Retrofitting Effect Handlers onto OCaml

KC Sivaramakrishnan
IIT Madras
Chennai, India
kcsrk@cse.iitm.ac.in

Stephen Dolan
OCaml Labs
Cambridge, UK
stephen.dolan@cl.cam.ac.uk

Leo White
Jane Street
London, UK
leo@lpw25.net

Tom Kelly
OCaml Labs
Cambridge, UK
tom.kelly@cantab.net

Sadiq Jaffer
Opsian and OCaml Labs
Cambridge, UK
sadiq@toao.com

Anil Madhavapeddy
University of Cambridge and OCaml Labs
Cambridge, UK
avsm2@cl.cam.ac.uk

Abstract

Effect handlers have been gathering momentum as a mechanism for modular programming with user-defined effects. Effect handlers allow for non-local control flow mechanisms such as generators, async/await, lightweight threads and coroutines to be composably expressed. We present a design and evaluate a full-fledged efficient implementation of effect handlers for OCaml, an industrial-strength multi-paradigm programming language. Our implementation strives to maintain the backwards compatibility and performance profile of existing OCaml code. Retrofitting effect handlers onto OCaml is challenging since OCaml does not currently have any non-local control flow mechanisms other than exceptions. Our implementation of effect handlers for OCaml: *(i)* imposes a mean 1% overhead on a comprehensive macro benchmark suite that does not use effect handlers; *(ii)* remains compatible with program analysis tools that inspect the stack; and *(iii)* is efficient for new code that makes use of effect handlers.

CCS Concepts: • **Software and its engineering → Runtime environments**; **Concurrent programming structures**; **Control structures**; *Parallel programming languages*; **Concurrent programming languages**.

Keywords: Effect handlers, Backwards compatibility, Fibers, Continuations, Backtraces

ACM Reference Format:
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 '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3453483.3454039

PLDI '21, June 20–25, 2021, Virtual, Canada

ACM ISBN 978-1-4503-8391-2/21/06.
https://doi.org/10.1145/3453483.3454039

1 Introduction

Effect handlers [45] provide a modular foundation for user-defined effects. The key idea is to separate the definition of the effectful operations from their interpretations, which are given by *handlers* of the effects. For example,

```
effect In_line : in_channel -> string
```

declares an *effect* `In_line`, which is parameterised with an input channel of type `in_channel`, which when *performed* returns a `string` value. A computation can perform the `In_line` effect without knowing how the `In_line` effect is implemented. This computation may be enclosed by different handlers that handle `In_line` differently. For example, `In_line` may be implemented by performing a blocking read on the input channel or performing the read asynchronously by offloading it to an event loop such as `libuv`, without changing the computation. Thanks to the separation of effectful operations from their implementation, effect handlers enable new approaches to modular programming. Effect handlers are a generalisation of exception handlers, where, in addition to the effect being handled, the handler is provided with the delimited continuation [14] of the perform site. This continuation may be used to resume the suspended computation later. This enables non-local control-flow mechanisms such as resumable exceptions, lightweight threads, coroutines, generators and asynchronous I/O to be composably expressed.

One of the primary motivations to extend OCaml with effect handlers is to natively support asynchronous I/O in order to express highly scalable concurrent applications such as web servers in *direct style* (as opposed to using *callbacks*). Many programming languages, including OCaml, require non-local changes to source code in order to support asynchronous I/O, often leading to a dichotomy between synchronous and asynchronous code [10]. For asynchronous I/O, OCaml developers typically use libraries such as Lwt [53] and Async [40, §18], where asynchronous functions are represented as monadic computations. In these libraries, while asynchronous functions can call synchronous functions directly, the converse is not true. In particular, any function that calls an asynchronous function will also have to be marked as asynchronous. As a result, large parts of the applications using these libraries end up being in monadic form.

Languages such as GHC Haskell and Go provide lightweight threads, which avoids the dichotomy between synchronous and asynchronous code. However, these languages bake-in the lightweight thread implementation into the runtime system. With effect handlers, asynchronous I/O can be implemented directly in OCaml as a library without imposing a monadic form on the users.

There are many research languages and libraries built around effect handlers [3, 6, 7, 11, 26, 34]. Unlike these efforts, our goal is to retrofit effect handlers onto the OCaml programming language, which has been in continuous use for the past 25 years in large codebases including verification tools [4, 12], mission critical software systems [39] and latency sensitive networked applications [38]. OCaml is particularly favoured for its competitive yet predictable performance, with a fast foreign-function interface (FFI). It has excellent compatibility with program analysis tools such as debuggers and profilers that utilise DWARF stack unwind tables [18] to obtain a backtrace.

OCaml currently does not support any non-local control flow mechanisms other than exceptions. This makes it particularly challenging to implement the delimited continuations necessary for effect handlers without sacrificing the desirable properties of OCaml. A standard way of implementing continuations is to use continuation-passing style (CPS) in the compiler's intermediate representation (IR) [34]. OCaml does not use a CPS IR, and changing the compiler to utilise a CPS IR would be an enormous undertaking that would affect the performance profile of existing OCaml applications due to the increased memory allocations as the continuation closures get allocated on the heap [20]. Moreover, with CPS, an explicit stack is absent, and hence, we would lose compatibility with tools that inspect the program stack. Hence, we choose not to use CPS translation and represent the continuations as call stacks.

The search for an expressive effect system that guarantees that all the effects performed in the program are handled (*effect safety*) in the presence of advanced features such as polymorphism, modularity and generativity is an active area of research [5, 6, 26, 34]. We do not focus on this question in this paper, and our implementation of effect handlers in OCaml does not guarantee effect safety. We leave the question of effect safety for future work.

1.1 Requirements

We motivate our effect handler design based on the following ideal requirements:

- R1 **Backwards compatibility.** Existing OCaml programs do not break under OCaml extended with effect handlers. OCaml code that does not use effect handlers will pay minimal performance and memory cost.
- R2 **Tool compatibility.** OCaml programs with effect handlers produce well-formed backtraces and remain compatible with program analysis tools such as debuggers and profilers that inspect the stack using DWARF unwind tables.
- R3 **Effect handler efficiency.** The program must accommodate millions of continuations at the same time to support highly-concurrent applications. Installing effect handlers, capturing and resuming continuations must be fast.
- R4 **Forwards compatibility.** As a cornerstone of modularity, we also want blocking I/O code to *transparently* be made asynchronous with the help of effect handlers.

The need to host millions of continuations at the same time rules out the use of a large contiguous stack space as in C for continuations. Instead, we resort to using small initial stacks and growing the stacks on demand. As a result, OCaml functions, irrespective of whether they use effect handlers, need to perform stack overflow checks, and external C functions (which do not have stack overflow checks) must be performed on a separate system stack. Additionally, we must generate DWARF stack unwind tables for stacks that may be non-contiguous. In this work, we develop the compiler and runtime support required for implementing efficient effect handlers for OCaml that satisfy these requirements.

Our work is also timely. The WebAssembly [25] community group is considering effect handlers as one of the mechanisms for supporting concurrency, asynchronous I/O and generators [54]. Project Loom [37] is an OpenJDK project that adds virtual threads and delimited continuations to Java. The Swift roadmap [52] includes direct style asynchronous programming and structured concurrency as milestones. We believe that our design choices will inform similar choices to be made in other industrial-strength languages.

1.2 Contributions

Our contributions are to present:

- the design and implementation of effect handlers for OCaml. Our design retains OCaml's compatibility with program analysis tools that inspect the stack using DWARF unwind tables. We have validated our DWARF unwind tables with the assistance of an automated validator tool [2].
- a formal operational semantics for the effect handler implementation in OCaml. Our formalism explicitly models the interactions with the C stack, which is generally overlooked by other formal models, but which the implementations must handle.
- extensive evaluation which shows that our implementation has minimal impact on code that does not use effect handlers, and serves as an efficient foundation for scalable concurrent programming.

We have implemented effect handlers in a multicore extension of the OCaml programming language which we call **Multicore OCaml** to distinguish it from *stock OCaml*. Multicore OCaml delineates concurrency (overlapped execution

of tasks) from parallelism (simultaneous execution of tasks) with distinct mechanisms for expressing them. Sivaramakrishnan et al. [49] describe the parallelism support in Multicore OCaml enabled by *domains*. The focus of this paper is the concurrency support enabled by effect handlers.

The remainder of the paper continues with a description of the stock OCaml program stack (§2). We then describe effect handlers in Multicore OCaml focussing on the challenges in retrofitting them into a mainstream *systems* language (§3), followed by the static and dynamic semantics for Multicore OCaml effect handlers (§4). We then discuss the compiler and the runtime system support for implementing effect handlers (§5), and present an extensive performance evaluation of effect handlers (§6) against our design goals (§1.1). Finally, we discuss the related work (§7) and conclude (§8).

2 Background: OCaml Stacks

The main challenge in implementing effect handlers in Multicore OCaml is managing the program stack and preserving its desirable properties. In this section, we provide an overview of the program stack and related mechanisms in stock OCaml.

Consider the layout of the stock OCaml stack for the program shown in Figures 1a and 1b. The OCaml main function `omain` installs two exception handlers `h1` and `h2` to handle the exceptions `E1` and `E2`. `omain` calls the external C function `ocaml_to_c`, which in turn calls back into the OCaml function `c_to_ocaml`, which raises the exception `E1`. OCaml supports raising exceptions in C as well as throwing exceptions across external calls. Hence, the exception `E1` gets caught in the handler `h1`, and `omain` returns `42`. The layout of the stack in the native code backend just before raising the exception in `c_to_ocaml` is illustrated in Figure 1c. Note that the stack grows downwards.

OCaml uses the same program stack as C, and hence the stack has alternating sequences of C and OCaml frames. However, unlike C, OCaml does not create pointers into OCaml frames. OCaml uses the hardware support for `call` and `return` instructions for function calls and returns. OCaml does not perform explicit stack overflow checks in code, and, just like C, relies on the guard page at the end of the stack region to detect stack overflow. Stack overflow is detected by a memory fault and a `Stack_overflow` exception is raised to unwind the stack.

2.1 External calls and callbacks

OCaml does not use the C calling convention. In particular, there are no callee-saved registers in OCaml. In the `x86-64` backend, the OCaml runtime makes use of two C callee-saved registers for supporting OCaml execution. The register `r15` holds the *allocation pointer* into the minor heap used for bump pointer allocation, and `r14` holds a reference to the `Caml_state`, a table of global variables used by the runtime. This makes external calls extremely fast in OCaml. If the external function does not allocate in the OCaml heap, then it can be called directly and no bookkeeping is necessary. For external functions which allocate in the OCaml heap, the cached allocation pointer is saved to `Caml_state` before the external call and it is restored on return. Similarly, callbacks into OCaml from C are also cheap: these involve loading the arguments in the right registers and calling the OCaml function. OCaml callbacks are relatively common as the garbage collector (GC), which is implemented in C, executes OCaml finalisation functions as callbacks.

