High-level effect handlers for C

MARIO ALVAREZ-PICALLO, Huawei Research Centre, United Kingdom
TEODORO FREUND, Huawei Research Centre, United Kingdom
DAN R. GHICA, Huawei Research Centre, United Kingdom
SAM LINDLEY, The University of Edinburgh, United Kingdom

Effect handlers provide a structured way to define and use custom computational effects, ranging from exceptions to generators to lightweight threads. We introduce **libseff**, a library that brings the power of effect handlers to C. In contrast to other existing effect handler libraries, **libseff** is designed to be used directly from C by C programmers writing idiomatic direct-style code. Through a series of examples and benchmarks we demonstrate the expressiveness that effect handlers can bring without sacrificing readability or performance.

1 INTRODUCTION

Effect handler oriented programming languages and libraries empower programmers to define custom effectful operations whose semantics is specified later by a suitable *effect handler* [23]. The power of handlers lies in their ability to support fine-grained customisation (a given effectful computation can be handled by different handlers that give it different behaviours, such as implementing a different scheduling strategy), and their composability (handlers can be composed to allow using multiple different effects in the same program).

A central aspect of effect handlers is that when handling an operation they are provided with an explicit representation of the *continuation* of the code that performed the operation (that is the rest of the computation from that point up to the point at which the handler was installed). A continuation is a first-class object that can be resumed immediately, aborted entirely, or delayed for later execution. In this sense, effect handlers can be seen as providing a form of first-class resumable exceptions, and allow for the implementation of sophisticated forms of control flow, such as async/await, exceptions, generators and varied forms of lightweight concurrency, entirely as user-defined libraries.

Though effect handlers are often deployed in the context of high-level functional programming languages such as OCaml [25], we believe that lower-level languages also stand to gain much from such features. Indeed, if one enumerates all of the features that are enabled by the introduction of effect handlers, the only language in common use today to lack all of these is C. On the other hand, the C ecosystem is rife with ad-hoc implementations of complex control-flow operators that are intended to support exactly these features, often on a per-project basis.

There already exist two C effect handler libraries, **libhandler** [16] and **libmpeff** [17]. However, they are both geared towards compiler writers, being more suited as compilation targets for high-level languages with effects, rather than being used directly from C by C programmers.

In contrast, in this paper we introduce and evaluate **libseff**, a new effect handler library designed to be used as part of a C codebase to write efficient code that looks and feels as much as possible like idiomatic C.

The **libseff** library differs from prior approaches in several respects:

- Unlike **libhandler** which relies on stack-copying (unsafe in C as there may be pointers into the stack) and **libmpeff** which relies on virtual memory (not feasible for embedded

Authors' addresses: Mario Alvarez-Picallo, Huawei Research Centre, Edinburgh, United Kingdom, mario.alvarez.picallo@huawei.com; Teodoro Freund, Huawei Research Centre, Edinburgh, United Kingdom, teodoro.freund@huawei.com; Dan R. Ghica, Huawei Research Centre, Edinburgh, United Kingdom, dan.ghica@huawei.com; Sam Lindley, The University of Edinburgh, Edinburgh, United Kingdom, sam.lindley@ed.ac.uk.

systems), **libseff** implements stack resizing via segmented stacks. Stack resizing is often important for applications such as web servers that spawn many lightweight threads, each of which needs its own stack.
- Unlike traditional effect handler implementations **libseff** is oriented around mutable coroutine objects rather than immutable continuation objects. This offers a simple way of avoiding allocating a new continuation object every time an effectful operation (such as yielding to another thread) is performed. Moreover, it provides a more familiar interface for C programmers, who may treat **libseff** like a conventional coroutine library and integrate the effectful features as necessary.
- Unlike traditional effect handler implementations there is no special mechanism for dispatching on effects. Instead performed effects are reified as request objects which are then typically dispatched on using a standard `switch` statement.

The main contributions of this paper are the following:

- The design of **libseff**, illustrated through a series of examples that introduce techniques for programming with effects and handlers in C using **libseff** (§2).
- The implementation of **libseff**, including a description which details the runtime representation, low-level primitives, and stack-management strategy (§3).
- A quantitative evaluation of **libseff** through a series of benchmarks, which shows that **libseff** enjoys competitive performance with other systems (§4).
- A qualitative comparison between **libseff** and **libmpeff** from the point of view of a C programmer using both libraries (§5).

§6 discusses related work and §7 concludes and outlines planned improvements for **libseff**.

The supplementary material includes an appendix with a formal calculus and abstract machine that specifies the semantics of the variant of effect handlers underlying the design of **libseff**.

2 DESIGN

We introduce **libseff** and motivate its design by way of a series of examples that highlight the features and common idioms of the library.

2.1 Mutable state

To illustrate the core features of the library we begin with mutable state as a simple, albeit somewhat artificial (C has built-in support for mutable state), example. The following code declares two new *effects* for reading and writing an integer state value.

```
1 DEFINE_EFFECT(get, 0, int64_t, {});
2 DEFINE_EFFECT(put, 1, void, { int64_t new_value; });
```

In order to define an effect we use the macro `DEFINE_EFFECT(name, tag, ret_ty, { param_decls... })`, which takes an effect name (`name`), a tag (`tag`), a return type (`ret_ty`), and a possibly empty collection of parameter declarations (`param_decls`). The snippet above declares effect `get`, which returns a value of type `int64_t` and takes no parameters, and an effect `put`, which does not return a value and takes a single parameter `new_value` of type `int64_t`. At this stage these effects have type signatures, but no implementation. Together they can be thought of as providing an interface to integer state.

Tags. As C macros provide no mechanism for generating fresh numeric tags, we require the user to manually provide a tag for each defined effect. It is the responsibility of the user to ensure that no two effects are assigned the same tag. In fact, different effects with identical tags may be used safely, provided that no code performs one effect in the scope of a handler for another effect that is assigned the same tag. Due to **libseff**'s use of 64-bit wide bitsets to represent handled effects,

only numbers 0-63 may be used as effect tags. This limitation can easily be relaxed at the cost of increasing the overhead of performing an effect, but for the examples we have considered so far this has not proven necessary.

Terminology. More properly, `get` and `put` are *effect operations* and conceptually we might group them together to form an interface for a single integer state *effect*. However, as in OCaml 5 [25] **libseff** does not explicitly group such operations, and we refer to each individual effect operation as an *effect*. Elsewhere effect operations are sometimes referred to as *commands* [1, 7].

The following code uses the `get` and `put` effects to implement a countdown loop.

```
5  void* counter(void* parameter) {
6      int64_t counter;
7      do {
8          counter = PERFORM(get);
9          printf("Counter is %ld\n", counter);
10         PERFORM(put, counter - 1);
11     } while (counter > 0);
12     return NULL;
13 }
```

As C lacks closures and parametric polymorphism, any handled code must be defined inside a top-level function (here `counter`) conforming to the prototype `void* fn(void*)`. In order to perform an effect, we use the `PERFORM(name, {arg...})` macro, which takes an effect name and a possibly empty collection of arguments. This macro provides a convenient wrapper over the lower-level, untyped `seff_perform` primitive which we describe in detail in §3.2. From the perspective of an end-user of **libseff**, an invocation of `PERFORM` looks much like a function call whose parameter and return types match those declared by the corresponding `DEFINE_EFFECT` macro. In particular, the parameter and return types are checked by the C compiler.

If we were to call `counter` directly as a normal function at the top level, then this would result in a runtime error when line 8 is reached as it performs the `get` effect outside the scope of a handler for `get` (analogous to raising an exception outside the scope of an exception handler). The following code illustrates how to handle the effects inside `counter` by instantiating `counter` as a coroutine and then repeatedly resuming the coroutine inside an event loop that handles the performed effects.

```
14 int main(void) {
15     effect_set handles_state = HANDLES(get) | HANDLES(put);
16     seff_coroutine_t *k = seff_coroutine_new(counter, NULL);
17     seff_request_t req = seff_handle(k, NULL, handles_state);
18     int64_t state = 100;
19     bool done = false;
20     while (!done) {
21         switch (req.effect) {
22             CASE_EFFECT(req, get, { req = seff_handle(k, (void *)state, handles_state); break; })
23             CASE_EFFECT(req, put, {
24                 state = payload.new_value; req = seff_handle(k, NULL, handles_state);
25                 break; })
26             CASE_RETURN(req, {
27                 printf("The handled code has finished executing\n"); done = true;
28                 break; })
29         }
30     }
31     seff_coroutine_delete(k);
```

```
32     return 0;
33 }
```

The `handles_state` effect set encapsulates the ability to handle the `get` and `put` effects. The call `seff_coroutine_new(counter, NULL)` allocates a new coroutine object pointed to by `k` which when resumed will run the `counter` function with the argument `NULL`. The call `seff_handle(k, NULL, handle_state)` resumes the coroutine pointed to by `k` and handles the `get` and `put` effects. In fact, it only handles them to the extent that if performed, the coroutine will be suspended and they will be packaged up in the returned request object `req`. The actual handling code appears in the enclosing context, here an event loop which dispatches on `req.effect`. The mutable integer state is stored in the `state` variable. Inside the `switch` statement there is one clause (expressed using the `CASE_EFFECT` macro) for each of the possible effects that the coroutine may perform and a distinguished return clause (expressed using the `CASE_RETURN` macro) for the case where the coroutine returns normally without performing any effects. A `get` effect is handled by resuming the coroutine, passing in the current state (recall that the return type of `get` is `int64_t`). A `put` effect is handled by updating the current state and resuming the coroutine with a `NULL` argument (recall that the return type of `put` is `void`). The special `payload` variable contains the new state passed to the `put` effect. If the coroutine returns without performing an effect then a message is printed and the event loop is exited. Finally the coroutine object is deleted using `seff_coroutine_delete`.

Decoupling effect interception from handling code. Formally, the handler is simply the code that intercepts effects in the given effect set, yielding a corresponding request object. However, it is natural to refer to the code in the surrounding context that dispatches on the request object as a handler and we frequently do so. Conventional effect handlers fuse these two phases together, much like exception handlers, but we opt for a decoupled approach in **libseff** in order to circumvent the awkwardness of encoding a bespoke dispatch mechanism in C.

Function signatures. Type signatures for the three primitive functions seen so far are as follows.

```
seff_coroutine_t *seff_coroutine_new(void *(*fn)(void*), void *arg);
void seff_coroutine_delete(seff_coroutine_t* k);
seff_request_t seff_handle(seff_coroutine_t* k, void* arg, effect_set handled);
```

The API does not differentiate between starting and resuming a coroutine. However, when called on a coroutine for the first time `arg` is ignored (the underlying function has already been applied to an argument supplied to `seff_coroutine_new`), whereas on subsequent calls the continuation of the coroutine is applied to `arg`, which corresponds to the value returned by the effect.

