Loom: Fibers and Continuations for the Java Virtual Machine.* https://cr.openjdk.org/~rpressler/loom/Loom-Proposal.html

Lwt_domain 2023. *Lwt_domain: A library to use domains-based parallelism from Lwt.* https://github.com/ocsigen/lwt_domain

Simon Marlow, Simon Peyton Jones, and Satnam Singh. 2009. Runtime Support for Multicore Haskell. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming* (Edinburgh, Scotland) *(ICFP '09)*. Association for Computing Machinery, New York, NY, USA, 65–78. https://doi.org/10.1145/1596550.1596563

Stephen Muller, Umat K. Acar, and R. Harper. 2017. Responsive parallel computation: bridging competitive and cooperative threading. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 677–692. https://doi.org/10.1145/3140587

OCaml Effects 2023. *Language Extensions : Effect handlers.* https://kcsrk.info/webman/manual/effects.html

OCaml Lazy 2023. *Manual for Lazy module in OCaml.* https://v2.ocaml.org/api/Lazy.html#TYPEt

OCaml threads library 2023. *The threads library.* https://v2.ocaml.org/manual/libthreads.html

Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. 1996. Concurrent Haskell. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg Beach, Florida, USA) *(POPL '96)*. Association for Computing Machinery, New York, NY, USA, 295–308. https://doi.org/10.1145/237721.237794

Gordon Plotkin and Matija Pretnar. 2009. Handlers of Algebraic Effects. In *Programming Languages and Systems*, Giuseppe Castagna (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 80–94.

Gordon D. Plotkin and John Power. 2001. Adequacy for Algebraic Effects. In *Foundations of Software Science and Computation Structure.*

Rayon 2022. *A data parallelism library for the Rust programming language.* https://docs.rs/rayon/latest/rayon/

Rayon Tokio Crate 2021. *Rayon Tokio Crate.* https://github.com/andybarron/tokio-rayon

John H. Reppy. 1999. *Concurrent Programming in ML.* Cambridge University Press. https://doi.org/10.1017/CBO9780511574962

KC Sivaramakrishnan, Stephen Dolan, Leo White, Sadiq Jaffer, Tom Kelly, Anmol Sahoo, Sudha Parimala, Atul Dhiman, and Anil Madhavapeddy. 2020. Retrofitting Parallelism onto OCaml. *Proc. ACM Program. Lang.* 4, ICFP, Article 113 (Aug. 2020), 30 pages. https://doi.org/10.1145/3408995

KC Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. 2021. Retrofitting Effect Handlers onto OCaml. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 206–221. https://doi.org/10.1145/3453483

Tokio 2022. *An asynchronous runtime for the Rust programming language.* https://tokio.rs/

Jérôme Vouillon. 2008. Lwt: A Cooperative Thread Library. In *Proceedings of the 2008 ACM SIGPLAN Workshop on ML* (Victoria, BC, Canada) *(ML '08)*. Association for Computing Machinery, New York, NY, USA, 3–12. https://doi.org/10.1145/1411304.1411307

Wrk2 2020. *A constant throughput, correct latency recording variant of wrk.* https://github.com/giltene/wrk2