High-level effect handlers in 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 are an expressive language feature allowing the programmer to define custom computational effects in a compositional and principled way. 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 handlers [25] are an increasingly popular programming feature that empowers programmers to define and use custom computational effects, ranging from exceptions to generators to lightweight threads, in a structured way. With an effect handler oriented programming language or library the programmer can define custom effectful operations whose semantics is specified later by a suitable *effect handler*. 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 [28], 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 effect handler libraries for C, **libhandler** [18] and **libmpeff** [19]. However, they are both geared towards compiler writers, with the explicit goal of providing a compilation target 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 small 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 systems), **libseff** supports segmented stacks for resizing 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 form 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).
- An empirical evaluation of the performance profile of **libseff** through a series of benchmarks that demonstrate that the abstraction and expressiveness offered by effect handlers can be implemented in C in concert with competitive performance (§4).

§5 discusses related work and §6 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 illustrate 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.

```
DEFINE_EFFECT(get, 0, int64_t, {});
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 do not provide a 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 within 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.

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 [28]

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* [3, 9].

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_coroutine(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. In effect a continuation is like 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 [13]. 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 [24], 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.

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.

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 id = (int64_t)param;
    for (int64_t iteration = 0; iteration < 10; iteration++) {
        printf("Worker %ld, iteration %ld\n", id, iteration);
        PERFORM(yield);
    }
    return NULL;
}
```

To run this code, we need to 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 [6, 7, 9, 24, 28, 29].

```
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; })
        }
    }
}
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 [24, 28] 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 fail. 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 seen, 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.

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

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* [13][11, 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 id is 0) we have taken care to give it the id 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;
}
```

In order to access the elements of this generator, we must define a handler for it. A more sophisticated generator library could provide iteration combinators for consuming the elements of a generator. Here we simply define a `print_all` function that prints every element produced by this 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 completely 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 }");
    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 as runtime.

As an example, we define one handler that simply prints to standard output, and another one that redirects all output to a designated buffer. However, it would be cumbersome and inefficient if we had to install a handler every time we `print`. In this case, there is a reasonable default way to implement `print`: simply send the payload to `stdout`. The **`libseff`** library supports *default handlers* [7] which are functions of type `void *(*)(void *)` that handle a given effect if no other handler is in scope. Default handlers, however, do not interrupt normal control flow of execution; instead, they are executed exactly as a normal function would, with control returning to the caller code immediately after executing the body of the handler. We now define a function `default_print` that is used as the default handler, as well as a more sophisticated handler that stores all output in a buffer instead.

```
void *default_print(void *print_payload) {
```

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

Note that the API for establishing 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 through the `EFF_PAYLOAD_T` macro, which desugars to the payload type of the given effect tag.

We can install `default_print` as a default handler by calling `seff_set_default_handler` and providing the id of the effect to be handled. For convenience, we provide the `EFF_ID` macro which expands to the 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` prints its output to the screen, whereas the call inside `with_output_to_buffer` is instead 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 [9, 17, 28] **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 active coroutine acts as the top of a linked list of 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`, which keeps track of the execution state of the coroutine and its environment as well as the set of effects that can be handled from it. More in detail, each coroutine object contains:

- The *state* of the coroutine, which can be one of `RUNNING`, `SUSPENDED` or `FINISHED`. These names are somewhat misleading and the values should be understood as preconditions to the **libseff** API: a value of `SUSPENDED` indicates that a coroutine can be resumed via `seff_handle` and a value of `RUNNING` indicates only that a coroutine can be suspended; multiple coroutines can simultaneously be in the RUNNING state even in single-threaded applications despite only one of them being executed at a given point in time (This can happen when a coroutine spawns and resumes another coroutine: at this point, both parent and child are in the RUNNING state). Similarly, the child of a coroutine in the SUSPENDED state can itself be in the RUNNING state, indicating that it could suspend.