Coroutines as mutable continuations. Traditional accounts of effect handlers do not take coroutines as primitive, but rather *continuations*. A continuation (also sometimes called a resumption) is an immutable object that represents the rest of a computation. A continuation amounts to an immutable `seff_coroutine_t`, but in **libseff** we always manipulate coroutines as pointers to a mutable `seff_coroutine_t` object which is updated in place whenever an effect is handled.

Handlers in **libseff** *are sheep handlers.* Traditional effect handlers are classified as deep or shallow [11]. A deep handler implicitly wraps itself around the continuation of a suspended effect, ensuring that all effects in a computation must be handled uniformly; a shallow handler does not. Following WasmFX [22], handlers in **libseff** are a hybrid sometimes called *sheep* handlers. Sheep handlers are: like shallow handlers in that the original handler need not be installed each time a continuation is resumed; and like deep handlers in that some handler (though not necessarily the original one) must be installed every time a continuation is resumed. In **libseff** this behaviour

manifests as the need to supply an effect set every time we call `seff_handle` on a coroutine, though this effect set may be empty.

2.2 Lightweight concurrency

A much more compelling application of effect handlers, and the central motivation behind the initial development of **`libseff`**, is *lightweight concurrency*. We begin by defining two effects.

```
DEFINE_EFFECT(fork, 0, void, { void *(*fn)(void *); void *arg; });
DEFINE_EFFECT(yield, 1, void, {});
```

The `fork` effect takes a function pointer (`fn`) and an argument to apply it to (`arg`); it spawns a new thread that invokes `fn(arg)`. (In a language with closures we would typically implement `fork` as a one argument effect.) The `yield` effect suspends the current thread, allowing other threads to run at the discretion of the handler.

We write a small example application that initialises a root thread which is responsible for spawning 10 worker threads. These threads then each print 10 messages to the screen.

```
void *root(void *param) {
    for (int64_t i = 1; i <= 10; i++) PERFORM(fork, worker, (void *)(i));
    return NULL;
}
void *worker(void *param) {
    int64_t tag = (int64_t)param;
    for (int64_t iteration = 0; iteration < 10; iteration++) {
        printf("Worker %ld, iteration %ld\n", tag, iteration);
        PERFORM(yield);
    }
    return NULL;
}
```

To run this code, we must define a handler for the `yield` and `fork` effects which amounts to implementing a custom scheduler. The ability of effect handlers to describe APIs to communicate with a scheduler is at the heart of effect handlers' applications to concurrency [4, 5, 7, 22, 25, 29]. For this example, however, we will implement a toy single-threaded scheduler, which we abstract into a function `with_scheduler`.

```
void with_scheduler(seff_coroutine_t *initial_coroutine) {
    effect_set handles_scheduler = HANDLES(yield) | HANDLES(fork);
    tl_queue_t queue;
    tl_queue_init(&queue, 5);
    tl_queue_push(&queue, initial_coroutine);
    while (!tl_queue_empty(&queue)) {
        seff_coroutine_t *next = (seff_coroutine_t *)tl_queue_steal(&queue);
        seff_request_t req = seff_handle(next, NULL, handles_scheduler);
        switch (req.effect) {
            CASE_EFFECT(req, yield, { tl_queue_push(&queue, (struct task_t *)next); break; })
            CASE_EFFECT(req, fork, {
                tl_queue_push(&queue, (struct task_t *)next);
                seff_coroutine_t *new = seff_coroutine_new(payload.fn, payload.arg);
                tl_queue_push(&queue, (struct task_t *)new);
                break; })
            CASE_RETURN(req, { seff_coroutine_delete(next); break; })
        }
    }
```

```
19 }
20 int main(void) { with_scheduler(seff_coroutine_new(root, (void*)0)); return 0; }
```

As in §2.1, the body of the handler is a `switch` statement nested inside a loop. The main difference with the state example is that now a variable number of coroutines are managed simultaneously by the scheduler, and these are stored in the task queue `queue`. On each iteration, the scheduler pops a coroutine off the head of the queue and proceeds to resume it with `seff_handle`. A `fork` or `yield` request is handled by pushing the suspended coroutine to the back of the queue. The `CASE_RETURN` clause is responsible for releasing the coroutine structures as they finish execution.

One-shot continuations. In performance-oriented implementations of effect handlers [22, 25] it is common to restrict continuations to be invoked at most once. This restriction simplifies the runtime system by precluding the duplication of continuations (which would involve creating a copy of the stack frame captured by the continuation). A similar limitation applies in **libseff**, which provides no facilities to copy stack frames. Doing so in C is inherently unsafe, as programmers often manipulate pointers into the stack which would be invalidated if the stack was copied elsewhere. However, in **libseff** there is no way to resume a continuation twice, as continuations per se are not exposed by the API — each time we handle a coroutine its continuation changes. On the other hand, a new kind of bug can occur if a coroutine pointer is copied accidentally (recall that we always refer to coroutines via a `seff_coroutine_t` pointer). For example, in the scheduler code above, if the programmer duplicated line 11 by accident, the coroutine `next` would be enqueued twice. This would not cause an immediate crash, but would lead to surprising behaviour: every time a thread were to yield it would subsequently be scheduled to run twice as often. However, once finished its coroutine object would be deleted and further attempts to dereference the other copy of the pointer in the queue would result in undefined behaviour. It is important with **libseff** for the programmer to take care to manually manage the lifetime of coroutines, but this is quite standard for heap-allocated objects in C.

2.3 Resources

One technique supported by handlers, which we have thus far not shown, is the ability to "delay" a computation to be performed *after* an effect has been handled. This can be done by having the handler explicitly maintain a stack keeping track of all the effects that have been handled so far which is then "unwound" after a coroutine finishes execution. A more elegant approach is to write our handler as a recursive function, rather than a direct imperative loop, and writing additional code after the recursive call.

As a motivating example, we implement scoped resource handling using a single `defer` effect, whose purpose is to schedule a clean-up function `defer_fn` to be called with argument `defer_arg` when the enclosing coroutine ends its execution. We will also define our own variants of resource-allocating primitives (for this example, `malloc` and `fopen`), which immediately perform the `defer` effect to ensure that the corresponding clean-up function is called in a timely fashion.

```
1 DEFINE_EFFECT(defer, 0, void, { void (*defer_fn)(void*); void *defer_arg; });
2 void *malloc_scoped(size_t size) {
3     void *ptr = malloc(size); PERFORM(defer, free, ptr); return ptr;
4 }
5 FILE *fopen_scoped(const char *path, const char *mode) {
6     FILE *f = fopen(path, mode); PERFORM(defer, fclose, f); return f;
7 }
```

These functions may be used as drop-in replacements for `malloc` and `fopen`, the only caveat being that any code that uses them must be run inside a coroutine that handles the `defer` effect.

```
void *uses_resources(void *arg) {
    ... void *ptr1 = malloc_scoped(256); ... void *ptr2 = malloc_scoped(512);
    ... FILE *f = fopen_scoped("example", "r"); ...
}
```

Calling any of these scoped resource acquisition functions will result in the `defer` effect being performed, communicating the need for resource clean-up to any installed handler. One possible implementation for such a handler is given by the recursive function `handle_defer` below.

```
void *handle_defer(seff_coroutine_t *k) {
    seff_request_t req = seff_handle(k, NULL, HANDLES(defer));
    switch (req.effect) {
        CASE_EFFECT(req, defer, {
            void *result = handle_defer(k);
            // Run the clean-up function
            payload.defer_fn(payload.defer_arg);
            return result; })
        CASE_RETURN(req, { return payload.result; })
    }
}
```

Observe that the structure is similar to a recursive version of the event loop of §2.2, with the crucial difference that the recursive call does not take place in tail position; instead, it is followed by a call to the deferred function. At runtime, the call stack of `handle_defer` will match the order in which the different invocations of `defer` were performed, and the corresponding clean-up functions will be called starting from the last.

We abstract away the creation and management of the coroutine object inside a helper function which takes as an argument the function pointer to be run within the scope of the `defer` handler. We can now run `uses_resources` like so:

```
void *run_with_handle_defer(void *(*fn)(void*), void *arg) {
    seff_coroutine_t *k = seff_coroutine_new(fn, arg);
    handle_defer(k);
    seff_coroutine_delete(k);
}
int main(void) { run_with_handle_defer(uses_resources, NULL); }
```

2.4 Composition

An important property of effect handlers is their *composability* [11][9, Chapter 2]. This allows different libraries to define different effects which programmers can then mix within the same function. To illustrate effect handler composition, we use the `defer` effect from the previous section together with a new effect for defining generators. Throughout the rest of this subsection we assume that all the definitions from §2.3 are still in scope.

A generator is a function that yields a stream of multiple values, suspending its execution each time a value is produced and resuming from the same place next time it is invoked. In languages without native support for generators, they can be simulated by a global transformation. With effect handlers we can implement them directly using a single effect.

```
DEFINE_EFFECT(yield_str, 1, void, { char *elt; });
```

In this case, the `yield_str` effect yields a string. As we wish to compose it with `defer` (whose tag is 0) we have taken care to give it the tag 1.

Any function can now be turned into a generator by having it perform the `yield_str` effect. For example, we now define a generator that yields squares up to a certain number, formatted as heap-allocated strings. We use the previously-defined `malloc_scoped` function to reserve memory.

```
void *squares(void *arg) {
    int64_t limit = (size_t)arg;
    for (int64_t i = 0; i < limit; i++) {
        char *str = malloc_scoped(32);
        sprintf(str, "%5lu", i * i);
        PERFORM(yield_str, str);
    }
    return NULL;
}
```

To consume the elements of this generator, we must define a handler for it. A more sophisticated generator library could provide iteration combinators. Here we simply define a `print_all` function that prints each element produced by the generator in sequence.

```
void *print_all(void *arg) {
    seff_coroutine_t *k = seff_coroutine_new(squares, arg);
    while (true) {
        seff_request_t req = seff_handle(k, NULL, HANDLES(yield_str));
        switch (req.effect) {
            CASE_EFFECT(req, yield_str, { puts(payload.elt); break; })
            CASE_RETURN(req, { seff_coroutine_delete(k); return NULL; })
        }
    }
}
```

If we run `print_all` directly, then it crashes on the first call to `malloc_scoped`, as there is no handler for `defer` in scope. Instead, we use the `run_with_handle_defer` combinator from §2.3.

```
int main(void) { run_with_handle_defer(print_all, (void*)50); }
```

This code prints the squares of all integers from 0 to 50, while also ensuring that all of the memory allocated by the underlying generator is freed. Notice that the handlers for the `yield_str` and `defer` are independent — they can be defined in separate modules and combined freely by the programmer.

2.5 Overriding and default handlers

An effect `eff` is always handled by the innermost handler whose effect set includes `eff`. In contrast to function calls, where the callee is determined statically at compile-time, this allows us to redefine the handling of effects at runtime, providing a form of dynamic binding.