2.2 Exception handlers

The lack of callee-saved registers also makes exception handling fast. In the absence of callee-saved registers, no registers need to be saved when entering a `try` block. Similarly, no registers need to be restored when handling an exception. Installing an exception handler simply pushes the program counter (`pc`) of the handler and the current exception pointer (`exn_ptr` – a field in `Caml_state`) onto the stack. After this, the current exception pointer is updated to be the current stack pointer (`rsp`). This creates a linked-list of exception handler frames on the stack as shown in Figure 1c. Raising an exception simply sets `rsp` to `exn_ptr`, loads the saved `exn_ptr`, and jumps to the `pc` of the handler.

In order to forward exceptions across C frames, the C stub function `caml_call_ocaml`, pushes an exception handler frame that either forwards the exception to the innermost OCaml exception handler (`raise_exn_c` in Figure 1c) or prints a fatal error (`fatal_uncaught`) if there are no enclosing handlers. Exceptions are so cheap in OCaml that it is common to use them for *local* control flow.

2.3 Stack unwinding

OCaml generates *stack maps* in order to accurately identify roots on the stack for assisting the GC. For every call point in the program, the OCaml compiler emits the size of the frame and the set of all live registers in the frame that point to the heap. During a GC, the OCaml stack is walked and the roots are marked, skipping over the C frames.

OCaml also generates precise DWARF unwind information for OCaml, thanks to which debuggers such as `gdb` and `lldb`, and profilers such as `perf` work out-of-the-box. For example, for the program in Figures 1a and 1b, one could set a break point in `gdb` at `caml_raise_exn` to get the backtrace in Figure 1d which corresponds to the stack in Figure 1c.

The same backtrace can also be obtained by using *frame pointers* instead of DWARF unwind tables. OCaml allows compiling code with frame pointers, but they are not enabled by default. The OCaml stack tends to be deep with small frames due to the pervasive use of recursive functions, not all of which are tail-recursive. Hence, the addition of frame pointers can significantly increase the size of the stack[1]. Moreover, not using frame pointers saves two instructions

[1] https://github.com/ocaml/ocaml/issues/5721#issuecomment-472965549

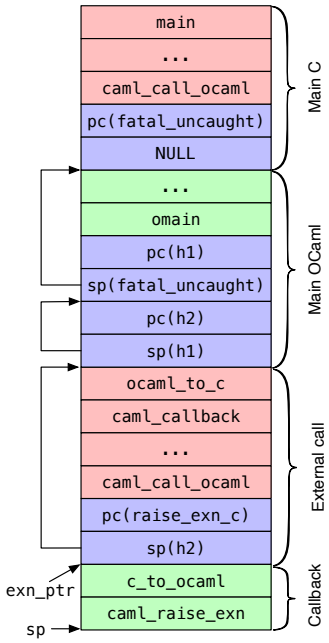
```
external ocaml_to_c
  : unit -> int = "ocaml_to_c"
exception E1
exception E2
let c_to_ocaml () = raise E1
let _ = Callback.register
    "c_to_ocaml" c_to_ocaml
let omain () =
  try (* h1 *)
    try (* h2 *) ocaml_to_c ()
    | with E2 -> 0
  | with E1 -> 42;;
let _ = assert (omain () = 42)
```

(a) meander.ml

```
#include <caml/mlvalues.h>
#include <caml/callback.h>

value ocaml_to_c (value unit) {
  caml_callback(*caml_named_value
    ("c_to_ocaml"), Val_unit);
  return Val_int(0);
}
```

(b) meander.c



(c) Stack layout before raise E1.

```
#0  0x925dc in caml_raise_exn ()
#1  0x6fd3e in camlMeander__c_to_ocaml_83 () at
     meander.ml:5
#2  0x925a4 in caml_call_ocaml ()
#3  0x8a84a in caml_callback_exn (...) at
     callback.c:145
#4  caml_callback (...) at callback.c:199
#5  0x76e0a in ocaml_to_c (unit=1) at meander.c:5
#6  0x6fd77 in camlMeander__omain_88 () at
     meander.ml:10
#7  0x6fe92 in camlMeander__entry () at
     meander.ml:13
#8  0x6f719 in caml_program ()
#9  0x925a4 in caml_call_ocaml ()
#10 0x92e4c in caml_startup_common (...) at
     startup_nat.c:162
#11 0x92eab in caml_startup_exn (...) at
     startup_nat.c:167
#12 caml_startup (...) at startup_nat.c:172
#13 0x6f55c in main (...) at main.c:44
```

(d) gdb backtrace before raise E1.

Figure 1. Program stack on stock OCaml.

in the function prologue and epilogue, and makes an extra register (rbp on x86_64) available. Note that the DWARF unwind information is complementary to the information used by OCaml to walk the stack for GC.

3 Effect Handlers

In this section, we describe the effect handlers in Multicore OCaml, and refine the design to retrofit them onto OCaml.

3.1 Asynchronous I/O

Since our primary motivation is to enable composable asynchronous I/O, let us implement a cooperative lightweight thread library with support for forking new threads and yielding control to other threads. We will then extend this library with support for synchronously reading from channels and subsequently make it asynchronous *without changing the client code for asynchrony*. In order to support forking and yielding threads, we declare the following effects:

```
effect Fork : (unit -> unit) -> unit
effect Yield : unit
```

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:

```
let fork f = perform (Fork f)
let yield () = perform Yield
```

The implementation of the scheduler queue is defined in the run function, which handles the effects appropriately:

```
1  let run main =
2    let runq = Queue.create () in
3    let suspend k = Queue.push k runq in
4    let rec run_next () =
5      match Queue.pop runq with
6      | k -> continue k ()
7      | exception Queue.Empty -> ()
8    in
9    let rec spawn f =
10     match f () with
11     | () -> run_next () (* value case *)
12     | effect Yield k -> suspend k; run_next ()
13     | effect (Fork f') k -> suspend k; spawn f'
14   in
15   spawn main
```

The function spawn (line 9) evaluates the computation f in an effect handler. The computation f may return normally with a value, or perform effects Fork f' and Yield. The pattern effect Yield k handles the effect Yield and binds k to the continuation of the corresponding perform delimited by this handler. The scheduler queue runq maintains a queue of these continuations. suspend pushes continuations into the queue, run_next pops continuations from the queue and resumes them with () value using the continue primitive. In the case of the Yield effect, we suspend the current continuation k and resume the next available continuation. In the case of the Fork f' effect, we suspend the current continuation and recursively call spawn on f' in order to run f' concurrently. Observe that we can change the scheduling algorithm from FIFO to LIFO by changing the scheduler queue to a stack.

We can implement support for synchronous read from channels by adding the following case to the effect handler in `spawn`:

```
let rec spawn f =
  match f () with
  ...
  | effect (In_line ic) k -> continue k (input_line ic)
```

This uses OCaml's standard `input_line` function to read a line synchronously from the channel `ic` and resume the continuation `k` with the resultant string. However, performing reads synchronously blocks the entire scheduler, preventing other threads from running until the I/O is completed.

We can make the I/O asynchronous by modifying the `run` function as follows:

```
1  let run main =
2    let runq = Queue.create () in
3    let suspend k = Queue.push (continue k) runq in
4    let pending_reads = ref [] in
5    let rec run_next () =
6      match Queue.pop runq with
7      | f -> f ()
8      | exception Queue.Empty ->
9        match !pending_reads with
10       | [] -> () (* no pending reads *)
11       | todo ->
12         let compl,todo = do_reads todo in
13         List.iter (fun (s,k) ->
14          Queue.push (fun () -> continue k s) runq) compl;
15         pending_reads := todo;
16         run_next ()
17   in
18   let rec spawn f =
19     match f () with
20     | () -> run_next () (* value case *)
21     | effect Yield k -> suspend k; run_next ()
22     | effect (Fork f') k -> suspend k; spawn f'
23     | effect (In_line ic) k ->
24       pending_reads := (ic,k)::!pending_reads;
25       run_next ()
26   in
27   spawn main
```

The scheduler queue `runq` now holds thunks instead of continuations. The value `pending_reads` maintains a list of pending reads and the associated continuations (line 4). At line 24, we handle the `In_line` effect by pushing the pair of input channel `ic` and continuation `k` to `pending_reads`, allowing other threads in the scheduler to run.

When the scheduler queue is empty, the `run_next` function performs the pending reads. We abstract away the details of the event-based I/O using the `do_reads` function (line 12). `do_reads` takes a list of pending reads and blocks until at least one of the reads succeeds. It returns a pair of lists `compl` and `todo`. `compl` contains the result strings from successful reads and corresponding continuations. These continuations are arranged to be resumed with the read result and pushed into the scheduler queue. `todo` contains the channels on which input is still pending and their corresponding continuations. `pending_reads` is updated to point to the `todo` list so that they may be attempted later. Observe that all of the changes to add asynchrony are localised to the `run` function, and the computation that performs these effects can remain in direct style (as opposed to the monadic-style in Lwt and Async).

This example does not resume a continuation more than once. This also holds true for other use cases such as generators and coroutines. Hence, our continuations are one-shot, and resuming the continuation more than once raises an `Invalid_argument` exception. It is well-known that one-shot continuations can be implemented efficiently [8].

While OCaml permits throwing exceptions across C frames, we do not allow effects to propagate across C frames as the C frames would become part of the captured continuation. Managing C frames as part of the continuation is a complex endeavour [33], and we find that the complexity budget outweighs the relatively fewer mechanisms enabled by this addition in our setting.

3.2 Resource cleanup

The interaction of non-local control flow with systems programming is quite subtle [17, 35]. Consider the following function that uses blocking I/O functions from the OCaml standard library to copy data from the input channel `ic` to the output channel `oc`:

```
let copy ic oc =
  let rec loop () =
    output_string oc ((input_line ic) ^ "\n"); loop () in
  try loop () with
  | End_of_file -> close_in ic; close_out oc
  | e -> close_in ic; close_out oc; raise e
```

The function `input_line` raises an `End_of_file` exception on reaching the end of input, which is handled by the exception handler which closes the channels. The `close_*` functions do nothing if the channel is already closed. The code is written in a defensive style to handle other exceptional cases such as the channels being closed externally. Both `input_line` and `output_string` raise a `Sys_error` exception if the channel is closed. In this case, the catch-all exception handler closes the channels and reraises the exception to communicate the exceptional behaviour to the caller.

One of our goals (§1.1) is to make this code transparently asynchronous. We can define effects for performing the I/O operations and wrap them up in functions with the same signature as the one from the standard library:

```
effect In_line : in_channel -> string
effect Out_str : out_channel * string -> unit
let input_line ic = perform (In_line ic)
let output_string oc s = perform (Out_str (oc, s))
```

We can then use the `run` function that we defined earlier, to discharge the I/O operations asynchronously and resume

with the result. While this handles value return cases, what about the exceptional cases `End_of_file` and `Sys_error`? To this end, we introduce a `discontinue` primitive to resume a continuation by raising an exception. In this example, on reaching the end of file, we would discontinue the captured continuation of the `input_line` function with `discontinue k End_of_file`, which raises the exception at `input_line` call site, and the open channels will be closed.