- The set of *handled effects*, which should not be understood as the effects that can be handled by this coroutine, but instead as the effects that can be handled by suspending this coroutine.
- A pointer to a *parent coroutine* used when performing an effect, to locate the corresponding handler.
- A *resumption state* containing the execution state when the coroutine was last resumed or suspended. More precisely, when the coroutine is in the RUNNING state, this field contains the execution state of the context that last resumed it, and is used for suspending the coroutine. When the coroutine is in the SUSPENDED state, this field instead holds the execution state of the coroutine at the moment of suspending, and is used for resuming it. The specific contents of the execution context are architecture-dependent but 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 pointer to a region in the heap containing the allocated *stack space* for the coroutine. As we shall explain in more detail in §3.3, **libseff** can use multiple approaches to stack management, depending on which this 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 location of the system stack are also stored in global (thread-local, more precisely) variables. As we shall explain 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)); seff_request_t req2 = seff_handle(k2, NULL, HANDLES(eff2));
7  }
8  void main() {
9      seff_coroutine_t *k1 = seff_coroutine_new(f, NULL);
10     seff_request_t req1 = seff_handle(k1, NULL, HANDLES(eff1) | HANDLES(eff2));
11     seff_request_t req2 = seff_handle(k1, NULL, HANDLES(eff1) | HANDLES(eff2));
12 }
```

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 in the RUNNING state, with the current_coroutine variable pointing to k2. Since both k1 and k2 are currently running, the resumptions stored in them contain the program state immediately before they were started (the resumption for k1 points to the state right before the seff_handle call in line 11, and the resumption for k2 to that in 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 featured in the handled effect set of k1, so PERFORM(eff1) immediately suspends k1 and relinquishes control to its environment, which is then responsible

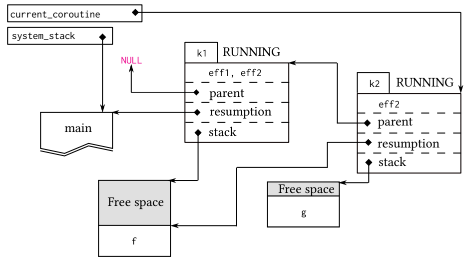


Fig. 1. Before `PERFORM(eff1)`

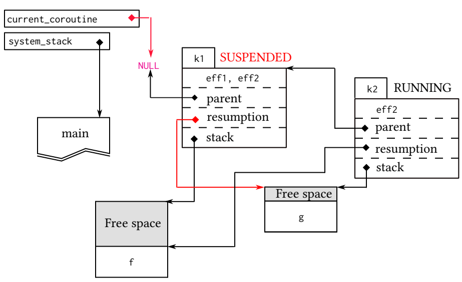


Fig. 2. After `PERFORM(eff1)`

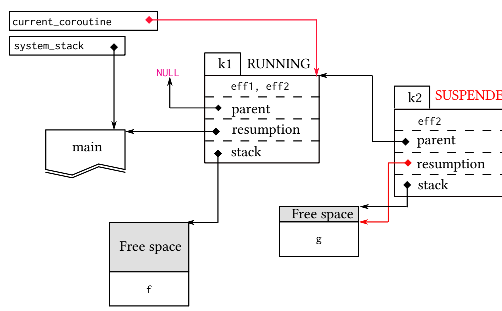


Fig. 3. After `PERFORM(eff2)`

for handling the effect. The system state at this point is depicted in Figure 2: coroutine `k1` is now suspended, and its resumption stores the program 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 considered to be in a running state.

After the effect `eff1` has been (trivially) handled, execution of the suspended `k1` is resumed in line 12 and continues until the call to `PERFORM(eff2)` is reached. At that point, the stack of active coroutines is traversed again until a handler for `eff1` is reached; in this case, this effect can be handled directly by `k2`, and so `k2` is suspended and control is transferred back to line 6 in `f`. This corresponds to the diagram in Figure 3. Note how the resumption for `k1` is updated to hold the program state of the now-paused coroutine.

In order to make it efficient to perform an effect, **`libseff`** takes particular care when passing the payload of a command 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 only two 64-bit fields, namely the tag and a pointer to the aforementioned payload, hence it can be returned from `seff_handle` directly via processor registers.

3.2 Primitives

Throughout the examples in the previous section, we have shown only the higher-level interface provided by **`libseff`**, which is intended for general use and provides convenience and some degree of compiler checking of input and output types. Internally, operations such as the PERFORM macro are implemented in terms of a simpler set of primitives, that we describe here.

At the lowest level, we have three primitives for performing context-switching. These have the single task of resuming or suspending an active coroutine and 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`. `seff_yield` is responsible for suspending the given coroutine `self` and returning control to the point where this coroutine was last resumed. The coroutine to be suspended is provided as an explicit argument to this function, and the caller is responsible for ensuring that, at the moment of invoking `seff_yield`, the coroutine to be suspended is either the current coroutine or an ancestor of it, otherwise the call to `seff_yield` will result in undefined behaviour[1]. `seff_exit` 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 does not need to be saved, and so `seff_exit` is more efficient in general.

Note that both of the functions `seff_yield`, `seff_exit` take an `effect_id` argument, which is used to construct the `seff_request_t` object that will result from suspending the coroutine. However, this argument is not 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 set of primitives is provided for looking up the appropriate handler in scope (if any) when selecting which coroutine to suspend.