Consider a `print` effect for printing strings, along with a function `print_point` that formats a point given by two coordinates and prints it, and an example function that prints two points.

```
DEFINE_EFFECT(print, 0, void, { char *msg; });
void print_point(int64_t x, int64_t y) {
    char buffer[256];
    sprintf(buffer, "{ x: %ld, y: %ld }", x, y);
    PERFORM(print, buffer);
}
void *example(void *arg) { print_point(0, 0); print_point(1, 2); }
```

If `print` was simply a function then the behaviour would be fixed, but because it is an effect we can substitute in different implementation at runtime. We will give two different handlers for `print`.

Our first handler simply prints to standard output. It is a special *default handler* [5].

```
void *default_print(void *print_payload) {
    EFF_PAYLOAD_T(print) payload = *(EFF_PAYLOAD_T(print) *)(print_payload);
    fputs(payload.msg, stdout);
    return NULL;
}
```

A default handler is given by a function of type `void *(*)(void *)` (here `default_print`) which handles a given effect if no other handler is in scope. Default handlers do not interrupt normal control flow, but instead execute as plain functions would, with control returning to the caller code immediately after the body of the handler is executed. Note that the API for default handlers is not type-safe: the payload of the handled effect is passed as a `void` pointer that must be manually cast to the correct type with the `EFF_PAYLOAD_T` macro, which desugars to the payload type of the given effect tag.

Our second handler is a standard (non-default) handler that stores all output in a buffer.

```
void *with_output_to_buffer(char *buffer, void *(*fn)(void*), void *arg) {
    seff_coroutine_t *k = seff_coroutine_new(fn, arg);
    while (true) {
        seff_request_t req = seff_handle(k, NULL, HANDLES(print));
        switch (req.effect) {
            CASE_EFFECT(req, print, {
                strcpy(buffer, payload.msg); buffer += strlen(payload.msg); break; })
            CASE_RETURN(req, { seff_coroutine_delete(k); return payload.result; })
        }
    }
}
```

We install `default_print` as a default handler by calling `seff_set_default_handler` and providing the tag of the effect to be handled. For convenience, **`libseff`** provides the `EFF_ID` macro which expands to the tag (id) of the given effect.

```
int main(void) {
    seff_set_default_handler(EFF_ID(print), default_print);
    example(NULL);
    char buffer[256];
    with_output_to_buffer(buffer, example, NULL);
}
```

After installing the default handler, the direct call to `example` sends the output to the screen. In contrast, the call inside `with_output_to_buffer` sends the output to `buffer`.

3 IMPLEMENTATION

This section provides an overview of the implementation strategy for **`libseff`**, and some of the tradeoffs involved. Unlike other implementations [7, 15, 25] **`libseff`** does not keep a separate stack of handlers, but instead handlers coincide with coroutines: the context that resumed a coroutine becomes the handler for any effects that may be performed within the coroutine. As a coroutine executes, it keeps a pointer to its parent coroutine, creating a runtime configuration where the currently executing coroutine acts as the top of a stack of active coroutines. This list plays a role analogous to the handler stack in other implementations, obviating the bookkeeping and additional allocations involved in keeping track of both continuations and handlers.

3.1 Runtime Representation

During the execution of the program, any effectful computation is instantiated as an object of type `seff_coroutine_t`. This object stores the execution state of the coroutine and its stack frame as well as the set of effects that can be handled from it. In detail, each coroutine object contains:

- A *coroutine state*, which can be one of `RUNNING`, `SUSPENDED` or `FINISHED` and should be understood as preconditions to the **`libseff`** API. A `SUSPENDED` coroutine can be resumed via `seff_handle`. A `RUNNING` (active) coroutine can be suspended. A `FINISHED` coroutine cannot be resumed or suspended. Multiple coroutines can be simultaneously `RUNNING` even in single-threaded applications despite only one of them actually being executed at a given point in time (this can happen when a coroutine spawns and resumes another coroutine, at which both parent and child are `RUNNING`). Similarly, the child of a `SUSPENDED` coroutine can itself be `RUNNING`.
- A set of *handled effects*, that is, the effects that can be handled by suspending this coroutine.
- A pointer to the *parent coroutine*, used to locate the appropriate handler when performing an effect.
- A *resumption* context representing the execution state when the coroutine was last resumed or suspended. When the coroutine `RUNNING`, the resumption context represents the execution state of the context in which it was last resumed, and is used for suspending the coroutine. When the coroutine `SUSPENDED`, it instead represents the execution state of the coroutine at the moment of suspending, and is used for resuming it. The encoding of the resumption context is architecture-dependent. For x86-64 Linux, the only architecture currently supported, it consists of the instruction, stack, and frame pointers, as well as all callee-saved registers according to the standard System V calling convention.
- A *stack* pointer to a region in the heap containing the allocated stack space for the coroutine. As explained in more detail in §3.3, **`libseff`** supports different stack management modes. Depending on the mode, the stack pointer may be a pointer to a fixed-size heap-allocated stack, or to a linked list of heap-allocated "stacklets".

A pointer to the coroutine being currently executed (if any), as well as a pointer to the top of the system stack are also stored in global (more precisely, thread-local) variables. As explained in §3.3.2, this information is used for avoiding allocating larger stack frames when calling library code from a coroutine. For a concrete example, consider the following code.

```
1  DEFINE_EFFECT(eff1, 0, void, {});
2  DEFINE_EFFECT(eff2, 1, void, {});
3  void *g(void *arg) { PERFORM(eff1); PERFORM(eff2); }
4  void *f(void *arg) {
5      seff_coroutine_t *k2 = seff_coroutine_new(g, NULL);
6      seff_request_t req1 = seff_handle(k2, NULL, HANDLES(eff2));
7      seff_request_t req2 = seff_handle(k2, NULL, HANDLES(eff2));
8  }
9  void main() {
10     seff_coroutine_t *k1 = seff_coroutine_new(f, NULL);
11     seff_request_t req1 = seff_handle(k1, NULL, HANDLES(eff1) | HANDLES(eff2));
12     seff_request_t req2 = seff_handle(k1, NULL, HANDLES(eff1) | HANDLES(eff2));
13 }
```

It sets up two nested coroutines and performs effects `eff1`, `eff2` from the innermost one. After both coroutines have been created and started, and immediately before the call to `PERFORM(eff1)`, the state of the system is as depicted in Figure 1. Both coroutines have been instantiated and are `RUNNING`, with the `current_coroutine` variable pointing to `k2`. As `k1` and `k2` are both `RUNNING`, their resumption contexts

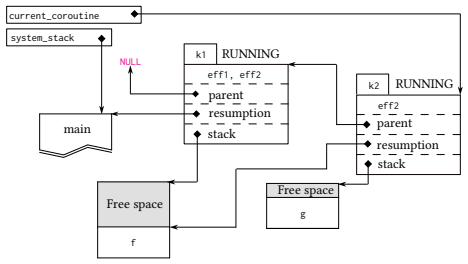


Fig. 1. Before `PERFORM(eff1)`

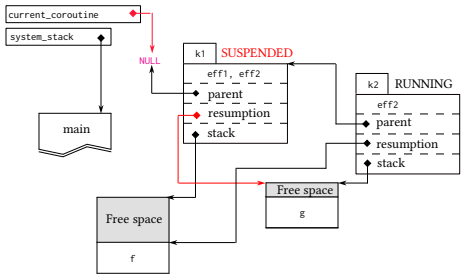


Fig. 2. After `PERFORM(eff1)`

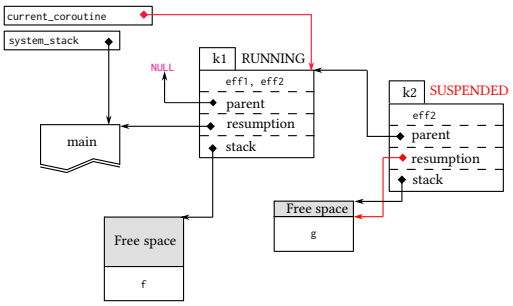


Fig. 3. After `PERFORM(eff2)`

both represent the execution state immediately before they were started (for `k1` the execution state right before the `seff_handle` call in line 11, and for `k2` the one on line 6).

When performing `eff1`, the linked list of coroutines is traversed upwards, starting at `k2`, to locate a suitable handler. In this case, `eff1` is in the effect set of `k1`, so `PERFORM(eff1)` immediately suspends `k1` and relinquishes control to its environment, which is then responsible for handling the effect. The system state is depicted in Figure 2: coroutine `k1` is `SUSPENDED` and its resumption is updated with the execution state immediately preceeding the call to `PERFORM(eff1)` inside the stack frame of the call to `g`. Note that `k2` remains unchanged and is still `RUNNING`.

After `eff1` has been handled (in this case the request object `req1` is simply ignored), execution of `k1` is resumed on line 12 and continues until the call to `PERFORM(eff2)`. Now the stack of active coroutines is traversed again until a handler for `eff2` is found. As `eff2` is in the effect set of `k2`, it is suspended and control is transferred back to line 6 in `f`. The system state corresponds to the diagram in Figure 3. Note that the resumption for `k1` is updated with the execution state of the now-paused coroutine.

In order to optimise the handling of effects, **`libseff`** takes particular care when passing the payload of an effect from the coroutine that performs the effect to the context that handles it. Effect payloads are marshalled, together with an effect tag, into a `seff_request_t` struct which effectively functions as an untyped discriminated union. In many high-level languages, creating such a data structure would involve allocating memory on the heap and thus incur a significant performance overhead. To avoid this, **`libseff`** uses two low-level tricks: first, the effect payload is allocated directly on the stack of the coroutine performing it, and the handler receives a pointer into this stack-allocated payload, which also saves the overhead of copying. Second, the `seff_request_t` struct consists of just two 64-bit fields: the tag and a pointer to the payload; hence it can be returned from `seff_handle` directly via processor registers.

3.2 Primitives

So far, we have illustrated only the higher-level interface provided by **libseff**, which is intended for general use and provides convenience and a degree of compiler checking of input and output types. Internally, operations such as the `PERFORM` macro are implemented in terms of a lower-level set of primitives, which we now describe.

At the lowest level, **libseff** has three context-switching primitives used for resuming or suspending an active coroutine, which are written directly in assembly.

```
seff_request_t seff_handle(seff_coroutine_t *k, void *arg, effect_set handled);
void *seff_yield(seff_coroutine_t *self, effect_id effect, void *payload);
void seff_exit(seff_coroutine_t *self, effect_id effect, void *payload);
```

We have already discussed `seff_handle` as it is also part of the higher-level interface. The `seff_yield` primitive suspends the coroutine `self` and returns control to the point where it was last resumed. The caller is responsible for ensuring that `self` is either the current coroutine or an ancestor of it, otherwise the call to `seff_yield` will result in undefined behaviour. (It would be possible to check this condition at runtime by traversing the list of coroutines to ensure that the coroutine being suspended is reachable from the currently active one, but we judged the overhead to be prohibitive.) The `seff_exit` primitive behaves similarly to `seff_yield`, with the difference that a coroutine that is suspended via `seff_exit` is considered terminated and can no longer be resumed. This means that the execution context need not be saved, so `seff_exit` is more efficient.