OCaml programs that use resources such as channels are usually written defensively with the assumption that calling a function will return *exactly once*, either normally or exceptionally. Since effect handlers in Multicore OCaml do not ensure that all the effects are handled, if the function performs an effect with no matching handler, then the function *will not return at all*. To remedy this, when such an effect bubbles up to the top-level, we discontinue the continuation with an `Unhandled` exception so that the exception handlers may run and clean up the resources.

4 Semantics

In this section, we formalise the effect handler design for Multicore OCaml.

4.1 Static semantics

As mentioned earlier, effect handlers in Multicore OCaml do not guarantee effect safety. Programs without matching effect handlers are well-typed Multicore OCaml programs. As a result, our static semantics is simpler than languages that ensure effect safety [3, 5, 11, 26, 34, 47]. This is important for backwards compatibility as our goal is to retrofit effect handlers to a language with large legacy codebases; programs that do not use effects remain well-typed, and those that do compose well with those that don't.

The static semantics of effect handlers in OCaml is captured succinctly by its API:

```
type 'a eff = ..
type ('a,'b) continuation
val perform: 'a eff -> 'a
val continue: ('a,'b) continuation -> 'a -> 'b
val discontinue: ('a,'b) continuation -> exn -> 'b
(* Internal API *)
type 'a comp = unit -> 'a
type ('a,'b) handler =
{retc: 'a -> 'b;
 effc: 'c.'c eff -> ('c,'b) continuation -> 'b; }
val match_with: 'a comp -> ('a,'b) handler -> 'b
```

We introduce an extensible variant type [44] `'a eff` of effect values, which when performed using the `perform` primitive returns an `'a` value. Constructors for the value of type `'a eff` are declared using the effect declarations. For example, the declaration `effect E : string -> int` is syntactic sugar for adding a new constructor to the variant type `type _ eff += E : string -> int eff`. We introduce the type `('a,'b) continuation` of delimited continuations which expects a `'a` value for resumption and returns a `'b` value. The continuations may be continued with a suitably typed value or discontinued with an exception.

For effect handling, we extend OCaml's `match ... with` syntax with effect patterns. The expression

```
match e with
| None -> false | Some b -> b
| effect (E s) k1 -> e1 | effect (F f) k2 -> e2
```

is translated to the equivalent of

```
match_with (fun () -> e)
{ retc = (function None -> false | Some b -> b);
  effc = (function
  | (E s) -> (fun k1 -> e1)
  | (F f) -> (fun k2 -> e2)
  | e -> (fun k -> match perform e with
          | v -> continue k v
          | exception e -> discontinue k e)); }
```

For the sake of exposition, we introduce a `('a,'b) handler` type. This handler handles a `'a comp` that returns a `'a` value, and itself returns a `'b` value. The handler has a return field `retc` of type `'a -> 'b`. The effect field `effc` handles effects of type `'c eff` with `('c,'b) continuation` and returns a value of type `'b`. The last case in `effc` *reperforms* any unmatched effect to the outer handler and returns the value and exceptions back to the original performer. In the implementation, reperform is implemented as a primitive to avoid executing code on the resumption path.

4.2 Dynamic semantics

We present an operational semantics for a core language of effect handlers that faithfully captures the semantics of the Multicore OCaml implementation. An executable version of the semantics, implemented as an OCaml interpreter, along with examples, is included in the supplementary material.

4.2.1 Syntax. Our expressions (Figure 2a) consist of integer constants (n), variables (x), abstraction ($\Lambda x.e$), application ($e\ e$), arithmetic expressions ($e \odot e$) where $\odot$ ranges over $\{+, -, *, /\}$, raising exceptions (**raise** $l\ e$), performing effects (**perform** $l\ e$), and handling effects (**match** e **with** h). Abstractions come in two forms: OCaml abstractions (λ^o) and C abstractions (λ^c). The handler consists of a return case (**return** $x \mapsto e$), zero or more exception cases (**exception** $l\ x \mapsto e$) with label l, parameter x and body e, and zero or more effect cases (**effect** $l\ x\ k \mapsto e$) with label l, parameter x, continuation k and body e.

The operational semantics is an extension of the CEK machine semantics [21] for effect handlers, following the abstract machine semantics of Hillerstrom et al. [26]. The key difference from Hillerstrom et al. is that our stacks are composed of alternating sequence of OCaml and C stack segments. The program state is captured as configuration $\mathfrak{C} := \|\tau, \epsilon, \sigma\|$ with the current term τ under evaluation, its environment ϵ and the current stack σ. The term is either an expression e or a value v. The values are integer constants n,

$$
\begin{array}{rrcl}
\text{Constants} & n & \coloneqq & \mathbb{Z} \\
\text{Abstractions} & \Lambda & \coloneqq & \lambda^o \mid \lambda^c \\
\text{Expressions} & e & \coloneqq & n \mid x \mid e\ e \mid \Lambda x.e \mid e \odot e \mid \textbf{raise}\ l\ e \\
 & & \mid & \textbf{match}\ e\ \textbf{with}\ h \mid \textbf{perform}\ l\ e \\
\text{Handlers} & h & \coloneqq & \{\textbf{return}\ x \mapsto e\} \mid \{\textbf{exception}\ l\ x \mapsto e\} \uplus h \\
 & & \mid & \{\textbf{effect}\ l\ x\ k \mapsto e\} \uplus h \\
\text{Values} & v & \coloneqq & n \mid k \mid (\!|\Lambda x.e, \epsilon|\!) \mid \mathsf{eff}\ l \mid \mathsf{exn}\ l \\
\text{Frames} & r & \coloneqq & \langle e\ \epsilon\rangle_{\mathrm{a}} \mid \langle v\rangle_{\mathrm{f}} \mid \langle \odot\ e\ \epsilon\rangle_{\mathrm{b1}} \mid \langle \odot\ \mathbb{N}\rangle_{\mathrm{b2}} \\
\text{Environments} & \epsilon & \coloneqq & \emptyset \mid \epsilon[x \mapsto v]
\end{array}
$$

$$
\begin{array}{rrcl}
\text{Handler Closures} & \eta & \coloneqq & (h, \epsilon) \\
\text{Frame List} & \psi & \coloneqq & [] \mid r :: \psi \\
\text{Fibers} & \varphi & \coloneqq & (\psi, \eta) \\
\text{Continuations} & k & \coloneqq & [] \mid \varphi \lhd k \\
\text{C stacks} & \gamma & \coloneqq & \lceil \psi, \omega \rceil_{\mathrm{c}} \\
\text{OCaml stacks} & \omega & \coloneqq & \lceil k, \gamma \rceil_{\mathrm{o}} \mid \bullet \\
\text{Stacks} & \sigma & \coloneqq & \gamma \mid \omega \\
\text{Terms} & \tau & \coloneqq & e \mid v \\
\text{Configurations} & \mathfrak{C} & \coloneqq & \|\tau, \epsilon, \sigma\|
\end{array}
$$

(a) Syntax of expressions and configurations

$$\textsc{StepC}\ \frac{(\tau, \epsilon, \psi, \omega) \xrightarrow{\mathrm{c}} \mathfrak{C}}{\|\tau, \epsilon, \lceil \psi, \omega \rceil_{\mathrm{c}}\| \to \mathfrak{C}} \qquad \textsc{StepO}\ \frac{(\tau, \epsilon, k, \gamma) \xrightarrow{\mathrm{o}} \mathfrak{C}}{\|\tau, \epsilon, \lceil k, \gamma \rceil_{\mathrm{o}}\| \to \mathfrak{C}}$$

(b) Top-level reductions

$$
\begin{array}{rrcl}
\textsc{Var} & (x, \epsilon, \psi) & \rightsquigarrow & (\epsilon(x), \epsilon, \psi) \\
\textsc{Arith1} & (e_1 \odot e_2, \epsilon, \psi) & \rightsquigarrow & (e_1, \epsilon, \langle \odot\ e_2\ \epsilon\rangle_{\mathrm{b1}} :: \psi) \\
\textsc{Arith2} & (n_1, \_, \langle \odot\ e_2\ \epsilon\rangle_{\mathrm{b1}} :: \psi) & \rightsquigarrow & (e_2, \epsilon, \langle \odot\ n_1\rangle_{\mathrm{b2}} :: \psi) \\
\textsc{Arith3} & (n_2, \epsilon, \langle \odot\ n_1\rangle_{\mathrm{b2}} :: \psi) & \rightsquigarrow & ([\![n_1 \odot n_2]\!], \epsilon, \psi) \\
\textsc{App1} & (e_1\ e_2, \epsilon, \psi) & \rightsquigarrow & (e_1, \epsilon, \langle e_2\ \epsilon\rangle_{\mathrm{a}} :: \psi) \\
\textsc{App2} & (\Lambda x.e, \epsilon, \psi) & \rightsquigarrow & ((\!|\Lambda x.e, \epsilon|\!), \epsilon, \psi) \\
\textsc{App3} & ((\!|\Lambda x.e_1, \epsilon_1|\!), \_, \langle e_2\ \epsilon_2\rangle_{\mathrm{a}} :: \psi) & \rightsquigarrow & (e_2, \epsilon_2, \langle (\!|\Lambda x.e_1, \epsilon_1|\!)\rangle_{\mathrm{f}} :: \psi) \\
\textsc{Resume1} & (k, \_, \langle e_1\ \epsilon_1\rangle_{\mathrm{a}} :: \langle e_2\ \epsilon_2\rangle_{\mathrm{a}} :: \psi) & \rightsquigarrow & (e_1, \epsilon_1, \langle k\rangle_{\mathrm{f}} :: \langle e_2\ \epsilon_2\rangle_{\mathrm{a}} :: \psi) \\
\textsc{Resume2} & ((\!|\Lambda x.e_1, \epsilon_1|\!), \_, \langle k\rangle_{\mathrm{f}} :: \langle e_2\ \epsilon_2\rangle_{\mathrm{a}} :: \psi) & \rightsquigarrow & (e_2, \epsilon_2, \langle k\rangle_{\mathrm{f}} :: \langle (\!|\Lambda x.e_1, \epsilon_1|\!)\rangle_{\mathrm{f}} :: \psi) \\
\textsc{Perform} & (\textbf{perform}\ l\ e, \epsilon, \psi) & \rightsquigarrow & (e, \epsilon, \langle \mathsf{eff}\ l\ [[], (\{\textbf{return}\ x \mapsto x\}, \emptyset)]\rangle_{\mathrm{f}} :: \psi) \\
\textsc{Raise} & (\textbf{raise}\ l\ e, \epsilon, \psi) & \rightsquigarrow & (e, \epsilon, \langle \mathsf{exn}\ l\rangle_{\mathrm{f}} :: \psi)
\end{array}
$$