```
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);
```

Dynamic dispatch is taken care of by `seff_locate_handler`, which walks through the stack of currently active coroutines until it finds the first whose `handled_effects` bitset covers the effect `effect`. As explained before, this indicates the context which resumed that coroutine last is able to handle the corresponding effect.

`seff_perform` and `seff_throw` are analogous to `seff_yield` and `seff_exit`, except that they use the given `effect_id` to select which coroutine to suspend. Effectively, `seff_perform(e, p)` is equivalent to `seff_yield(seff_locate_handler(e), e, p)`, with the only difference that, if no appropriate handler can be found in scope, `seff_perform` will invoke a default handler, whereas the `seff_yield` version will dereference a null pointer.

The PERFORM macro (illustrated in §2) is the preferred method of performing an effect. It is defined as a thin wrapper over `seff_perform`. A call to `PERFORM(eff, args...)` simply constructs a payload object of type `EFF_PAYLOAD_T(eff)` on the current stack frame and initialises it with the provided arguments, then calls `seff_perform` with the effect and a pointer to the stack-allocated payload. Unlike in other systems with resizable stacks [23, 31], **libseff** guarantees that the stack area for a given coroutine always remains at the same location, hence pointers into the stack of a coroutine will remain valid while 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

[1]It is possible to check this condition at runtime and fail gracefully if it is not satisfied, by traversing the list of coroutines and ensuring that the coroutine that is being suspended is reachable from the currently active one, but this would impose a prohibitive overhead.

different approaches, which we detail below. Currently, the first two (fixed and segmented stacks) are implemented and can be switched between via a build flag (but should not be mixed together in the same project). The third approach (overcommiting) is planned but currently unimplemented and we have deemed the last (stack copying) to be unsuitable for our target setting.

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 simple to implement and introducing no any additional runtime overhead. However, it can result in a significant waste of memory. Given that 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 against the risk of stack overflow.

3.3.2 *Segmented stacks.*

Segmented stacks, also called split stacks, replace the traditional contiguous fixed-size stack area by a linked list of stack segments or "stacklets". The compiler then instruments every function with a small prelude that checks whether the current stack is large enough to accommodate the stack frame of the current function. If not, a new stacklet is allocated to hold the new frame.

Conveniently, support for segmented stacks is provided by both GCC and Clang via the `-fsplit-stack` flag, which will add stack overflow checks to every function preamble. As shown 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 simple implementation of this routine is provided by Clang, but **libseff** defines its own instead in order to give us finer-grained control over memory allocation.

Though they enable the programmer to write code without concerning themselves with stack frame sizes, segmented stacks are not without disadvantages. If no memory needs to be allocated, the overhead of the function prelude is mostly negligible; however, 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 is sometimes known as the "hot split" problem and caused Go to move away from segmented stacks [23]. **libseff** 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.

```
lea -0x108(%rsp),%r11   } Stack overflow check
cmp %fs:0x70,%r11       }
ja 8                    }
mov $0x108,%r10d        } Segment and argument size
mov $0x0,%r11d          }
call <__morestack>
ret
push %rbp
mov %rsp,%rbp
...
pop %rbp
ret
```

```
void split_stack() {
  char buffer[256];
  ...
}
```

Fig. 4. Segmented stack prelude in Clang-12

Then, if a later function call requires the allocation of additional stack memory, this stacklet can be recycled, avoiding the need for an additional allocation. As we shall see in §4.1.2, with this optimisation, the cost of calling a function inside a hot split loop is 9x the cost of a normal function call.

We argue that, in practice, this is not a significant problem: if the cost of the hot split overhead dominates the execution time of the called function, then it is likely that this is a small function that should get inlined by the compiler. Even then, there is a lot of space 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 these cases [21].

One more 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 the user builds it from scratch with support for segmented stacks enabled) will lack this prelude and any stack overflow will cause a crash or, in the worst case scenario, silent memory corruption. To avoid this issue, Clang'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 sensible compromise, but it can lead to much higher memory consumption than necessary.

When using **libseff**, this overhead can usually be avoided: a function that was not compiled with segmented stacks enabled cannot make use of the context-switching features of `libseff`, therefore it 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:0x70, %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:0x70;"
        "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 will switch to the system stack instead of allocating a stack segment. We warn the user that this macro is only correct when the wrapped function takes all parameters and returns its result through processor registers. In addition, the wrapped function must only be called 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 for physically resizing a coroutine's stack is to use overcommitting and reserve 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 an excellent balance between performance and convenience in systems that support it. However, we intend **libseff** to be also deployable in embedded systems, which do not necessarily provide virtual memory or a large address space. Thus, while we are planning to eventually provide virtual memory-based stack management for **libseff**, it is not among our top priorities.

3.3.4 Stack copying. Finally, 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 stack copying when a resize is needed. However, it is unsuitable for a low-level language like C since the process of copying the stack necessarily invalidates any pointers into it.

Other solutions exist. Go automatically rewrites any pointers into the stack as it is being copied, but this relies on an amount of runtime information which is simply not available to a C program.

4 EVALUATION

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, 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;
}
```