Both `seff_yield` and `seff_exit` take an `effect_id` argument, which is used to construct a request object. This argument is not, however, used to locate an appropriate handler. Instead, control is always relinquished to the last resumer of the given coroutine, whether or not it is able to handle the given effect.

A separate collection of primitives support handler-lookup.

```
seff_coroutine_t *seff_locate_handler(effect_id effect);
void *seff_perform(effect_id effect, void *payload);
void seff_throw(effect_id effect, void *payload);
```

The `seff_locate_handler` primitive traverses the stack of currently active coroutines to the first whose `handled_effects` bitset includes the effect `effect`. The `seff_perform` and `seff_throw` primitives are analogous to `seff_yield` and `seff_exit`, except that they use the given `effect_id` to select which coroutine to suspend. Semantically `seff_perform(e, p)` is equivalent to `seff_yield(seff_locate_handler(e), e, p)`, except that if no matching handler is found then the former will invoke a default handler, whereas the latter will dereference a null pointer.

The `PERFORM` macro (illustrated in §2) is the preferred interface for performing an effect. It is defined as a thin wrapper over `seff_perform`. A call to `PERFORM(eff, args...)` constructs a payload object of type `EFF_PAYLOAD_T(eff)` on the current stack frame, initialises it with the provided arguments, and then calls `seff_perform` with the effect and a pointer to the stack-allocated payload. Unlike in other systems with resizable stacks [16, 20, 33], **libseff** guarantees that coroutine's stack always remains at the same location; hence pointers into the stack remain valid when the coroutine is suspended.

3.3 Stack Management

One of the most important technical decisions when implementing stackful coroutines is how stack frames are allocated and, most importantly, resized. When designing **libseff**, we considered four different approaches, which we detail below. Currently, the first two (fixed and segmented stacks) are implemented and can be selected via a build flag (but should not be mixed together in the same project). The third approach (overcommiting) is planned but currently unimplemented. The last (stack copying) is unsuitable for C as pointers into the stack are pervasive.

3.3.1 Fixed-Size Stacks. The simplest approach to stack management is to reserve a fixed-size block of memory to hold the coroutine stack. This has the dual advantages of being easy to implement and introducing no additional runtime overhead. However, it can lead to a significant waste of memory. As it is hard to determine in advance how much stack space a given program will eventually need, the programmer must preemptively allocate larger stacks than necessary in order to mitigate the risk of stack overflow.

3.3.2 Segmented Stacks. Segmented stacks, also known as split stacks, replace the traditional contiguous fixed-size stack area by a linked list of stack segments or "stacklets". The compiler instruments every function with a small prelude which checks whether the current stack is large enough to accommodate a new stack frame for the function; if not, a new stacklet is allocated.

Conveniently, both GCC and Clang support segmented stacks via the `-fsplit-stack` flag, which adds stack overflow checks to every function preamble. As depicted in Figure 4, the compiler-generated prelude checks for a potential stack overflow and, if required, calls a routine `__morestack` which is responsible for allocating a new segment, copying any parameters that were passed through the stack, and setting the return address to point to an epilogue that frees the newly-allocated stacklet. A basic default implementation of `__morestack` is offered by both compilers, but **`libseff`** defines its own version in order to obtain finer-grained control over memory allocation.

Segmented stacks allow programmers to write code without being concerned about stack frame sizes. However, segmented stacks are not without disadvantages. If no memory needs to be allocated, the overhead of the function prelude is mostly negligible, but it is possible for a function call inside a tight loop to require the repeated allocation and deallocation of a large segment, resulting in a significant slowdown. This phenomenon is sometimes known as the "hot split" problem and motivated the Go designers to move away from segmented stacks [20]. The **`libseff`** library mitigates this issue by holding its stacklets in a doubly-linked list; when a stacklet is no longer necessary, instead of being released immediately it is simply kept at the end of this list. If a subsequent function call requires the allocation of additional stack memory then this stacklet can be recycled, avoiding an additional allocation. As we shall see in §4.1.2, with this optimisation, the worst-case cost of calling a function inside a hot split loop is 11x the cost of a normal function call.

```
1  lea -0x108(%rsp),%r11
2  cmp %fs:0x70,%r11          } Stack overflow check
3  ja 8
4  mov $0x108,%r10d           } Segment and argument size
5  mov $0x0,%r11d
6  call <__morestack>
7  ret
8  push %rbp
9  mov %rsp,%rbp
10 ...
11 pop %rbp
12 ret
```

```
1 void split_stack() {
2   char buffer[256];
3   ...
4 }
```

Fig. 4. Segmented Stack Prelude in Clang-12

This may sound excessive, but in practice is not a significant problem: if the cost of the hot split overhead dominates the execution time of the called function, then the function is small and so will be inlined by the compiler. Nonetheless, there is a plenty of scope for further improvement: micro-optimisations such as lowering the segment reuse code path to assembly, analysis-based optimisations like preemptively inlining functions that are likely to cause a hot split, or even more sophisticated runtime detection of such cases [19].

Another concern about code using segmented stacks is interoperability with library code. The use of segmented stacks relies on instrumenting every function with an overflow check, but any functions that are compiled separately (including the standard library, unless rebuilt from scratch with support for segmented stacks enabled) will lack this prelude and any stack overflow will cause

a crash, or worse, silent memory corruption. To avoid this issue, Clang and GCC's implementation of segmented stacks conservatively requests a much larger amount of stack space if a function calls any other functions that have been compiled without segmented stack support. This is a pragmatic compromise, but can lead to significantly higher memory consumption than necessary.

When using **libseff**, the space overhead can usually be avoided: a function not compiled with segmented stacks enabled cannot make use of the context-switching features of `libseff`, and therefore can be run directly on the system stack instead of the stack of whichever coroutine happens to be executing. This obviates the need to preemptively allocate a larger segment. For this purpose, **libseff** defines the `MAKE_SYSCALL_WRAPPER` macro, which wraps a given function in code that handles switching to and from the system stack.

```
MAKE_SYSCALL_WRAPPER(int, puts, const char *s);
// Expands to:
int __attribute__((no_split_stack)) puts_syscall_wrapper(const char *c);
__asm__("puts_syscall_wrapper:"
        "movq %rsp, %fs:_seff_paused_coroutine_stack@TPOFF;"
        "movq %fs:_seff_system_stack@TPOFF, %rsp;"
        "movq %fs:_seff_current_coroutine_stack_top, %rax;"
        "movq %rax, %fs:_seff_paused_coroutine_stack_top@TPOFF;"
        "movq $0, %fs:0x70;"
        "callq puts;"
        "movq %fs:_seff_paused_coroutine_stack@TPOFF, %rsp;"
        "movq %fs:_seff_paused_coroutine_stack_top@TPOFF, %rcx;"
        "movq %rcx, %fs:_seff_current_Coroutine_stack_top;"
        "retq;");
```

In the example above, a new function `puts_syscall_wrapper` is defined which has the same interface as the standard library function `puts`, but switches to the system stack instead of allocating a stack segment. The macro can generate a wrapper for any function that passes all arguments and its return value through processor registers. The programmer must take care to only invoke this wrapper from within a coroutine (outside a coroutine, the original function should be called instead).

3.3.3 Overcommiting. Another approach to avoid stack overflow without the need to physically resize a coroutine's stack is to *overcommit* a large amount of (virtual) memory for each coroutine, leaving it to the operating system to allocate physical memory as necessary. This approach is used by **libmprompt**, striking a judicious balance between performance and convenience in systems that support it. However, we intend **libseff** to be also deployable in embedded systems, which do not always support virtual memory or a large address space. Thus, while we plan to eventually provide virtual memory-based stack management for **libseff**, it is not a top priority.

3.3.4 Stack Copying. A popular approach in managed languages is *stack copying*: coroutines are initialised with a small, fixed-size stack and dynamic checks for stack overflow are inserted (much like in the case of segmented stacks). However, whenever a coroutine requires more stack space than is available, instead of initialising a new segment, an entire contiguous region is allocated to serve as the new stack and the contents of the old stack are copied onto it. This approach avoids the hot split problem, although it incurs the extra cost of copying the stack when resizing. Alas, stack copying is unsuitable for a low-level language like C as the process of copying the stack necessarily invalidates any pointers into it. This is not an insurmountable obstacle in some systems, for instance Go automatically rewrites any pointers into the stack as part of stack copying, but such rewriting relies on a degree of runtime information unavailable to a compiled C program.

4 PERFORMANCE

We evaluate **libseff** on a range of benchmarks, comparing it to other effect handler implementations as well as other concurrency mechanisms. All benchmarks were run on an Intel® Xeon® Gold 6154 x86-64 running Ubuntu 20.04, compiled with the clang 12.0.0 compiler. Except when stated otherwise we used **libseff** with segmented stacks.

4.1 Microbenchmarks

All benchmarks in this section are single-threaded.

4.1.1 State. Our first microbenchmark is based on the mutable state example of §2.1.

```
void *stateful(void *depth) {
  if (depth == 0){ for (int i = PERFORM(get); i > 0; i = PERFORM(get)) PERFORM(put, i - 1);
  } else {
    seff_coroutine_t *k = seff_coroutine_new(stateful, (void *)(uintptr_t)(depth - 1));
    seff_handle(k, NULL, HANDLES(error)); seff_coroutine_delete(k);
  }
  return NULL;
}
```

The `stateful` function recursively builds a stack of nested handlers for the `error` effect up to a specified depth. In the base case a counter, implemented using `get` and `put`, is decremented in a loop. The `error` effect is never actually performed, it is only used to construct the nested handlers sitting between the code performing the `get` and `put` effects and the code handling it.

In any effect handler framework, performing an effect involves two steps: (a) locating the appropriate handler; and (b) transferring control to the handler. This benchmark measures the cost of both steps and how they scale depending on the number of times an effect is performed and the depth of the target handler.