(c) Administrative Reductions – $(\tau, \epsilon, \psi) \rightsquigarrow (\tau, \epsilon, \psi)$.

$$
\begin{array}{rrcll}
\textsc{AdminC} & (\tau, \epsilon, \psi, \omega) & \xrightarrow{\mathrm{c}} & \|\tau', \epsilon', \lceil \psi', \omega \rceil_{\mathrm{c}}\| & \text{if } (\tau, \epsilon, \psi) \rightsquigarrow (\tau', \epsilon', \psi') \\
\textsc{CallC} & (v, \_, \langle (\!|\lambda^c x.e, \epsilon|\!)\rangle_{\mathrm{f}} :: \psi, \omega) & \xrightarrow{\mathrm{c}} & \|e, \epsilon[x \mapsto v], \lceil \psi, \omega \rceil_{\mathrm{c}}\| & \\
\textsc{Callback} & (v, \_, \langle (\!|\lambda^o x.e, \epsilon|\!)\rangle_{\mathrm{f}} :: \psi, \omega) & \xrightarrow{\mathrm{c}} & \|e, \epsilon[x \mapsto v], \lceil k, \lceil \psi, \omega \rceil_{\mathrm{c}} \rceil_{\mathrm{o}}\| & \text{if } k = [[], (\{\textbf{return}\ x \mapsto x\}, \emptyset)] \\
\textsc{RetToO} & (v, \epsilon, [], \lceil k, \gamma \rceil_{\mathrm{o}}) & \xrightarrow{\mathrm{c}} & \|v, \epsilon, \lceil k, \gamma \rceil_{\mathrm{o}}\| & \\
\textsc{ExnFwdO} & (v, \epsilon, \langle \mathsf{exn}\ l\rangle_{\mathrm{f}} :: \_, \lceil (\psi, \eta) \lhd k, \gamma \rceil_{\mathrm{o}}) & \xrightarrow{\mathrm{c}} & \|v, \epsilon, \lceil (\langle \mathsf{exn}\ l\rangle_{\mathrm{f}} :: \psi, \eta) \lhd k, \gamma \rceil_{\mathrm{o}}\| &
\end{array}
$$

(d) C Reductions – $(\tau, \epsilon, \psi, \omega) \xrightarrow{\mathrm{c}} \mathfrak{C}$.

$$
\begin{array}{rrcll}
\textsc{AdminO} & (\tau, \epsilon, (\psi, \eta) \lhd k, \gamma) & \xrightarrow{\mathrm{o}} & \|\tau', \epsilon', \lceil (\psi', \eta) \lhd k, \gamma \rceil_{\mathrm{o}}\| & \text{if } (\tau, \epsilon, \psi) \rightsquigarrow (\tau', \epsilon', \psi') \\
\textsc{CallO} & (v, \_, (\langle (\!|\lambda^o x.e, \epsilon|\!)\rangle_{\mathrm{f}} :: \psi, \eta) \lhd k, \gamma) & \xrightarrow{\mathrm{o}} & \|e, \epsilon[x \mapsto v], \lceil (\psi, \eta) \lhd k, \gamma \rceil_{\mathrm{o}}\| & \\
\textsc{ExtCall} & (v, \_, (\langle (\!|\lambda^c x.e, \epsilon|\!)\rangle_{\mathrm{f}} :: \psi, \eta) \lhd k, \gamma) & \xrightarrow{\mathrm{o}} & \|e, \epsilon[x \mapsto v], \lceil [], \lceil (\psi, \eta) \lhd k, \gamma \rceil_{\mathrm{o}} \rceil_{\mathrm{c}}\| & \\
\textsc{RetToC} & (v, \_, [([], (h, \emptyset))], \gamma) & \xrightarrow{\mathrm{o}} & \|v, \epsilon, \gamma\| & \text{if } h = \{\textbf{return}\ x \mapsto x\} \\
\textsc{RetFib} & (v, \_, ([], (h, \epsilon)) \lhd k, \gamma) & \xrightarrow{\mathrm{o}} & \|e, \epsilon[x \mapsto v], \lceil k, \gamma \rceil_{\mathrm{o}}\| & \text{if } \{\textbf{return}\ x \mapsto e\} \in h \text{ and } k \neq [] \\
\textsc{Handle} & (\textbf{match}\ e\ \textbf{with}\ h, \epsilon, k, \gamma) & \xrightarrow{\mathrm{o}} & \|e, \epsilon, \lceil ([], (h, \epsilon)) \lhd k, \gamma \rceil_{\mathrm{o}}\| & \\
\textsc{ExnHn} & (v, \_, (\langle \mathsf{exn}\ l\rangle_{\mathrm{f}} :: \_, (h, \epsilon)) \lhd k', \gamma) & \xrightarrow{\mathrm{o}} & \|e, \epsilon[x \mapsto v], \lceil k', \gamma \rceil_{\mathrm{o}}\| & \text{if } \{\textbf{exception}\ l\ x \mapsto e\} \in h \\
\textsc{ExnFwdC} & (v, \epsilon, [\langle \mathsf{exn}\ l\rangle_{\mathrm{f}} :: \_, (h, \_)], \lceil \psi', \omega \rceil_{\mathrm{c}}) & \xrightarrow{\mathrm{o}} & \|v, \epsilon, \lceil \langle \mathsf{exn}\ l\rangle_{\mathrm{f}} :: \psi', \omega \rceil_{\mathrm{c}}\| & \text{if } \{\textbf{exception}\ l\ \_ \mapsto \_\} \notin h \\
\textsc{ExnFwdFib} & (v, \epsilon, (\langle \mathsf{exn}\ l\rangle_{\mathrm{f}} :: \_, (h, \_)) \lhd (\psi', \eta') \lhd k', \gamma) & \xrightarrow{\mathrm{o}} & \|v, \epsilon, \lceil (\langle \mathsf{exn}\ l\rangle_{\mathrm{f}} :: \psi', \eta') \lhd k', \gamma \rceil_{\mathrm{o}}\| & \text{if } \{\textbf{exception}\ l\ \_ \mapsto \_\} \notin h \\
\textsc{EffHn} & (v, \_, (\langle \mathsf{eff}\ l\ k\rangle_{\mathrm{f}} :: \psi, (h, \epsilon)) \lhd k', \gamma) & \xrightarrow{\mathrm{o}} & \|e, \epsilon[r \mapsto k''][x \mapsto v], \lceil k', \gamma \rceil_{\mathrm{o}}\| & \text{if } \{\textbf{effect}\ l\ x\ r \mapsto e\} \in h \text{ and } k'' = k \mathbin{@} [(\psi, (h, \epsilon))] \\
\textsc{EffFwd} & (v, \epsilon', (\langle \mathsf{eff}\ l\ k\rangle_{\mathrm{f}} :: \psi, (h, \epsilon)) \lhd (\psi', \eta') \lhd k', \gamma) & \xrightarrow{\mathrm{o}} & \|v, \epsilon', \lceil (\langle \mathsf{eff}\ l\ k''\rangle_{\mathrm{f}} :: \psi', \eta') \lhd k', \gamma \rceil_{\mathrm{o}}\| & \text{if } \{\textbf{effect}\ l\ \_\ \_ \mapsto \_\} \notin h \text{ and } k'' = k \mathbin{@} [(\psi, (h, \epsilon))] \\
\textsc{EffUnHn} & (v, \_, [\langle \mathsf{eff}\ l\ k\rangle_{\mathrm{f}} :: \psi, (h, \epsilon)], \gamma) & \xrightarrow{\mathrm{o}} & \|e, \emptyset, \lceil k \mathbin{@} [(\psi, (h, \epsilon))], \gamma \rceil_{\mathrm{o}}\| & \text{if } \{\textbf{effect}\ l\ \_\ \_ \mapsto \_\} \notin h \text{ and } e = \textbf{raise}\ \mathsf{Unhandled}\ 0 \\
\textsc{Resume} & (v, \_, (\langle k\rangle_{\mathrm{f}} :: \langle (\!|\lambda^o x.e, \epsilon|\!)\rangle_{\mathrm{f}} :: \psi, \eta) \lhd k', \gamma) & \xrightarrow{\mathrm{o}} & \|e, \epsilon[x \mapsto v], \lceil k \mathbin{@} ((\psi, \eta) \lhd k'), \gamma \rceil_{\mathrm{o}}\| &
\end{array}
$$

(e) OCaml Reductions – $(\tau, \epsilon, k, \gamma) \xrightarrow{\mathrm{o}} \mathfrak{C}$.

Figure 2. Operational semantics of Multicore OCaml effect handlers.

continuations k, closures $(\!|\Lambda x.e, \epsilon|\!)$, effects being performed (eff l k) and exceptions being raised (exn l). The environment is a map from variables to values.

The stack σ is either a C stack (γ) or an OCaml stack (ω). The C stack $\lceil \psi, \omega \rceil_c$ consists of a list of frames ψ, and the OCaml stack ω under it. The OCaml stack is either empty $\bullet$ or non-empty $\lceil k, \gamma \rceil_o$ with the current continuation k and the C stack γ under it. Thus, the program stack is an alternating sequence of C and OCaml stacks terminating with an empty OCaml stack $\bullet$. The frame list ψ is composed of individual frames r, which is one of an argument frame $\langle e\ \epsilon \rangle_a$ with the expression e at the argument position of an application with its environment ϵ, a function frame $\langle v \rangle_f$ with the value v at the function position of an application, and frames for evaluating the arguments of an arithmetic expression.

A continuation k is either empty [] or a non-empty list of *fibers*. A fiber $\varphi := (\psi, \eta)$ is a list of frames ψ and a handler closure $\eta := (h, \epsilon)$, which is a pair of handler h and its environment ϵ. We use the infix operator @ for appending two lists.

4.2.2 Top-level reductions.

The initial configuration for an expression e is $\|e, \emptyset, \lceil [], \bullet \rceil_c \|$, where the environment and the stack are empty. The top-level reductions (Figure 2b) can be performed by either by taking a C step $\xrightarrow{c}$ or an OCaml step $\xrightarrow{o}$.

4.2.3 C reductions.

We can take a C step (Figure 2d) by taking an administrative reduction step $\rightsquigarrow$. The administrative reductions are common to both C and OCaml. The rules Var, Arith1, Arith2, App1, App2 and App3 are standard. Arith3 performs the arithmetic operation on the integers ($[\![n_1 \odot n_2]\!]$). Raise pushes an function frame with exception value to indicate that an exception is being raised. Similarly, Perform pushes a function frame with an effect value with an empty continuation $[[], (\{\textbf{return}\ x \mapsto x\}, \emptyset)]$ with no captured frames and an empty handler with an identity return case alone. We shall return to Resume1 and Resume2 in the next subsection.