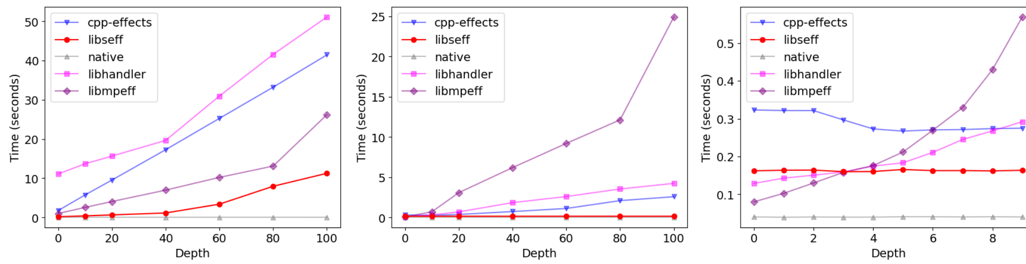
The `stateful` function recursively builds a stack of nested handlers for the `error` effect up to a specified depth. In the base case a counter, implemented using `get` and `put`, is decremented in a loop.

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.

In order 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 the `get` and `put` effects 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 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

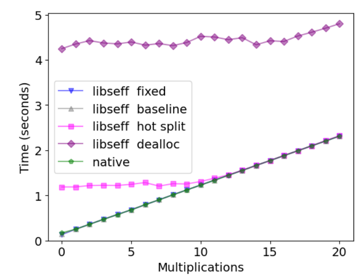


(a) General case (b) Optimal case (c) Optimal case, depths up to 10

Fig. 5. State benchmark results.

Multiplications	0	5	10	15	20
native	1.30	1.00	1.00	1.00	1.00
libseff *fixed*	1.00	1.00	1.00	1.00	1.00
libseff *baseline*	1.10	1.00	1.00	1.00	1.00
libseff *hot split*	9.00	1.83	1.06	1.00	1.00
libseff *dealloc*	32.22	6.47	3.68	2.50	2.08

(a) Relative execution time of the hot split benchmark.



(b)

Fig. 6. Hot split results

handlers or any kind of dynamic dispatching of operations; **cpp-effects** [9] is a C++ library for effect handlers; **libhandler** [17, 18] and **libmpeff** [19] are other C libraries for effect handlers.

The **cpp-effects** optimal avoids handler lookups in a similar fashion to **libseff**, but it also eliminates context switching by requiring that handler for `get` and `put` resume immediately and do 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** consistently the fastest. Figure 5b shows the optimal case. The elimination of traversing the stack of handlers gives **libseff** and **cpp-effects** a distinct advantage. For both **libhandler** and **libmpeff** the optimal case is still affected by the level of recursion. 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 depth smaller 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, becoming slower than **libseff** after depth 3.

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 and 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.

```
1  sub    $0x788,%rsp
2  movsd  0x191(%rip),%xmm0
3  movsd  0x2099(%rip),%xmm1   → Only present when m > 0
4  mulsd  %xmm0,%xmm1  }
5  ...                 } Repeated m times
6  mulsd  %xmm0,%xmm1  }
7  movsd  %xmm1,0x2041(%rip)
8  lea    -0x80(%rsp),%rax
9  add    $0x788,%rsp
10 retq
```

Fig. 7. Hot-split function body in assembly. m is the number of multiplications

Table 6a shows the results, comparing **`native`** with four different variations on **`libseff`**. *Baseline* 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).