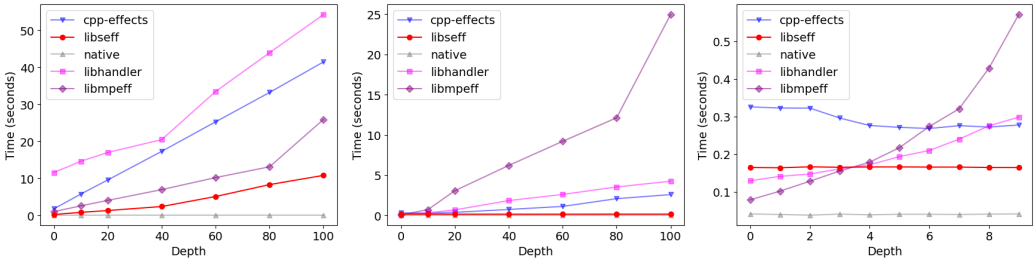
To separate out the cost of locating the handler from that of transferring control to the handler, we implement two versions of the benchmark. The first is the one above, where every execution of an effect triggers a search for its handler. The second is an optimised version that arises from observing that the handlers for `get` and `put` never change during execution of the loop, which allows us to locate the handlers once and then yield directly to the coroutine that handles them. This is shown in the code below, where `YIELD` wraps `seff_yield`, in the same way that `PERFORM` wraps `seff_perform`. If **libseff** were used as a backend for a higher-level language with effects, a compiler could systematically apply this optimisation.

```
    seff_coroutine_t *put_handler = seff_locate_handler(EFF_ID(put));
    seff_coroutine_t *get_handler = seff_locate_handler(EFF_ID(get));
    for (int i = YIELD(get_handler, get); i > 0; i = YIELD(get_handler, get))
      YIELD(put_handler, put, i - 1);
```

We compare against several libraries. For each library we implement a general case and an optimal case to compare against both of our implementations: **native** is plain C without effect handlers or any kind of dynamic dispatching of operations; **cpp-effects** [7] is a C++ effect handlers library; **libhandler** [15, 16] and **libmpeff** [17] are other C effect handlers libraries.

The **cpp-effects** optimal case avoids handler lookups in a similar fashion to **libseff**, but also eliminates context switching by requiring that the handler for `get` and `put` resumes immediately and does not need to capture a continuation. The **libmpeff** and **libhandler** optimal cases also similarly avoid context switches, but they do not allow for caching handler lookups.

Figure 5a shows the general case. All effect handler implementations degrade significantly as the number of installed handlers increases, with **libseff** being consistently the fastest. Figure 5b

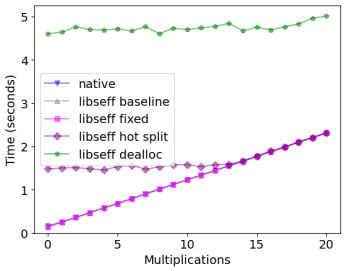


(a) General case (b) Optimal case (c) Optimal case, depths up to 10

Fig. 5. State Benchmark Results

Multiplications	0	5	10	15	20
native	1.27	1.00	1.00	1.00	1.00
libseff *baseline*	1.08	1.00	1.00	1.00	1.00
libseff *fixed*	1.00	1.00	1.00	1.00	1.00
libseff *hot split*	10.92	2.25	1.29	1.00	1.00
libseff *dealloc*	33.94	6.91	3.83	2.69	2.17

(a) Relative execution time of the hot split benchmark



(b)

Fig. 6. Hot Split Results

shows the optimal case. The elimination of traversing the stack of handlers gives **libseff** and **cpp-effects** a distinct advantage. Remarkably, **libseff** is asymptotically the fastest implementation despite not being able to optimise away the context switching when performing an effect. For both **libhandler** and **libmpeff** the optimal case is still affected by recursion depth. Whereas **libhandler** speeds up by avoiding copying the stack in this case, **libmpeff** shows little improvement over the general implementation when nested handlers are introduced. Figure 5c shows the optimal case with recursion depth less than 10. Whereas, **libmpeff** and **libhandler** are initially faster than both **cpp-effects** and **libseff**, the cost of searching for handlers quickly becomes a bottleneck, being slower than **libseff** beyond recursion depth 3. Surprisingly, **cpp-effects** performs better beyond a certain depth. Some preliminary testing indicates that this is due to cache alignment effects.

4.1.2 Hot Split. The next benchmark is designed to quantify the cost of the hot split problem, as discussed in §3.3.2. It forces a function call to require more stack space than available in the current segment, and therefore request a larger one every time it is called. This function is then repeatedly called from a tight loop executing 10^8 times.

We compare four different configurations for **libseff** against the optimal case in plain C without segmented stacks, where a function call translates to exactly one assembly call operation. We vary the called function slightly to include a number of floating point multiplications, ranging from 0 to 20. Figure 7 shows the resulting compiled code for the **native** case, where the `movsd` instructions are only present if there is at least one multiplication and in between there is a fixed number of `mulsd` operations. Lines 1 and 9 reserve and release a large array on the stack. When using **libseff** with segmented stacks, this function includes runtime checks similar to those of Figure 4.

Table 6a shows the results, comparing **native** with four different variations of **libseff**. The *baseline* variation performs the same function call, repeatedly checking that there is enough space but never requesting a larger segment; *fixed* uses **libseff** with a fixed-size stack (§3.3), which

yields exactly the code in Figure 7; *hot split* is the case we are most interested in, where a new segment is requested every time the function is called; and *dealloc* is a special case where each segment is freed after it is used instead of being recycled (§3.3.2).

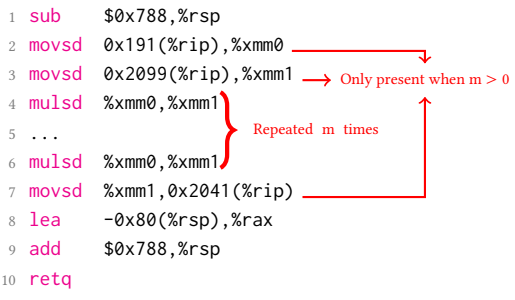


Fig. 7. Hot Split Function Body in Assembly (*m* is the number of multiplications)

The results show that the hot split problem is observable in **libseff**, causing a call to an empty function to be 11 times slower. However, as we increase the number of operations executed by, the function the relative overhead incurred by the segment switching rapidly diminishes. The cost of a mere 13 multiplications dominates this overhead, and the difference in cost of the function call becomes negligible, as shown in Figure 6b. The results for *dealloc* illustrate the significant performance difference that recycling segments provides.

When no multiplications are inserted, we observed that changing the position of the functions in the compiled code, by adding `nop` instructions, could affect performance by up to 40%, which explains why both *fixed* and *baseline* are faster than **native**. A similar behaviour was noted in [25]. when evaluating the cost of low level operations.

The hot split problem is only observable at all if functions are not inlined or completely optimised away. Modern compilers invariably do inline functions that are simple enough that the call is the dominant cost. Indeed, to force the compiler to generate the code from Figure 7 without writing the assembly instructions by hand it was necessary to deliberately disable inlining and introduce empty inline assembly blocks.

4.2 Macrobenchmarks

In this section we benchmark **libseff** against other systems running whole applications.

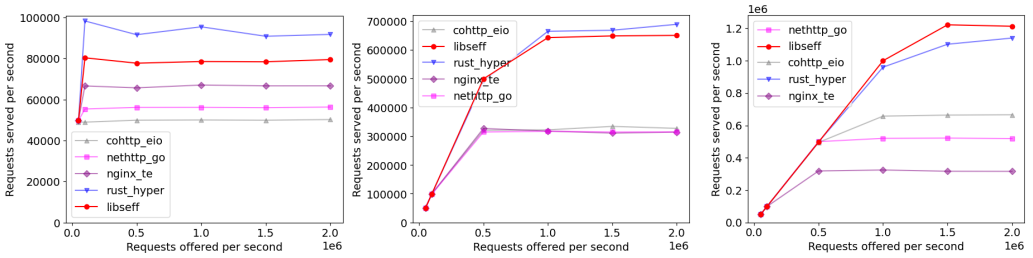
4.2.1 HTTP Server. Our first macrobenchmark is a simple HTTP server similar to the one used to benchmark OCaml 5 (formerly Multicore OCaml) [25, 31] and to the *Plaintext* benchmark from TechEmpower [32]. The server must receive `GET` requests and respond to them asynchronously with a constant `text/plain` message.

The **libseff** implementation uses a single coroutine per connection that is kept alive until the connection is closed; it is based on a multithreaded work-stealing scheduler backed by a number of OS threads which we set to 1, 8, and 16. We use asynchronous I/O functions built on top of our scheduler and a small but complete HTTP request parser [21].

We used the `wrk2` tool [34] to generate the workload and record the results, 32 OS threads were dedicated to the clients, while varying the requests per second and the connections as shown in the figures. Each experiment was run for 30 seconds.

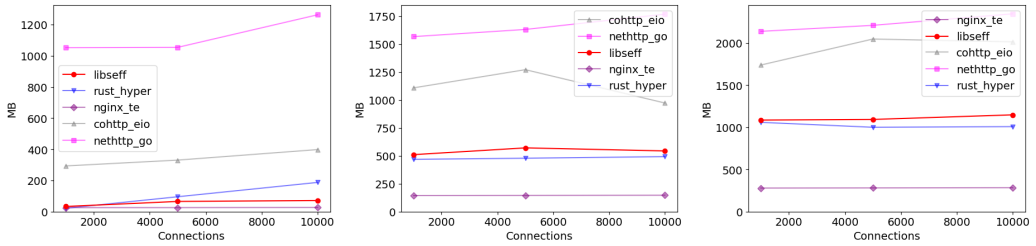
We compare against four alternative implementations:

- *nethttp_go* is built using Go's `net/http` package, that is part of the standard library of the language.
- *rust_hyper* is a server built on top of Hyper, a highly performant HTTP library for Rust for the Tokio runtime, a state of the art runtime for Rust async/await concurrency.
- *cohttp_eio* is a server implemented for OCaml 5 over an effect based I/O library [29] and an HTTP library built on top of it [30].
- *nginx* is a server based on the Nginx framework.



(a) 1 OS thread (b) 8 OS threads (c) 16 OS threads

Fig. 8. Requests Per Second Served Per Offered (with 1000 live connections)



(a) 1 OS thread (b) 8 OS threads (c) 16 OS threads

Fig. 9. Maximum Memory Consumed (with 1000000 requests offered per second)

The first two are the ones used to compare against the original implementation of effect handlers in OCaml [25]. The third is an updated version of the OCaml code used in that comparison. The fourth is one of the most widely used industry-standard web servers.

Figure 8 shows the throughput when running on 1, 8, and 16 OS threads. **`libseff`** and *rust_hyper* perform consistently better than *nethttp_go*, *cohttp_eio*, and *nginx*, regardless of the number of OS threads. However, *nginx*, does not scale as well as any of the other frameworks when extra OS threads are added. Figure 9 shows maximum memory consumption. We observed the maximum memory used by each implementation by varying the number of live connections, which coincides with the maximum number of coroutines spawned by the **`libseff`** implementation. Both *nethttp_go* and *cohttp_eio* have the most memory consumption of all frameworks. Again, *rust_hyper* and **`libseff`** are comparable whereas *nginx* is consistently better, which we expect is due to it being based on an event-driven architecture rather than coroutines.

4.2.2 Prefetching. Our last benchmark, inspired by Jonathan et al. [10], uses C++'s coroutines to improve performance in memory-bound applications by alternating multiple concurrent runs and prefetching memory locations to cache before executing reads. The application executes multiple binary searches of different values over the same array. The array is big enough to not fit entirely in cache, and accesses to memory are not linear, making cache misses a significant part of the cost.