Continuing with the rest of the C reduction steps, CallC captures the behaviour of calling a C function. Since the program is currently executing C, we can perform the call on the current stack. In case the abstraction is an OCaml abstraction (Callback), we create an OCaml stack with the C stack as its tail, with the current continuation being empty. This captures the behaviour of calling back into OCaml from C. RetToO returns a value to the enclosing OCaml stack. ExnFwdO forwards a raised exception to the enclosing OCaml stack, unwinding the rest of the frames. This captures the semantics of raising OCaml exceptions from C.

4.2.4 OCaml reductions.

In OCaml (Figure 2e), reductions always occur on the top-most fiber in the current stack. AdminO performs administrative reductions. CallO evaluates an OCaml function on the current stack. ExtCall captures the behaviour of external calls, which are evaluated on an empty C stack with the current OCaml stack as its tail. RetToC returns a value to the enclosing C stack. In this case, we have exactly one fiber on the stack, and this was created in the rule Callback, whose handler has identity return case alone and the environment is empty. RetFib returns the value from a fiber to the previous one, evaluating the body of the return case.

The rule Handle installs a handler by pushing a fiber with no frames and the given handler. The rule ExnHn handles an exception, if the current handler has a matching exception case, unwinding the current fiber. The rule ExnFwdC forwards the exception to C. Here, there is exactly one fiber on the current stack, and the handler does not have a matching exception case, which we know is the case (see Callback rule). The rule ExnFwdFib forwards the exception to the next fiber if the current handler does not handle it.

The rule EffHn captures the handling of effects when the current handler has a matching effect case. We evaluate the body of the matching case, and bind the continuation parameter `r` to the captured continuation `k''`. Observe that the captured continuation `k''` includes the current handler. Intuitively, the handler wraps around captured continuation. This gives Multicore OCaml effect handlers deep handler semantics [26]. EffFwd forwards the effect to the outer fiber, and extends the captured continuation `k''` in the process. Recall that we do not capture C frames as part of a continuation. To this end, EffUnHn models unhandled effect. If the effect bubbles up to the top fiber — which we know does not have an effect case (see Callback rule) — we raise Unhandled exception at the point where the corresponding effect was performed. This is achieved by appending the captured continuation to the front of the current continuation.

Observe that `continue` and `discontinue` are not part of the expressions. They are encoded as continue $k\ e = (k\ (\lambda^o x.x))\ e$ and discontinue $k\ l\ e = (k\ (\lambda^o x.\textbf{raise}\ l\ x))\ e$. Intuitively, resuming a continuation in both the cases involves evaluating the appropriate abstraction on top of the continuation. We perform the administrative reductions Resume1 and Resume2 to evaluate the arguments to `continue` and `discontinue`. The rule Resume appends the given continuation to the front of the current continuation, and evaluates the body of the closure.

5 Implementation

We now present the implementation details of effect handlers in Multicore OCaml. While we assume an x86_64 architecture for the remainder of this paper, our design does not preclude other architectures and operating systems.

5.1 Exceptions

The implementation follows the operational semantics, but has a few key representational differences. Unlike the operational semantics, handlers with just exception patterns

(exception handlers) are implemented differently than effect handlers. As mentioned in §2.2, exceptions are pervasive in OCaml and are so cheap that they are used for local control flow. Hence, we retain the linked exception handler frame implementation of stock OCaml in Multicore OCaml to ensure performance backwards compatibility. This differs from other research languages with effect handlers [3, 11, 26], which implement exceptions using effects (by ignoring the continuation argument in the handler).

5.2 Heap-allocated fibers

In the operational semantics, the continuations may be resumed more than once. Captured continuations are copies of the original fibers and resuming the continuation copies the fibers and leaves the continuation as it is. Since our primary use case is concurrency, continuations will be resumed at most once, and copying fibers is unnecessary and inefficient. Instead, Multicore OCaml optimises fibers for one-shot continuations. Fibers are allocated on the C heap using `malloc` and are `free`d when the handled computation returns with a value or an exception. Similar to Farvardin et al. [20], we use a *stack cache* of recently freed stacks in order to speed up allocation.

Figure 3a shows the layout of a fiber in Multicore OCaml. At the bottom of the stack, we have the `handler_info`, which contains the pointer to the parent fiber, and the closures for the value, exception and effect cases. The closures are created by the translation described in §4.1; Multicore OCaml supports exception patterns in addition to effect patterns in the same handler. This is followed by a context block needed for DWARF and GC bookkeeping with callbacks. Then, there is a top-level exception handler frame that forwards exceptions to the parent fiber. When the exceptions are caught by this handler, the control switches to the parent stack, and the exception handler closure `clos_hexn` is invoked. This is followed by the `pc` of the code that returns values to the parent fiber. This stack is laid out such that when the handled computation returns, the control switches to the parent fiber and the value handler `clos_hval` is invoked.

Next, we have the variable-sized area for the OCaml frames. In order to keep fibers small, this area is initially 16 words in length. When the stack pointer `rsp` becomes less than the *stack threshold* (maintained in the `Caml_state` table), the stack is said to have overflowed. On stack overflow, we copy the whole fiber to a new area with double the size. In Multicore OCaml, we introduce stack overflow checks into the function prologue of OCaml functions. These stack overflows are rare and so the overflow checks will be correctly predicted by the CPU branch predictor.

In our evaluation of real world OCaml programs (§6), we observed that most function calls are to leaf functions with small frame sizes. Can we eliminate the stack overflow checks for these functions? To this end, we introduce a small, fixed-sized red zone at the top of the stack. The compiler elides the stack overflow check for leaf functions whose frame size is less than the size of the red zone. The default size of the red zone in Multicore OCaml is 16 words.

Finally, we have the saved exception pointer, which points to the top-most exception frame, and the saved stack pointer, which points to the top of the stack. Switching between fibers only involves saving the exception and the stack pointer of the current stack and loading the same on the target stack. Since OCaml does not generate pointers into the stack, the two `fiber_info` fields are the only ones that need to be updated when fibers are moved.

5.3 External calls and callbacks

Since C functions do not have stack overflow checks, we have to execute the external calls in the system stack. Calling a C function from OCaml involves saving the stack pointer in the current fiber, saving the allocation pointer value in `r15` in the `Caml_state`, updating `rsp` to the top of system stack (maintained in `Caml_state`), and calling the C function. The actions are reversed when returning from the external call. For C functions that take arguments on the stack, the arguments must be copied to the C stack from the OCaml stack.

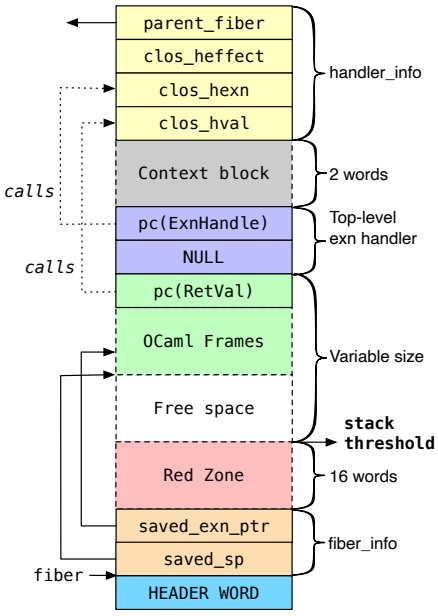
When we first enter OCaml from C, a new fiber is allocated for the main OCaml stack. Since callbacks may be frequent in OCaml programs that use finalisers, we run the callbacks on the same fiber as the current one. For example, the layout of the Multicore OCaml stack at `caml_raise_exn` in the `meander` example from §2 is shown in Figure 3b. The functions `caml_call_c` and `caml_call_ocaml` switch the stacks, and hence are shown in both the system stack and the fiber. Since we are reusing the fiber for the callback, care must be taken to save and restore the `handler_info` before calling and after returning from `c_to_ocaml` function, respectively. Thanks to the fiber representation, external calls and callbacks remain competitive with stock OCaml.

5.4 Effect handlers

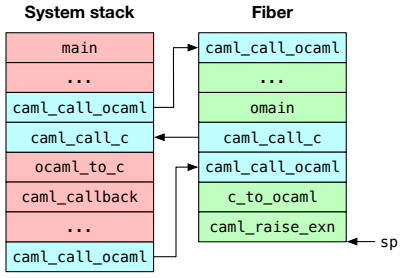
Similar to exception handlers, the lack of callee-saved registers in OCaml benefits effect handlers. There is no register state to save when entering an effect handler or performing an effect. Similarly, there is no register state to restore when handling an effect or resuming a continuation. This fortuitous design choice in stock OCaml has a significant impact in enabling fast switching between fibers in Multicore OCaml.

In order to illustrate the runtime support for handling effects, consider the example presented in Figure 3c. The layout of the program state as the program executes is captured in Figure 3d. The code performs effect `E` which is handled in the outer-most handler, and is immediately resumed. The arrows between the fibers are parent pointers. At position `p1`, `rsp` is at the top of the fiber `f`.

When the effect `E` is performed, we allocate a continuation object `ke` in the OCaml heap that points to the current fiber



(a) Fiber layout

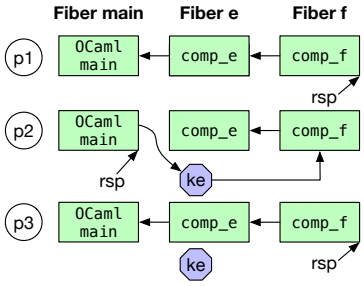


(b) Stack layout for `meander` example from §2.

```
effect E : unit
effect F : unit;;

match (* comp_e *)
  match (* comp_f *)
    (*p1*) perform E (*p3*)
  with | v -> v | effect F kf -> ()
with | v -> v
     | effect E ke -> (*p2*) continue ke ()
```

(c) Constructing continuation objects



(d) Program state for code in 3c

Figure 3. Layout of Multicore OCaml effect handlers.

`f`, set fiber `f`'s parent pointer to `NULL`, and evaluate the continuation closure `clos_heffect` on the parent fiber `e` with the effect `E` and the continuation `ke` as arguments. Since the first handler does not handle effect `E`, the effect is *reperformed* (§4.1) by appending the fiber `e` to the tail of continuation `ke`, set fiber `e`'s parent pointer to `NULL`, and evaluate the current continuation closure on the parent fiber `main` with `E` and `ke` as arguments, which handles `E` (position `p2`). Thus, *continuations are captured without copying frames*. Since every handler closure is evaluated until a matching one is found, the time taken to handle an effect is linear in the number of handlers. We observed that the handler stack is shallow in real programs.

When the continuation is resumed, we overwrite the value of `ke` to `NULL` to enforce at-most once semantics. Resuming a continuation involves traversing the linked-list of fibers and making the last fiber point to the current fiber. Just as in the operational semantics, the implementation invokes the appropriate closure to either `continue` or `discontinue` the continuation (position `p3`). We perform tail-call optimisation so that resumptions at tail positions do not build up stack.