The results show that the hot split problem is observable in **`libseff`**, causing a call to an empty function to be 9 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 10 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 a few `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 [28]. when evaluating the cost of low level operations.

It is worth noting that 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. To force the compiler to generate the code from Figure 7 without writing the assembly instructions by hand, we had to deliberately disable inlining and introduce empty inline assembly blocks so it would still compile both the function and the actual call.

4.2 Macrobenchmarks

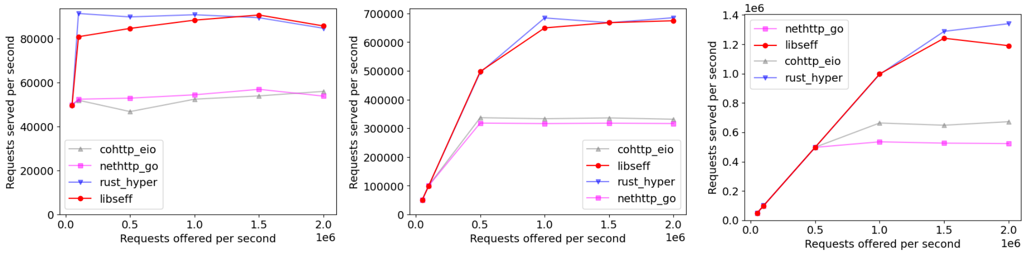
In this section we benchmark **`libseff`** against other systems running whole applications.

4.2.1 HTTP Server. Our first macrobenchmark is a simple HTTP server as used to benchmark OCaml 5 (formerly Multicore OCaml) [28, 30]. The server receives `GET` requests and respond to them asynchronously with a `text/plain` message. It serves each request with a single coroutine that is released when the connection is closed.

We compare against three alternative implementations:

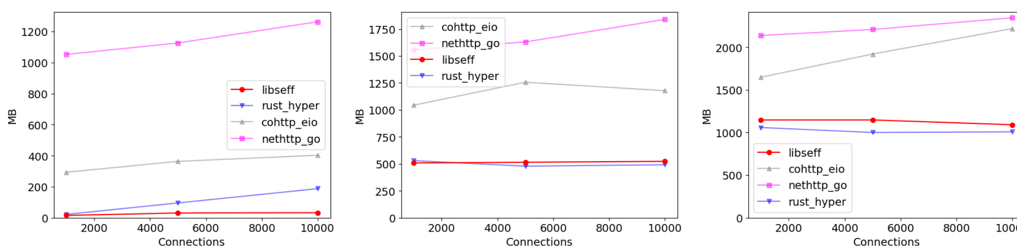
- *nethttp_go* is built using Go's `net/http` package.
- *rust_hyper* is a server built on top of Hyper, a highly performant HTTP library for Rust for the Tokio runtime, a state of the art runtime for Rust async/await concurrency.
- *cohttp_eio* is a server implemented for OCaml 5 over an effect based I/O library [29] and an HTTP library built on top of it [22].

The three variants are rather diverse in that they include an extremely simple to implement server (*nethttp_go*), a low-level highly performant server (*rust_hyper*), and a server built on top of effect handlers like ours (*cohttp_eio*).



(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 800000 requests offered per second.

Figure 8 shows the speeds running on 1, 8, and 16 OS threads. Both **libseff** and *rust_hyper* perform consistently better than *nethttp_go* and *cohttp_eio*, regardless of the number of OS threads. Figure 9 shows maximum memory consumption. We observe the maximum memory used by each implementation by varying the number of live connections, which coincide with the maximum number of coroutines spawned by each implementation.

The **libseff** implementation is built on top of a multi-threaded work-stealing scheduler based on effect handlers that takes advantage of the fact that **libseff**'s coroutines can be safely moved between OS threads. We used async I/O functions built on top of that scheduler and a small but complete HTTP request parser [26].

4.2.2 *Prefetching.* Our next benchmark, inspired by Jonathan et al. [12], uses C++'s coroutines to improve performance of memory heavy 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 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

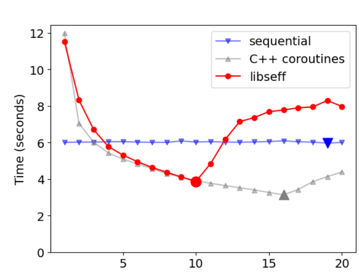


Fig. 10. Prefetch benchmark results. Large shapes mark the fastest execution for each framework.