The naïve version executes searches sequentially; both the C++ coroutine implementation and the **`libseff`** implementation interleave multiple searches. Each search hints to the CPU to prefetch some address from memory and then, in a round-robin

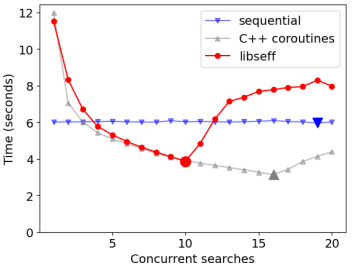


Fig. 10. Prefetch Benchmark Results (large shapes mark the fastest execution for each framework)

manner, moves on to the next search. Before execution returns to the coroutine that requested the prefetch, the values will already be stored in the cache and ready to be read, minimising cache misses. By varying the number of concurrent searches we can minimise the waiting time between execution returning to the search and the read being completed.

Whereas the C++ coroutine implementation is explicit about prefetching, then yielding execution, and finally reading the memory upon return, the **libseff** version treats dereferencing a memory location as an effect; it is the responsibility of the handler to prefetch the memory location, suspend the coroutine for some time, and eventually provide it with the contents of the memory.

```
bool seff_binary_search(int const *first, size_t len, int val) {
  while (len > 0) {
      size_t half = len / 2; int x = PERFORM(deref, first + half);
      if (x < val) { first += half + 1; len = len - half - 1; }
      else { len = half; }
      if (x == val) return true;
  }
  return false;
}
```

The code above is a simplified version of binary search from the **libseff** benchmark. The main difference from regular binary search is the effectful computation of the dereference operation `PERFORM(deref, first + half)`. Figure 10 shows the results: **libseff** incurs an overhead over the naïve sequential implementation whenever too few or too many streams are used but significantly improves upon it in the best case, taking around 2/3 of the time. The version with C++ coroutines is noticeably faster and allows for more concurrent searches to be executed simultaneously; this is unsurprising as C++ stackless coroutines have full compiler support and leverage a smaller memory footprint and stack allocation for better cache locality. Nonetheless, these results show that **libseff** effects are lightweight and efficient enough to materialise performance gains from cache prefetching, without these being obscured by context-switching overhead.

5 USABILITY

In this section we briefly compare the experience of writing programs directly in **libseff** and **libmpeff**. Whereas we explicitly set out to design **libseff** to support C programmers to write effect handlers directly, **libmpeff** is aimed at compiler backends.

We implement mutable state as an effect in both **libseff** and **libmpeff**. The **libseff** code is from Section 2.1 and the **libmpeff** version from its test suite.

Effectful operations and their signatures are defined similarly in both libraries.

libseff

```
DEFINE_EFFECT(get, 0, int64_t, {});
DEFINE_EFFECT(put, 1, void, {
    int64_t new_value; });
```

libmpeff

```
MPE_DEFINE_EFFECT2(state, get, set)
MPE_DEFINE_OP0(state, get, int)
MPE_DEFINE_VOIDOP1(state, set, int)
```

However, whereas **libseff** defines effects individually, **libmpeff** groups them together. Moreover, **libseff** requires the programmer to provide an explicit effect tag.

We implement an effectful computation, `counter`, using the mutable state operations (Figure 11). The code is essentially the same in both cases, the only difference being that **libmpeff** generates C functions `state_set` and `state_get`, whereas effects in **libseff** are all performed via the `PERFORM`

macro, which the programmer is free to manually wrap in a function. Where the libraries are most different is in the way in which effect handlers are written (Figure 12 and 13).

```c
void* counter(void* parameter) {
    int64_t counter;
    do {
        counter = PERFORM(get);
        printf("Counter is %ld\n", counter);
        PERFORM(put, counter - 1);
    } while (counter > 0);
    return NULL;
}
```

(a) **libseff**

```c
void* counter(void* parameter) {
    int64_t counter;
    do {
        counter = state_get();
        printf("Counter is %ld\n", counter);
        state_set(counter - 1);
    } while (counter > 0);
    return NULL;
}
```

(b) **libmpeff**

Fig. 11. A Counter in **libseff** and **libmpeff**

```c
int main(void) {
    effect_set handles_state = HANDLES(get) | HANDLES(put);
    seff_coroutine_t *k = seff_coroutine_new(counter, NULL);
    seff_request_t req = seff_handle(k, NULL, handles_state);
    int64_t state = 100;
    while (true) {
        switch (req.effect) {
            CASE_EFFECT(req, get, {
                req = seff_handle(k, (void *)state, handles_state);
                break;
            })
            CASE_EFFECT(req, put, {
                state = payload.new_value;
                req = seff_handle(k, NULL, handles_state);
                break;
            })
            CASE_RETURN(req, {
                printf("The handled code has finished executing\n");
                seff_coroutine_delete(k);
                return 0;
            })
        }
    }
}
```

Fig. 12. State Handler in **libseff**

The **libseff** approach is based on reified effects and mutable coroutines. The more standard **libmpeff** approach is based on families of closures that take the resumption as a parameter.

Representation of control. The **libmpeff** library follows the traditional approach of managing control by means of resumptions, which are allocated whenever an effect is performed and can be later resumed by the handler. Such allocations can sometimes be optimised away, for example if

```
static void* _state_get(mpe_resume_t* r, void* local, void* arg) {
    return mpe_resume_tail(r, local, local);
}
static void* _state_set(mpe_resume_t* r, void* local, void* arg) {
    return mpe_resume_tail(r, arg, NULL);
}
static const mpe_handlerdef_t state_hdef = { MPE_EFFECT(state), NULL, {
    { MPE_OP_SCOPED_ONCE, MPE_OPTAG(state,get), &_state_get },
    { MPE_OP_SCOPED_ONCE, MPE_OPTAG(state,set), &_state_set },
    { MPE_OP_NULL, mpe_op_null, NULL }
}};
static void* state_handle(mpe_actionfun_t action, int init, void* arg) {
    return mpe_handle(&state_hdef, mpe_voidp_int(init), action, arg);
}
int main(void) {
  int res = mpe_int_voidp(state_handle(&counter, 100, NULL));
  printf("The handled code has finished executing\n");
  return 0;
}
```

Fig. 13. State Handler in **libmpeff**

they never escape the scope of the handler. This design also offers some degree of safety in garbage-collected languages, where resumptions can be updated after the first time they are resumed to indicate that they should not be resumed again, enforcing that their usage is linear at runtime. On the other hand it is not such a natural fit for a language like C with manual memory management.

In contrast, **libseff**'s approach ensures that heap allocation is only necessary when a coroutine is first created. Subsequently when effects are performed the same coroutine object is reused. (At the cost of some implementation complexity it is possible to mitigate the downsides of the traditional approach using a representation based on a sequence counter and fat pointers [22].)

Handler dispatch. Typical implementations of effect handlers, including **libmpeff**, encode a handler as a closure stored in an operation-indexed map. When performing an effect, the appropriate handler closure is selected and invoked with an operation and a resumption as parameters. This scheme is particularly convenient for implementing deep handlers, since handler code, and any data it references, may outlive and escape the lexical scope in which it was defined.

An important advantage of this representation is that handler code, stored as a closure, can be executed in any given scope. However, such representations introduces significant accidental complexity to a C program. As the language has no native support for closures, the programmer has to effectively closure-convert their code "by hand". In Figure 13, the closures that handle `get` and `set` in **libmpeff** must take an additional, untyped `local` parameter which contains the state that is shared across them. In our example, this is just an integer stored as a `void*`, but to share richer complex state, for example a reference to a scheduler when implementing light-weight concurrency, the programmer would need to define and allocate an ad-hoc structure to store it.

In contrast, **libseff** implements shallow (more properly, *sheep* [22]) handlers. As a consequence of this, there is no need to reify or store handler code, as every time a coroutine is resumed a new handler is installed. Instead, we implement effects as requests which are returned from a call to `seff_handle` and can be inspected by an explicit `switch` statement. This eliminates the need for

manual closure-conversion and allows state to be shared across different clauses of the handler: observe that in Figure 12 the `state` variable is directly used in clauses for `get` and `put` operations.

Performance tuning. Due in part to the more sophisticated encoding, **libmpeff**'s handlers offer more opportunities for manual performance tuning. In the example above, the user can control the trade-off between expressiveness and performance by tuning the kind of resumption that is passed to the handler (we use `MPE_OP_SCOPED_ONCE`, which indicates that the resumption will only be used once, and only within the scope of the handler) and the way it is resumed (we specify `mpe_resume_tail`, which precludes any code in the handler from executing after the resumption).

This flexibility is a good fit for **libmpeff**'s role as a backend for an optimising compiler, which can safely select all the relevant parameters based on static analysis of higher-level code. However, when used directly from C, the many options is a potential source of bugs, as the programmer is now responsible for enforcing the restrictions assumed by the configuration options and Violating them may result in undefined behaviour.

When designing **libseff**, we focused on one particular use case which roughly corresponds to **libmpeff**'s non-scoped linear resumptions (`MPE_OP_ONCE`) and regular non-tail-recursive resume (`mpe_resume`), as more restrictive configurations are too limited to implement lightweight concurrency and application-specific schedulers. This allowed us to drastically simplify the library and optimise for this particular usage, obtaining comparable performance even when competing in unfavourable conditions against **libmpeff**, as shown in Figure 5b.

Defining effects. Like OCaml, **libseff** defines each effect operation individually and allows handlers to handle arbitrary combinations of operations. In contrast, **libmpeff** requires effect operations to be grouped together, with each handler handling exactly one such group of effects.

Although this design choice is largely orthogonal to other characteristics of both libraries, it has one important implication: **libseff** uses bit sets to encode which effects are handled by a handler, and therefore needs 1 bit per effect; since this value is stored in a 64-bit integer, there is a hard limit of 64 operations (which could straightforwardly be extended to a larger, but fixed, amount).

By grouping operations and forcing a handler to handle exactly one such group, **libmpeff** can allot a tag to each group. With 64-bit integers, this means that up to 2^{64} different families of operations can be in use at the same time in the same program. This also allows tags to be generated automatically by allocating a global variable per family of operations and using its address in memory as the corresponding tag.

We have not found **libseff**'s limitations to cause a problem with our experiments so far. However, we speculate that the additional convenience and expressiveness gained by grouping operations together may be worth trading for the flexibility of handling arbitrary sets of operations, and we will consider switching **libseff** to a similar system in future.

6 RELATED WORK

Effect handlers for C and C++. Unlike **libseff**, the existing **libhandler** [15, 16] and **libmpeff** [17] libraries are designed as targets for compiler writers rather than for writing code directly in C. Each of these uses a different stack-management strategy: **libhandler** copies stacks into a temporary structure before restoring them on resumption and **libmpeff** uses virtual memory to allow stacks to grow without moving in memory, whereas **libseff** can use segmented or fixed-size stacks. The **cpp-effects** library [7] is a C++ effect handlers library which heavily relies on C++ features both in its implementation and its API. It is implemented on top of the `Boost.Context` library [28], which provides low-level for undelimited continuations.