5.5 Stack unwinding

The challenge with DWARF stack unwinding is to make it aware of the non-contiguous stacks. While the complete details of DWARF stack unwinding is beyond the scope of the paper, it is beneficial to know how DWARF unwind tables are constructed in order to appreciate our solution. We refer the interested reader to Bastian et al. [2] for a good overview of DWARF stack unwinding.

Logically, DWARF call-frame information maintains a large table which records for every machine instruction where the return address and callee-saved registers are stored. To avoid reifying this large table, DWARF directives represent the table using a compact bytecode representation that describes the unwind table as a sequence of edits from the start of the function. In order to compute the call-frame information at any given instruction within a function, the DWARF bytecode from the start of that function must be interpreted on demand. For each function, DWARF maintains a *canonical frame address* (CFA) and is traditionally the stack pointer before entering this function. Hence, on x86-64, where the return address is pushed on the stack on call, the return address is at `CFA - 8`.

Our goal is to compute the CFA of the caller when stacks are switched using the DWARF directives. Recall that stack switching occurs in effect handlers, external calls and callbacks. At the entry to an effect handler block, we insert DWARF bytecode to follow the `parent_fiber` pointer and dereference the `saved_sp` to get the CFA (`saved_sp + 8`). During callbacks into OCaml, we save the current system stack pointer in the `context block` in Figure 3a to identify the CFA in the C stack. DWARF unwinding for external calls is implemented by following a link to the current OCaml stack pointer. With these changes, we get the same backtrace for the `meander` program from §2.3, modulo runtime system functions due to effect handlers. We have verified the correctness of our DWARF directives using the verification tool from Bastian et al. [2].

Despite the correct DWARF unwind information, using DWARF to record call stack information in `perf` only captures

the call stack of the current fiber in Multicore OCaml. Since stack unwinding using DWARF is slow due to bytecode interpretation overhead, `perf` dumps the (user) call stack when sampled [2]. This only includes the frames from the current fiber. This is a limitation of `perf` and not of our stack layout. Bastian et al. [2] report on a technique to pre-compile the unwind table to assembly, which speeds up DWARF-based unwinding by up to 25×. With this technique, `perf` can unwind the stack at sample points rather than dumping the call stack, which would capture the complete backtrace rather than just the current fiber.

5.6 Garbage collection

Recall that OCaml programs are written with the expectation that function calls return exactly once (§3). Consider the scenario when a continuation is never resumed. Since fibers allocate memory for the stack using `malloc`, which are `freed` when the computation returns, not resuming continuations leaks memory. In addition, unresumed continuations may also leak other system resources such as sockets and open file descriptors.

We make a pragmatic trade-off and expect the user code to resume captured continuations *exactly once*. One can use the GC support to free up resources by installing a finaliser that discontinues the continuation and ignores the result:

```
Gc.finalise (fun k ->
  try ignore (discontinue k Unwind) with _ -> () k
```

This frees up both the memory allocated for the fiber stack as well as other system resources, assuming that user code does not handle `Unwind` exception and fails to re-raise it. Since installing a finaliser on every captured continuation introduces significant overhead (§6.3.3), we choose not to do it by default. It is also useful to note that even if the memory for the fiber stack is managed by the GC, we would still need a finalisation mechanism to unwind the stack and release other system resources that may be held by the continuation.

The challenges and the solutions for integrating fibers with the concurrent mark-and-sweep GC of Multicore OCaml have been discussed previously [49].

6 Evaluation

In this section, we evaluate the performance of Multicore OCaml effect handlers against the performance requirements set in §1.1. Multicore OCaml is an extension of the OCaml 4.10.0 compiler with support for shared memory parallelism and effect handlers. Since our objective is to evaluate the impact of effect handlers, none of our benchmarks utilise parallelism. These results were obtained on a 2-socket Intel®Xeon®Gold 5120 x86-64 [28] server running Ubuntu 18.04 with 64GB of main memory.

6.1 No effects benchmarks

In this section, we measure the impact of the addition of effect handlers on code that does not use effect handlers. Our macro benchmark suite consists of 54 real OCaml workloads including verification tools (`Coq`, `Cubicle`, `AltErgo`), parsers (`menhir`, `yojson`), storage engines (`irmin`), utilities (`cpdf`, `decompress`), bioinformatics (`fasta`, `knucleotide`, `revcomp2`, `regexredux2`), numerical analysis (`grammatrix`, `LU_decomposition`) and simulations (`nbody`, `game_of_life`). In addition to Stock (`stock`) and Multicore OCaml (`MC`), we also ran the benchmarks on Multicore OCaml with no red zone (`MC+RedZone0`) in the fibers (all OCaml functions will have a stack overflow check) and a red zone size of 32 words (`MC+RedZone32`). Recall that the default red zone size in Multicore OCaml is 16 words (§5.2).

Table 1. Micro benchmarks without effects. Each entry is the percentage difference for Multicore OCaml over stock OCaml.

	exnval	*exnraise*	*extcall*	*callback*	*ack*	*fib*	*motzkin*	*sudan*	*tak*
Time	+0.0	-1.9	+17	+65	+5.3	+2.2	+10	+0.0	+4.2
Instr	+0.0	+0.0	+10	+72	+16	+24	+16	+14	+17

Figure 4 presents the running time of the different multicore variants normalized against the sequential baseline `stock`. On average (geometric mean of the normalized values against `stock` as the baseline), the multicore variants were less than 1% slower than `stock`. The outliers (on either ends) were due to the difference in the allocator and the GC between stock and Multicore OCaml. Of the 54 programs in the benchmark suite, 32 programs had an overhead of 5% or lower, and 8 programs had more than 10% overhead.

The biggest impact was the increase in the OCaml text section size (OTSS) due to the stack overflow checks. We define OTSS as the sum of the sizes of all the OCaml text sections in the compiled binary file ignoring the data sections, the debug symbols, the text sections associated with OCaml runtime and other statically linked C libraries. Figure 5 presents the OTSS of the multicore variants normalized against the sequential baseline `stock`. Compared to `stock`, OTSS is 19% more for `MC` and `MC+RedZone32`, and 30% more for `MC+RedZone0`. The result shows that our 16-word red zone is effective at reducing OTSS compared to having no red zone, whereas the 32-word red zone does not noticeably reduce OTSS further. Further work is required to bring OTSS closer to `stock`.

We also present micro benchmarks results in Table 1. Since micro benchmarks magnify micro-architectural optimisations, we also report the number of instructions executed (obtained using `perf`) along with time. `exnval` performs 100 million iterations of installing exception handlers and returning with a value. `exnraise` is similar, but raises an exception in each iteration. `extcall` and `callback` perform 100 million external calls and callbacks to identity functions. The other micro benchmarks are highly recursive programs and were taken from Farvardin et al. [20]. For micro benchmarks, we observed that padding tight loops with a few `nop` instructions, which changes the loop alignment, makes the code up to

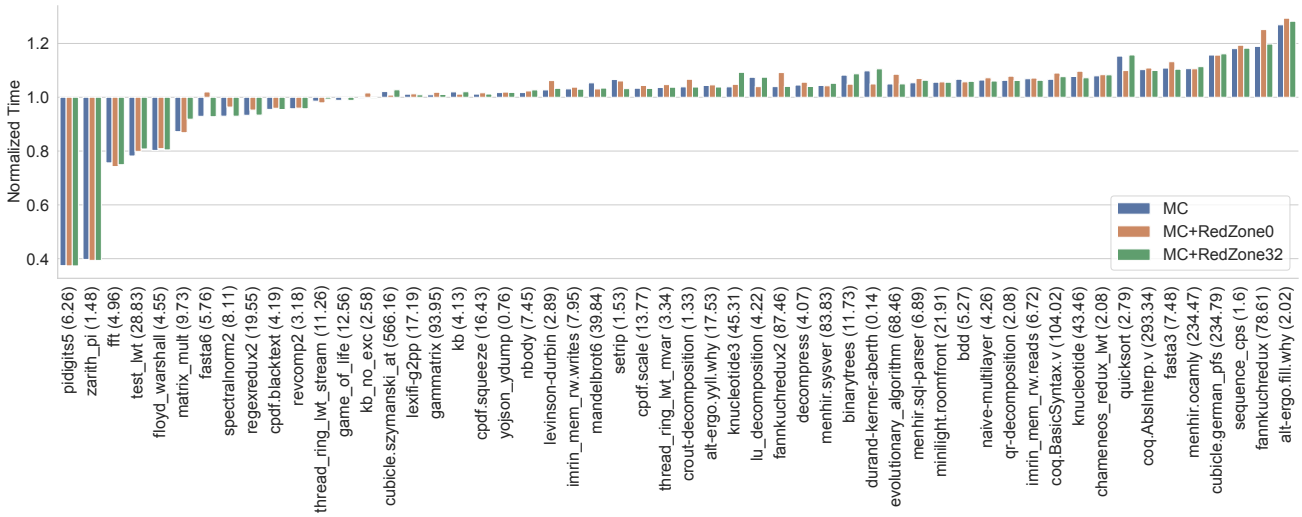


Figure 4. Normalized time of macro benchmarks. Baseline is Stock OCaml, whose running time in seconds in given in parenthesis.

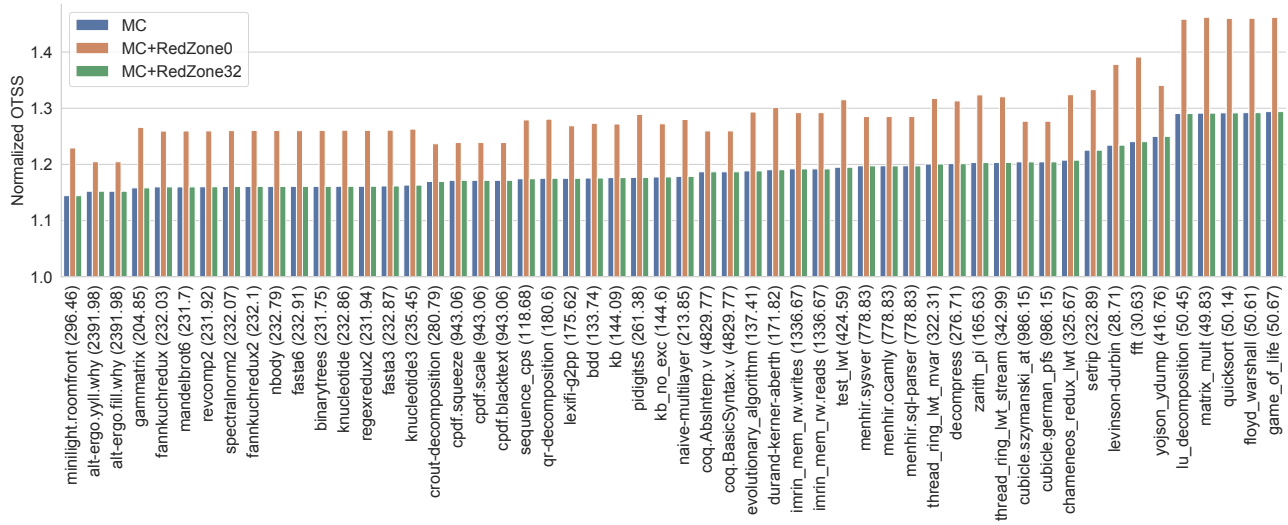


Figure 5. Normalized OCaml text section size (OTSS) of macro benchmarks. Baseline is Stock OCaml, whose size in kilobytes given in parenthesis.

15% faster. Hence, the difference in running times under 15% may not be statistically significant.

The results show that exceptions are no more expensive in `MC` compared to `stock`. In the other programs, `MC` executes more instructions due to stack overflow checks. The performance impact on callbacks is more significant than external calls. For callbacks, since we reuse the current fiber stack, we need to ensure it has enough room for inserting additional frames, while `stock` does not need to do this. Callback performance is less important than external calls, which are far more numerous.

6.2 No perform benchmarks

Next, we aim to quantify the overhead of setting up and tearing down effect handlers compared to a non-tail function call. To this end, we surround the non-tail calls in the recursive micro benchmarks with an effect handler. These programs do not `perform` effects. We also implemented the same benchmarks using a concurrency monad [9] (`monad`) as a proxy for CPS versions. Recall that the OCaml compiler does not use CPS in its IR. In the `monad` version, we use a `fork` to invoke the non-tail call and use an `MVar` to collect its result.

Table 2. Micro benchmarks with handlers but no perform. Each entry is the slowdown factor (× times) over its idiomatic implementation in stock OCaml.

	ack	fib	motzkin	sudan	tak
MC	12.25	12.05	11.44	6.74	8.9
monad	348.69	69.77	39.24	33.29	42.79

The results are presented in Table 2. They show that using effect handlers (concurrency monad) is 10.02× (67.09×) more expensive than the idiomatic implementation using non-tail

calls. The concurrency monad suffers due to the heap allocation of continuation frames (which need to be garbage collected), whereas effect handlers benefit from stack allocation of the frames. For example, the number of major collections for the `ack` benchmark is 0 for stock OCaml, 1 for `MC` and 112 for `monad`. Our concurrency monad (and other monadic concurrency libraries such as Lwt [53] and Async [40, §18]) also have other downsides – exceptions, backtraces, and DWARF unwinding are no longer useful due to the lack of a stack.

We note that a compiler that uses CPS IR will be faster than the concurrency monad implementation due to optimisations to reduce the heap allocation of continuation frames. But Farvardin et al. [20] show that CPS with optimisations is still slower than using the call stack.

6.3 Concurrent benchmarks

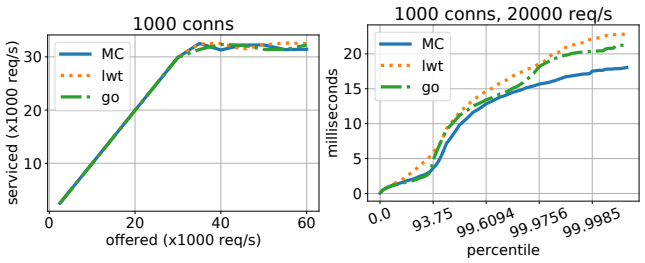
Next we look at benchmarks that utilize non-local control flow using effect handlers. First, we quantify the cost of individual operations in effect handling. Consider the following annotated code:

```
effect E : unit
(* a *) match (* b *) perform E (* d *) with
| v -> (* e *) v
| effect E k -> (* c *) continue k ()
```

The sequence `a-b` involves allocating a new fiber and switching to it. `b-c` is performing the effect and handling it. `c-d` is resuming the continuation. `d-e` is returning from the fiber with a value and freeing the fiber. We measured the time taken to execute these sequences using `perf` support for cycle-accurate tracing on modern Intel processors. We executed 10 iterations of the code, with 3 warm-up runs. For calibration, the idle memory load latency for the local NUMA node is 93.2 ns as measured using the Intel MLC tool [41]. We observed that the sequences `a-b`, `b-c`, `c-d` and `d-e` took **23 ns, 5 ns, 11 ns and 7 ns**, respectively. The time in the sequence `a-b` is dominated by the memory allocation. Thus, the individual operations in effect handling are fast.

6.3.1 Generators. Generators allow data structures to be traversed on demand. Many languages including JavaScript and Python provide generators as a built-in primitive. Using effect handlers (`MC`), given any data structure (`'a t`) and its iterator (`val iter: 'a t -> ('a -> unit) -> unit`), we can derive its generator function (`val next : unit -> 'a option`)[2]. We evaluate the performance of traversing a complete binary tree of depth 25 using this generator. This involves 2^{26} stack switches in total. For comparison, we implemented a hand-written, selective CPSed [43], defunctionalised [15] version (`cps`) and a concurrency monad (`monad`) version of the generator for the tree. Both `cps` and `monad` versions are specialised to the binary tree with the usual caveats of not using the stack for function calls. We observed that the `cps` version was the fastest, thanks to specialisation and hand optimisation. `MC` version was only 2.76× slower than `cps` while being a generic solution, and the `monad` version was 8.69× slower than `cps`.

[2]https://gist.github.com/kayceesrk/eb0ab496c22861f21b1d9484772e982d



(a) Throughput **(b)** Tail latency

Figure 6. Web server performance.

6.3.2 Chameneos. Chameneos [29] is a concurrency game aimed at measuring context switching and synchronization overheads. Our implementation uses `MVars` for synchronization. We compare effect handler (`MC`), concurrency monad (`monad`) and Lwt, a widely used concurrency programming library for OCaml (`lwt`) versions. We observed that `MC` was the fastest, and `monad` (`lwt`) was 1.67× (4.29×) slower than `MC`.

6.3.3 Finalised continuations. In §5.6, we described how continuation resources can be cleaned up by attaching a finaliser. Attaching this finaliser to every captured continuation slows down generator (chameneos) benchmark by 4.1× (2.1×) compared to not attaching a finaliser. Hence, Multicore OCaml does not attach such finaliser to every continuation by default.

6.3.4 Webserver. Using effect handlers, we have implemented a full-fledged HTTP/1.1 web server by extending the example from §3.1 (`MC`). The web server spawns a lightweight thread per request. We use `httpaf` [27] for HTTP handling, and `libev` [36] for the eventloop. We compare our implementation against an Lwt version (`lwt`) which also uses `httpaf` and `libev`. Unlike using effect handlers, the Lwt version is written in monadic style and does not have the notion of a thread per request. For comparison, we include a Go 1.13 version (`go`) that uses the `net/http` [42] package. As both the OCaml versions are single threaded, the Go benchmark is run with `GOMAXPROCS=1`.

The client workload was generated with `wrk2` [55]. We maintain 1k open connections and perform requests for a static web page at different rates, and record the service rate and latency. The throughput and tail latency graphs are given in Figures 6a and 6b. In all the versions, the throughput plateaus at around 30k requests per second. We measure the tail latencies at 2/3rd of this rate (20k requests per second) to simulate optimal load. We observe that both of the OCaml versions remain competitive with `go`, and `MC` performs best in terms of tail latency.

Multicore OCaml supports backtraces for continuations in addition to backtraces of the current stack as in stock OCaml. Using effect handlers in a system such as a web server

aids debugging and profiling because it is possible to get a backtrace snapshot of all current requests. This feature is available in Go [24], but not in OCaml concurrency libraries such as Async and Lwt which lack the notion of a thread.

7 Related Work

There are several strategies for implementing effect handlers. Eff [3], Helium [6], Frank [11] and the Links server backend [26] use an interpreter similar to our operational semantics to implement effect handlers. Effekt [47], Links JavaScript backend [26] and Koka [33] use type-directed selective CPS translation. These language are equipped with an effect system, which allows compiling pure code in direct style and effectful code in CPS. Leijen [34] implements effect handlers as a library in C using stack copying. C allows pointers into the stack, so care is taken to ensure that when continuations are resumed, the constituent frames are restored to the same memory addresses as at the time of capture. Kiselyov et al. [32] use an implementation of multi-prompt delimited continuations as an OCaml library [31] to embed the Eff language in OCaml. Indeed, Forster et al. [23] showed that in an untyped setting, effect handlers, monadic reflection and delimited control can macro-express each other.

Multicore OCaml uses the call stack for implementing continuations (as do [31, 34]), but with one-shot continuations. Bruggeman et al. [8] show how to implement one-shot continuations efficiently using segmented stacks in Scheme. Farvardin et al. [20] perform a comprehensive evaluation of various implementation strategies for continuations on modern hardware. Multicore OCaml stacks do not neatly fit the description of one of these implementation strategies – they are best described as using the resize strategy from Farvardin et al. for each of the fibers, which are linked to represent the current stack and the captured continuations. Kawahara et al. [30] implement one-shot effect handlers using coroutines as a macro-expressible translation, and present an embedding in Lua and Ruby. Lua provides asymmetric coroutines [16] where each coroutine uses its own stack similar to how each handled computation runs in its own fiber in Multicore OCaml.

Multicore OCaml is not the first language to support stack inspection in the presence of non-local control operators. Chez Scheme supports continuation marks [22] which permit stack inspection as a language feature. This enables implementation of dynamic binding, exceptions, profilers, debuggers, etc, in the presence of first-class continuations. As the authors note, continuation marks can be implemented using effect handlers, but direct support for continuation marks leads to better performance. In this work, we focus on retaining the support for stack inspection through DWARF unwind tables in the presence of effect handlers.

The interaction of non-local control flow and resources has been studied extensively. Scheme uses `dynamic-wind` [46], which is a generalisation of Common Lisp `unwind-protect` [51], which ensures de-allocation and re-allocation of resources every time the non-local control leaves and enters back into a context. `dynamic-wind` is not quite the right abstraction as resources need to be cleaned up only on non-returning exits [19, 48]. This requires distinguishing returning exits from non-returning ones.

Multicore OCaml builds on the existing defensive coding practices against exceptions to clean up resources on non-returning exits. We assume that the continuations are resumed exactly once using `continue` or `discontinue`. Under this assumption, when a computation performs an effect, we expect the control to return. For the non-returning cases (value and exceptional return), the code already handles resource cleanup.

OCaml does not have a `try/finally` construct commonly used for resource cleanup in many programming languages. The OCaml standard library [50] as well as alternative standard libraries such as Base [1] and Core [13] provide mechanisms analogous to `unwind-protect`, which are in turn implemented using exception handlers. Thus, the linear use of continuations enabled by the `discontinue` primitive ensures backwards compatibility of legacy OCaml systems code under non-local control flow introduced by effect handlers.

Leijen [35] explicitly extends effect handlers with `initially` and `finally` clauses in Koka for resource safety. Dolan et al. [17] describe the interaction of effect handlers and asynchronous exceptions. This is orthogonal to the contributions of this paper. Our focus is the compiler and runtime system support for implementing effect handlers.