Coroutines in C/C++. There exist many different coroutine libraries for C and C++, including Boost coroutines [27], **libco** [33], **libmill** [26], and C++20 stackless coroutines.

Varieties of coroutine. de Moura and Ierusalimschy [3] give a comprehensive classifications of the different notions of coroutine. The kind of coroutines provided by **libseff** are *asymmetric first-class stackful* coroutines. Moreover, **libseff** provides stacks that are guaranteed not to be moved and coroutines that can migrate between system threads.

Effect handlers as coroutines. A distinctive aspect of effect handlers in **libseff** is their foundation on mutable asymmetric coroutines rather than immutable continuations. Nonetheless, the close connections between asymmetric coroutines and effect handlers have been exploited, in a somewhat different way, elsewhere. Kawahara and Kameyama [13] show how to translate one-shot effect handlers into asymmetric coroutines. Phipps-Costin et al. [22] exploit essentially the same encoding to implement effect handlers on top of the Wasmtime Fiber API [2], which implements coroutines for the Wasmtime engine for WebAssembly.

Optimising effect handlers. Much of the research on effect handlers has focused on programming and reasoning with them. Nonetheless, there have also been various attempts to compile effect handlers to efficient code. Kammar et al. [11] take advantage of Haskell's aggressive inlining of type classes to speed up an implementation based on a continuation monad. Wu and Schrijvers [35] explain the essence of this optimisation as an instance of fusion. Kiselyov and Ishii [14] introduce so-called "Freer monads" as another means to speed up implementations of effect handlers in Haskell. Karachalias et al. [12] optimise effect handler code by aggressively inlining as many handlers as possible. Schuster et al. [24] achieve a similar end by way of staged computation. Xie and Leijen [36] and Ghica et al. [7] apply an instance of the optimisation we describe in Section 4 to avoid searching the handler stack, or indeed context-switching at all, when a handler is known to be "tail-resumptive" meaning that it immediately invokes the continuation in tail-position. Another optimisation performed by both of the latter two systems is for the case in which the continuation is never invoked (as in exception handlers).

7 CONCLUSION AND FUTURE WORK

We have described the design and implementation of **libseff**, a library for effect handlers in C. While other effect handlers for C exist, these are primarily designed as targets for compilers, whereas **libseff** offers a more idiomatic interface for programming with effect handlers in C. The key challenge we had to overcome is C's lack of high-level features, especially closures and generics. This led us to a design based on sheep handlers, coroutines, and explicit request objects, which enables writing handlers as simple, direct-style loops that should be familiar to C developers.

Our benchmarks demonstrate that effect handler programs, even without special compiler support, can be compiled to efficient code that is competitive with other state-of-the-art approaches, notably Rust's stackless coroutines. The **libseff** library outperforms most other libraries in this space due to simpler handler dispatch logic and hand-written context-switching code. It is also, to our knowledge, the first such C library to offer a choice of stack management strategies, currently supporting both segmented and fixed-size stacks, with planned support for a third approach based on overcommitting of virtual memory as well as some form of arena allocation. We are currently actively working on porting **libseff** to other architectures including ARM and 32-bit Intel processors.

Perhaps unsurprisingly for a C library, the interface provided by **libseff** is prone to certain kinds of errors: using coroutines non-linearly or performing unhandled effects can crash the program at runtime. The C type system is not rich enough to encode the necessary constraints to avoid these

errors, but we would like to develop a set of Rust bindings on top of **`libseff`** that leverage Rust's rich type system and borrow checker to ensure safety at compile time.

We plan to explore further low-level improvements to the **`libseff`** implementation. A common optimisation in other effect handler libraries is to avoid creating new continuations or stack frames for certain effects where the continuation is either never invoked (such as exceptions) or invoked immediately at the end of the handler (such as mutable state or dynamic binding). This optimisation promises additional performance for such use-cases allowing us to efficiently take fuller advantage of the expressive power of effect handlers.

REFERENCES

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

[2] Alex Crichton. 2021. Wasmtime Fiber API. https://docs.wasmtime.dev/api/wasmtime_fiber/index.html. Accessed 2023-11-15.

[3] Ana Lúcia de Moura and Roberto Ierusalimschy. 2009. Revisiting coroutines. *ACM Trans. Program. Lang. Syst.* 31, 2 (2009), 6:1–6:31.

[4] KC Sivaramakrishnan Deepali Ande, Sudha Parimala. 2023. Effectively Composing Concurrency Libraries. Draft. https://kcsrk.info/papers/composable_concurrency.pdf.

[5] Stephen Dolan, Spiros Eliopoulos, Daniel Hillerström, Anil Madhavapeddy, KC Sivaramakrishnan, and Leo White. 2017. Concurrent System Programming with Effect Handlers. In *TFP (Lecture Notes in Computer Science, Vol. 10788)*. Springer, 98–117.

[6] Matthias Felleisen and Daniel P. Friedman. 1986. Control Operators, the SECD-machine, and the λ-Calculus. In *Formal Description of Programming Concepts III*. Elsevier, 193–217.

[7] Dan R. Ghica, Sam Lindley, Marcos Maroñas Bravo, and Maciej Piróg. 2022. High-level effect handlers in C++. *Proc. ACM Program. Lang.* 6, OOPSLA2 (2022), 1639–1667.

[8] Daniel Hillerström, Sam Lindley, and Robert Atkey. 2020. Effect handlers via generalised continuations. *J. Funct. Program.* 30 (2020), e5.

[9] Daniel Hillerström. 2021. *Foundations for Programming and Implementing Effect Handlers*. Ph. D. Dissertation. School of Informatics, The University of Edinburgh, Scotland, UK.

[10] Christopher Jonathan, Umar Farooq Minhas, James Hunter, Justin J. Levandoski, and Gor V. Nishanov. 2018. Exploiting Coroutines to Attack the "Killer Nanoseconds". *Proc. VLDB Endow.* 11, 11 (2018), 1702–1714. https://doi.org/10.14778/3236187.3236216

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

[12] Georgios Karachalias, Filip Koprivec, Matija Pretnar, and Tom Schrijvers. 2021. Efficient compilation of algebraic effect handlers. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–28.

[13] Satoru Kawahara and Yukiyoshi Kameyama. 2020. One-Shot Algebraic Effects as Coroutines. In *TFP (Lecture Notes in Computer Science, Vol. 12222)*. Springer, 159–179.

[14] Oleg Kiselyov and Hiromi Ishii. 2015. Freer monads, more extensible effects. In *Haskell*. ACM, 94–105.

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

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

[17] Daan Leijen and KC Sivamarakrishnan. 2023. libmprompt and libmpeff. https://github.com/koka-lang/libmprompt.

[18] Paul Blain Levy, John Power, and Hayo Thielecke. 2003. Modelling environments in call-by-value programming languages. *Inf. Comput.* 185, 2 (2003), 182–210.

[19] Zhiyao Ma and Lin Zhong. 2023. Bringing Segmented Stacks to Embedded Systems. In *Proceedings of the 24th International Workshop on Mobile Computing Systems and Applications, HotMobile 2023, Newport Beach, California, February 22-23, 2023*. ACM, 117–123. https://doi.org/10.1145/3572864.3580344

[20] Daniel Morsing. 2014. How Stacks are Handled in Go. https://blog.cloudflare.com/how-stacks-are-handled-in-go/.

[21] Kazuho Oku, Tokuhiro Matsuno, Daisuke Murase, and Shigeo Mitsunari. 2024. PicoHTTPParser. https://github.com/h2o/picohttpparser.

[22] Luna Phipps-Costin, Andreas Rossberg, Arjun Guha, Daan Leijen, Daniel Hillerström, KC Sivaramakrishnan, Matija Pretnar, and Sam Lindley. 2023. Continuing WebAssembly with Effect Handlers. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023), 460–485.

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

[24] Philipp Schuster, Jonathan Immanuel Brachthäuser, and Klaus Ostermann. 2020. Compiling effect handlers in capability-passing style. *Proc. ACM Program. Lang.* 4, ICFP (2020), 93:1–93:28.
[25] KC Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. 2021. Retrofitting effect handlers onto OCaml. In *PLDI*. ACM, 206–221.
[26] Martin Sustrik. 2021. libmill. https://github.com/sustrik/libmill.
[27] Boost team. 2021. Boost.coroutine library. https://www.boost.org/doc/libs/1_83_0/libs/coroutine/doc/html/index.html.
[28] Boost team. 2024. Boost.context library. https://github.com/boostorg/context.
[29] Eio team. 2024. Eio. https://github.com/ocaml-multicore/eio.
[30] Mirage team. 2024. Mirage. https://github.com/mirage/ocaml-cohttp.
[31] OCaml Multicore team. 2021. Multicore OCaml HTTP benchmarks. https://github.com/ocaml-multicore/retro-httpaf-bench.
[32] TechEmpower. 2023. Web Framework Benchmarks. https://www.techempower.com/benchmarks.
[33] Tencent. 2020. libco. https://github.com/Tencent/libco.
[34] wrk2 team. 2024. wrk2. https://github.com/giltene/wrk2.
[35] Nicolas Wu and Tom Schrijvers. 2015. Fusion for Free - Efficient Algebraic Effect Handlers. In *MPC (Lecture Notes in Computer Science, Vol. 9129)*. Springer, 302–322.
[36] Ningning Xie and Daan Leijen. 2021. Generalized evidence passing for effect handlers: efficient compilation of effect handlers to C. *Proc. ACM Program. Lang.* 5, ICFP (2021), 1–30.

A SEMANTICS

In this appendix we give an abstract characterisation of the variant of effect handlers that **libseff** is based on. Following the approach of Hillerström et al. [8] we do so by way of a CEK [6] abstract machine for a fine-grain call-by-value [18] lambda calculus. Our calculus is untyped whereas theirs is simply-typed. Other than that, the only substantive difference between our account and that of Hillerström et al. [8] is the treatment of effects and handlers. We return to these differences after presenting our calculus and abstract machine. Again following Hillerström et al. [8], we diverge somewhat from **libseff** by basing the effect handlers in this section on continuations rather than coroutines. We make no attempt here to prevent continuations from being invoked more than once in the abstract machine, but it would be entirely straightforward to do so.

The syntax of our calculus is given by the following grammar.

$$
\begin{array}{lll}
\text{Values} & V, W & ::= x \mid k \mid c \mid \lambda x.\, M \mid \mathbf{rec}\ f\ x.M \mid \langle\rangle \mid \langle V, W\rangle \mid \mathbf{inj}_\ell\ V \\
\text{Computations} & M, N & ::= V\ W \mid \mathbf{let}\ \langle x, y\rangle = V\ \mathbf{in}\ N \mid \mathbf{case}\ V\ \{\mathbf{inj}_\ell\ x \mapsto M; y \mapsto N\} \\
& & \quad \mid \mathbf{return}\ V \mid \mathbf{let}\ x \leftarrow M\ \mathbf{in}\ N \\
& & \quad \mid \mathbf{newcont}\ V \mid \mathbf{resume}\ \mathcal{L}\ V\ W \mid \mathbf{perform}\ \ell\ V
\end{array}
$$

We let V range over value terms, M range of computation terms, x range over value term variables, k range of literals, c range of primitive operations (e.g. addition), ℓ range over individual effects, and $\mathcal{L}$ range over sets of effects. Being fine-grained, there are different productions for value and computation terms. Apart from **newcont**, **resume**, and **perform** computation term constructors, everything else is standard.