8 Conclusions

Our design for effect handlers in OCaml walks the tightrope of maintaining compatibility (for profiling, debuggers and minimal overheads for existing programs), while unlocking the full power of non-local control flow constructs. Our evaluation shows that we have achieved our goal: we retain compatibility with a surprisingly low performance overhead for sequential code that preserves the spirit of "fast exceptions" that has always characterised OCaml programming. We believe that the introduction of effect handlers into OCaml implemented using lightweight fibers, along with a parallel runtime [49], as has been demonstrated in our work, will open OCaml to highly scalable concurrent and task-parallel applications with minimal hit to sequential performance.

Acknowledgements

We thank Sam Lindley, Francóis Pottier, the PLDI reviewers and our shepherd, Matthew Flatt, whose comments substantially helped improve the presentation. This research was funded via Royal Commission for the Exhibition of 1851 and Darwin College Research Fellowships, and by grants from Jane Street and the Tezos Foundation.

References

[1] Base.Exn.protect 2020. *Unwind-protect in JaneStreet Base library.* https://ocaml.janestreet.com/ocaml-core/v0.13/doc/base/Base/Exn/index.html#val-protectx

[2] Théophile Bastian, Stephen Kell, and Francesco Zappa Nardelli. 2019. Reliable and Fast DWARF-Based Stack Unwinding. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 146 (Oct. 2019), 24 pages. https://doi.org/10.1145/3360572

[3] 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.

[4] Karthikeyan Bhargavan, Barry Bond, Antoine Delignat-Lavaud, Cédric Fournet, Chris Hawblitzel, Catalin Hritcu, Samin Ishtiaq, Markulf Kohlweiss, Rustan Leino, Jay Lorch, Kenji Maillard, Jianyang Pang, Bryan Parno, Jonathan Protzenko, Tahina Ramananandro, Ashay Rane, Aseem Rastogi, Nikhil Swamy, Laure Thompson, Peng Wang, Santiago Zanella-Béguelin, and Jean-Karim Zinzindohoué. 2017. Everest: Towards a Verified, Drop-in Replacement of HTTPS. In *2nd Summit on Advances in Programming Languages.* http://drops.dagstuhl.de/opus/volltexte/2017/7119/pdf/LIPIcs-SNAPL-2017-1.pdf

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

[6] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2020. Binders by Day, Labels by Night: Effect Instances via Lexically Scoped Handlers. *Proc. ACM Program. Lang.* 4, POPL, Article 48 (Dec. 2020), 29 pages. https://doi.org/10.1145/3371116

[7] Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D. Goodman. 2018. Pyro: Deep Universal Probabilistic Programming. arXiv:1810.09538 [cs.LG]

[8] Carl Bruggeman, Oscar Waddell, and R. Kent Dybvig. 1996. Representing Control in the Presence of One-Shot Continuations. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation* (Philadelphia, Pennsylvania, USA) *(PLDI '96).* Association for Computing Machinery, New York, NY, USA, 99–107. https://doi.org/10.1145/231379.231395

[9] Koen Claessen. 1999. A Poor Man's Concurrency Monad. *J. Funct. Program.* 9, 3 (May 1999), 313–323. https://doi.org/10.1017/S0956796899003342

[10] Colour 2020. *What Color is Your Function?* http://journal.stuffwithstuff.com/2015/02/01/what-color-is-your-function/

[11] Lukas Convent, Sam Lindley, Conor McBride, and Craig McLaughlin. 2020. Doo bee doo bee doo. *Journal of Functional Programming* 30 (2020), e9. https://doi.org/10.1017/S0956796820000039

[12] Coq 2020. *The Coq Proof Assistant.* https://coq.inria.fr/

[13] Core.Exn.protect 2020. *Unwind-protect in JaneStreet Core library.* https://ocaml.janestreet.com/ocaml-core/109.20.00/doc/core/Exn.html

[14] Olivier Danvy and Andrzej Filinski. 1990. Abstracting Control. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming* (Nice, France) *(LFP '90).* Association for Computing Machinery, New York, NY, USA, 151–160. https://doi.org/10.1145/91556.91622

[15] Olivier Danvy and Lasse R. Nielsen. 2001. Defunctionalization at Work. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming* (Florence, Italy) *(PPDP '01).* Association for Computing Machinery, New York, NY, USA, 162–174. https://doi.org/10.1145/773184.773202

[16] Ana Lúcia de Moura, Noemi Rodriguez, and Roberto Ierusalimschy. 2004. Coroutines in Lua. *j-jucs* 10, 7 (jul 2004), 910–925. http://www.jucs.org/jucs_10_7/coroutines_in_lua

[17] 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.

[18] DWARF 2020. *The DWARF Debugging Standard.* http://dwarfstd.org/

[19] Dynamic Wind 2020. *The dynamic-wind problem.* http://okmij.org/ftp/continuations/against-callcc.html#dynamic_wind

[20] Kavon Farvardin and John Reppy. 2020. From Folklore to Fact: Comparing Implementations of Stacks and Continuations. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020).* Association for Computing Machinery, New York, NY, USA, 75–90. https://doi.org/10.1145/3385412.3385994

[21] Matthias Felleisen and Daniel P.Friedman. 1986. *Control Operators, the SECD-Machine, and the Lambda-Calculus.* Technical Report. https://help.luddy.indiana.edu/techreports/TRNNN.cgi?trnum=TR197

[22] Matthew Flatt and R. Kent Dybvig. 2020. Compiler and Runtime Support for Continuation Marks. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020).* Association for Computing Machinery, New York, NY, USA, 45–58. https://doi.org/10.1145/3385412.3385981

[23] Yannick Forster, Ohad Kammar, Sam Lindley, and Matija Pretnar. 2019. On the expressive power of user-defined effects: Effect handlers, monadic reflection, delimited control. *J. Funct. Program.* 29 (2019), e15. https://doi.org/10.1017/S0956796819000121

[24] Go PProf 2020. *Profiling a Go Program.* https://golang.org/pkg/runtime/pprof/#Profile

[25] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the Web up to Speed with WebAssembly. 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, 185–200. https://doi.org/10.1145/3062341.3062363

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

[27] httpaf 2020. *A high performance, memory efficient, and scalable web server written in OCaml.* https://github.com/inhabitedtype/httpaf

[28] Intel Xeon Gold 5120 2020. *Intel® Xeon® Gold 5120 Processor Specification.* https://ark.intel.com/content/www/us/en/ark/products/120474/intel-xeon-gold-5120-processor-19-25m-cache-2-20-ghz.html

[29] C. Kaiser and J. . Pradat-Peyre. 2003. Chameneos, a concurrency game for Java, Ada and others. In *ACS/IEEE International Conference on Computer Systems and Applications, 2003. Book of Abstracts.* 62–. https://doi.org/10.1109/AICCSA.2003.1227495

[30] Satoru Kawahara and Yukiyoshi Kameyama. 2020. One-Shot Algebraic Effects as Coroutines. In *Trends in Functional Programming*, Aleksander Byrski and John Hughes (Eds.). Springer International Publishing, Cham, 159–179.

[31] Oleg Kiselyov. 2010. Delimited Control in OCaml, Abstractly and Concretely: System Description. In *Functional and Logic Programming*, Matthias Blume, Naoki Kobayashi, and Germán Vidal (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 304–320.

[32] Oleg Kiselyov and KC Sivaramakrishnan. 2018. Eff Directly in OCaml. *Electronic Proceedings in Theoretical Computer Science* 285 (Dec 2018), 23–58. https://doi.org/10.4204/eptcs.285.2

[33] Daan Leijen. 2017. Implementing Algebraic Effects in C. In *Asian Symposium on Programming Languages and Systems*, Bor-Yuh Evan Chang (Ed.). Springer International Publishing, Cham, 339–363.

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

[35] Daan Leijen. 2018. *Algebraic Effect Handlers with Resources and Deep Finalization.* Technical Report MSR-TR-2018-10. 35 pages.

[36] libev 2020. *A high performance full-featured event loop written in C.* https://metacpan.org/pod/distribution/EV/libev/ev.pod#NAME

[37] Loom 2020. *Fibers, continuations and tail-calls for the JVM.* https://openjdk.java.net/projects/loom/

[38] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. Unikernels: Library Operating Systems for the Cloud. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Houston, Texas, USA) *(ASPLOS '13)*. Association for Computing Machinery, New York, NY, USA, 461–472. https://doi.org/10.1145/2451116.2451167

[39] Laurent Mauborgne. 2004. Astrée: Verification of Absence of Runtime Error. In *Building the Information Society*, Renè Jacquart (Ed.). Springer US, Boston, MA, 385–392.

[40] Yaron Minsky, Anil Madhavapeddy, and Jason Hickey. 2013. *Real World OCaml: Functional Programming for the Masses.* O'Reilly. https://realworldocaml.org

[41] MLC 2020. *Intel Memory Latency Checker v3.9.* https://software.intel.com/content/www/us/en/develop/articles/intelr-memory-latency-checker.html

[42] net/http 2020. *HTTP client and server implementations in Go.* https://golang.org/pkg/net/http/

[43] Lasse R. Nielsen. 2001. A Selective CPS Transformation. *Electronic Notes in Theoretical Computer Science* 45 (2001), 311 – 331. https://doi.org/10.1016/S1571-0661(04)80969-1 MFPS 2001,Seventeenth Conference on the Mathematical Foundations of Programming Semantics.

[44] OCaml Manual 2020. *Extensible variant types.* https://caml.inria.fr/pub/docs/manual-ocaml/extensiblevariants.html

[45] 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.

[46] 1998. Revised5 Report on the Algorithmic Language Scheme. *Higher-Order and Symbolic Computation* 11, 1 (Aug. 1998), 7–105. https://doi.org/10.1023/A:1010051815785

[47] Philipp Schuster, Jonathan Immanuel Brachthäuser, and Klaus Ostermann. 2020. Compiling Effect Handlers in Capability-Passing Style. *Proc. ACM Program. Lang.* 4, ICFP, Article 93 (Aug. 2020), 28 pages. https://doi.org/10.1145/3408975

[48] Dorai Sitaram. 2003. Unwind-protect in portable Scheme. In *Proceedings of the 4th Workshop on Scheme and Functional Programming (7 Nov. 2003), M. Flatt, Ed., no. UUCS-03-023 in Tech. Rep., School of Computing, University of Utah.* 48–52.

[49] 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

[50] Stdlib.Fun.protect 2020. *Unwind-protect in the OCaml 4.10.0 standard library.* https://caml.inria.fr/pub/docs/manual-ocaml/libref/Fun.html#exception

[51] Guy L. Steele. 1990. *Common LISP: The Language (2nd Ed.).* Digital Press, USA. https://www.cs.cmu.edu/Groups/AI/html/cltl/cltl2.html

[52] Swift 2020. *Swift Concurrency Roadmap.* https://forums.swift.org/t/swift-concurrency-roadmap/41611

[53] 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

[54] Wasm Effect Handlers 2020. *Typed continuations to model stacks.* https://github.com/WebAssembly/design/issues/1359

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