The term **newcont** V converts a function value V into a continuation value. It is an idealised analogue of `seff_coroutine_new(f, NULL)`, where V represents `f`. The term **resume** $\mathcal{L}\ V\ W$ resumes continuation V with argument W handling effects $\mathcal{L}$. It is an idealised analogue of `seff_handle(k, arg, effs)` where $\mathcal{L}$ represents `effs`, V represents `k`, and W represents `arg`. The term **perform** $\ell\ V$ performs effect ℓ with argument V. It is an idealised analogue of `seff_perform(eff, arg)` where ℓ represents `eff` and V represents `arg`.

Before giving the transition relation for the machine we spell out the grammar for abstract machine syntax.

$$
\begin{array}{lll}
\text{Configurations} & C & ::= \langle M \mid \gamma \mid \kappa\rangle \\
\text{Environments} & \gamma & ::= \emptyset \mid \gamma[x \mapsto v] \\
\text{Machine values} & v, w & ::= x \mid k \mid c \mid (\gamma, \lambda x.\, M) \mid (\gamma, \mathbf{rec}\ f\ x.\, M) \mid \langle\rangle \mid \langle v, w\rangle \mid \mathbf{inj}_\ell\ v \mid (\kappa, \sigma) \\
\text{Continuations} & \kappa & ::= [] \mid (\sigma, \mathcal{L}) :: \kappa \\
\text{Pure continuations} & \sigma & ::= [] \mid (\gamma, x, N) :: \sigma
\end{array}
$$

The configurations (C) of a CEK machine are triples: C (here ranged over by M, N) stands for control (the program, that is, current computation term), E (here ranged over by γ) for environment (a mapping from variables to machine values), K (here ranged over by κ) for kontinuation (what to do next).

The machine values are mostly quite standard, including corresponding forms for each basic term value form. Indeed, we define an interpretation $[\![V]\!]\gamma$ for value term V as a machine value, where free variables are given by the environment γ.

$$
\begin{array}{ccc}
[\![x]\!]\gamma = \gamma(x) & [\![\lambda x.M]\!]\gamma = (\gamma, \lambda x.M) & [\![\langle\rangle]\!]\gamma = \langle\rangle \\
[\![k]\!]\gamma = k & [\![\mathbf{rec}\ f\ x.M]\!]\gamma = (\gamma, \mathbf{rec}\ f\ x.M) & [\![\langle V, W\rangle]\!]\gamma = \langle [\![V]\!]\gamma, [\![W]\!]\gamma\rangle \\
[\![c]\!]\gamma = c & & [\![\mathbf{inj}_\ell\ V]\!]\gamma = \mathbf{inj}_\ell\ ([\![V]\!]\gamma)
\end{array}
$$

In particular, anonymous function terms and named recursive function terms are interpreted using closures. The final machine value form (κ, σ) is used to represent a continuation value (as returned

by **newcont** or when performing an effect). Let us defer explaining why continuation values are represented this way to the point at which we consider the transition rules.

Following Hillerström et al. [8] a continuation (κ) is a list (stack) of pairs of pure continuations σ and handlers $\mathcal{L}$ (here actually effects sets which denote the effects handled by a handler). A pure continuation (σ) is a list (stack) of let-binding closures. (A traditional CEK machine for coarse-grained call-by-value would need many more. The advantage of fine-grain call-by-value — or ANF or SSA or CPS — is that because the result of every intermediate step must be explicitly named we know that pure computation can only proceed through another let-binding.)

Now we present the transition relation for the abstract machine.

$$\text{M-Lam} \quad \langle V\ W \mid \gamma \mid \kappa\rangle \longrightarrow \langle M \mid \gamma'[x \mapsto [\![W]\!]\gamma] \mid \kappa\rangle, \quad \text{if } [\![V]\!]\gamma = (\gamma', \lambda x.\, M)$$

$$\text{M-Rec} \quad \langle V\ W \mid \gamma \mid \kappa\rangle \longrightarrow \langle M \mid \gamma'[f \mapsto (\gamma', \mathbf{rec}\ f\ x.M),\ x \mapsto [\![W]\!]\gamma] \mid \kappa\rangle, \quad \text{if } [\![V]\!]\gamma = (\gamma', \mathbf{rec}\ f\ x.M)$$

$$\text{M-Const} \quad \langle V\ W \mid \gamma \mid \kappa\rangle \longrightarrow \langle \mathbf{return}\ (\ulcorner c \urcorner\ ([\![W]\!]\gamma)) \mid \gamma \mid \kappa\rangle, \quad \text{if } [\![V]\!]\gamma = c$$

$$\text{M-Split} \quad \langle \mathbf{let}\ \langle x, y\rangle = V\ \mathbf{in}\ N \mid \gamma \mid \kappa\rangle \longrightarrow \langle N \mid \gamma[x \mapsto v, y \mapsto w] \mid \kappa\rangle, \quad \text{if } [\![V]\!]\gamma = \langle v, w\rangle$$

$$\text{M-CaseMatch} \quad \mathbf{case}\ V\ \{\mathbf{inj}_\ell\ x \mapsto M; y \mapsto N\} \mid \gamma \mid \kappa \longrightarrow \langle M \mid \gamma[x \mapsto v] \mid \kappa\rangle, \quad \text{if } [\![V]\!]\gamma = \mathbf{inj}_\ell\ v$$

$$\text{M-CaseDef} \quad \mathbf{case}\ V\ \{\mathbf{inj}_\ell\ x \mapsto M; y \mapsto N\} \mid \gamma \mid \kappa \longrightarrow \langle N \mid \gamma[y \mapsto \mathbf{inj}_{\ell'}\ v] \mid \kappa\rangle, \quad \text{if } [\![V]\!]\gamma = \mathbf{inj}_{\ell'}\ v \text{ and } \ell \neq \ell'$$

$$\text{M-Let} \quad \langle \mathbf{let}\ x \leftarrow M\ \mathbf{in}\ N \mid \gamma \mid (\sigma, \mathcal{L}) :: \kappa\rangle \longrightarrow \langle M \mid \gamma \mid ((\gamma, x, N) :: \sigma, \mathcal{L}) :: \kappa\rangle$$

$$\text{M-RetCont} \quad \langle \mathbf{return}\ V \mid \gamma \mid ((\gamma', x, N) :: \sigma, \mathcal{L}) :: \kappa\rangle \longrightarrow \langle N \mid \gamma'[x \mapsto [\![V]\!]\gamma] \mid (\sigma, \mathcal{L}) :: \kappa\rangle$$

$$\text{M-NewCont} \quad \langle \mathbf{newcont}\ V \mid \gamma \mid \kappa\rangle \longrightarrow \langle \mathbf{return}\ x \mid \gamma[x \mapsto ([], [(\gamma, y, V\ y)])] \mid \kappa\rangle$$

$$\text{M-Resume} \quad \langle \mathbf{resume}\ \mathcal{L}\ V\ W \mid \gamma \mid \kappa\rangle \longrightarrow \langle \mathbf{return}\ W \mid \gamma \mid \kappa' +\!\!+ [(\sigma', \mathcal{L})] +\!\!+ \kappa\rangle, \quad \text{if } [\![V]\!]\gamma = (\kappa', \sigma')$$

$$\text{M-Perform} \quad \langle \mathbf{perform}\ \ell\ V \mid \gamma \mid \kappa\rangle \longrightarrow \langle \mathbf{return}\ (\mathbf{inj}_\ell\ \langle V, x\rangle) \mid \gamma[x \mapsto (\kappa', \sigma')] \mid \kappa''\rangle \quad \text{if } \kappa \text{ handles } \ell \text{ at } ((\kappa', \sigma'), \kappa'')$$

$$\text{M-RetHandler} \quad \langle \mathbf{return}\ V \mid \gamma \mid ([], \mathcal{L}) :: \kappa\rangle \longrightarrow \langle \mathbf{return}\ (\mathbf{inj}_{\text{ret}}\ V) \mid \gamma \mid \kappa\rangle$$

The first six rules are routine. We write $\ulcorner c \urcorner$ for the function that implements c on machine values.

The M-Let rule reifies a let-binding at the head of the current pure continuation. The M-RetCont rule binds a returned value in the body of the reified let-binding at the head of the current pure continuation.

The M-NewCont rule allocates a new continuation value, binding it in the environment. This continuation value simply applies the function V to its argument. The M-Resume rule resumes a continuation value by concatenating it onto the front of the continuation component of the configuration. It is now that we see why a continuation value comprises a pair of a continuation and a pure continuation. Really (κ', σ') represents a continuation $\kappa' +\!\!+ [(\sigma', X)]$ with a hole X in it that is here replaced by the effect set $\mathcal{L}$. The M-Perform rule performs an effect by reifying it as a labelled variant value containing a pair of the payload and the continuation. The auxiliary relation κ handles ℓ at $((\kappa', \sigma'), \kappa'')$ splits the current continuation κ into two parts where (κ', σ') is the continuation object up to the handler for ℓ and κ'' is the remainder of the continuation.

$$\frac{\ell \in \mathcal{L}}{(\sigma, \mathcal{L}) :: \kappa \text{ handles } \ell \text{ at } (([], \sigma), \kappa)} \qquad \frac{\ell \notin \mathcal{L} \qquad \kappa \text{ handles } \ell \text{ at } ((\kappa', \sigma'), \kappa'')}{(\sigma, \mathcal{L}) :: \kappa \text{ handles } \ell \text{ at } ((\sigma, \mathcal{L}) :: \kappa', \sigma'), \kappa'')}$$

The M-RetHandler rule reifies a top-level return as a labelled variant value with a special ret label which denotes that the computation returned normally.

Comparison with standard effect handler calculi and abstract machines. Whereas the calculus of Hillerström et al. [8] includes both deep and shallow handlers ours provides hybrid sheep handlers [22]. A deep handler automatically wraps the original handler around the body of each suspended continuation. A shallow handler does not. A sheep handler does not automatically wrap the original handler around the body of each continuation, but does require a handler to be explicitly installed whenever the continuation is resumed. Sheep handlers guarantee that some handler must be installed whenever a continuation is resumed, but not necessarily the original one.

The other substantive difference between our calculus and more classical ones like that of Hillerström et al. [8] is that although **resume** specifies the effect set for a handler, there is no special construct for specifying a handler by dispatching on the effect. Instead the result of **resume** (either a normal return or a performed effect) is wrapped up in a variant value and the dispatch is implemented using **case**.