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++'s coroutine implementation is explicit about prefetching, then yielding execution, and finally reading the memory upon return, ours considers dereferencing a memory location to be 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;
}
```

This code 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. They show that **libseff** incurs an overhead over the naïve sequential implementation whenever too few or too many streams are used but significantly improves upon it at its best point, 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 RELATED WORK

Effect handlers for C and C++. The **libhandler** [17, 18] and **libmpeff** [19] are existing effect handlers libraries for C. Unlike **libseff** they are designed as targets for compiler writers rather than for writing code directly in C. Each of these C libraries uses a different stack-management strategy: **libhandler** copies stacks into a temporary structure before restoring them on resumption, **libmpeff** uses virtual memory to allow stacks to grow without moving in memory, and **libseff** uses segmented stacks. The **cpp-effects** library [9] is a C++ effect handlers library which heavily relies on C++ features both in its implementation and its API.

Coroutines in C/C++. There exist many different coroutine libraries for C and C++, including Boost coroutines [2], **libco** [31], **libmill** [1], and C++20 stackless coroutines.

Varieties of coroutine. de Moura and Ierusalimschy [5] 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 [15] show how to translate one-shot effect handlers into asymmetric coroutines. Phipps-Costin et al. [24] exploit essentially the same encoding to implement effect handlers on top of the Wasmtime Fiber API [4], 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 about them. Nonetheless, there have also been various attempts to compile effect handlers to efficient code. Kammar et al. [13] take advantage of Haskell's aggressive inlining of type classes to speed up an implementation based on a continuation monad. Wu and Schrijvers [32] explain the essence of this optimisation as an instance of fusion. Kiselyov and Ishii [16] introduce so-called "Freer monads" as another means to speed up implementations of effect handlers in Haskell. Karachalias et al. [14] optimise effect handler code by aggressively inlining as many handlers as possible. Schuster et al. [27] achieve a similar end by way of staged computation. Xie and Leijen [33] and Ghica et al. [9] 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).

6 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** is the first library to provide an idiomatic interface for programming with effect handlers in C. The key challenge we had to overcome is C's lack of modern 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 any C developer.

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.

We are currently actively working on porting **libseff** to other architectures including ARM and 32-bit Intel processors. We also plan to support more approaches to stack management: both virtual memory and some form of arena allocation of stack segments are under consideration.

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 developing a set of Rust bindings on top of **libseff** that will leverage Rust's rich type system and borrow checker to ensure safety at compile time.

Finally, 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 promises significant additional performance for such use-cases allowing us to efficiently take fuller advantage of the expressive power of effect handlers.

REFERENCES

[1] [n. d.]. libmill. https://github.com/sustrik/libmill.

[2] Boost. [n. d.]. Boost.coroutine library. https://www.boost.org/doc/libs/1_83_0/libs/coroutine/doc/html/index.html.

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

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

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

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

[7] 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.

[8] 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.

[9] 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.

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

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

[12] 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

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

[14] 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.

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

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

[17] 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.

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

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

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

[21] 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

[22] Mirage. [n. d.]. Eio. https://github.com/mirage/ocaml-cohttp.

[23] Daniel Morsing. [n. d.]. How Stacks are Handled in Go.

[24] 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.

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

[26] The H20 Project. [n. d.]. PicoHTTPParser. https://github.com/h2o/picohttpparser.

[27] 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.

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

[29] OCaml Multicore team. [n. d.]. Eio. https://github.com/ocaml-multicore/eio.

[30] OCaml Multicore team. 2021. Multicore OCaml HTTP benchmarks. https://github.com/ocaml-multicore/retro-httpaf-bench.

[31] Tencent. [n. d.]. libco. https://github.com/Tencent/libco.

[32] 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.

[33] 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. [10] we do so by way of a CEK [8] abstract machine for a fine-grain call-by-value [20] 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. [10] is the treatment of effects and handlers. We return to these differences after presenting our calculus and abstract machine. Again following Hillerström et al. [10], 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.

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

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** ℓ 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.

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

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.

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

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. [10] 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.

$$
\begin{array}{lrl}
\textsc{M-Lam} & \langle V\ W \mid \gamma \mid \kappa\rangle \longrightarrow & \langle M \mid \gamma'[x \mapsto [\![W]\!]\gamma] \mid \kappa\rangle, \\
& & \text{if } [\![V]\!]\gamma = (\gamma', \lambda x.\, M) \\
\textsc{M-Rec} & \langle V\ W \mid \gamma \mid \kappa\rangle \longrightarrow & \langle M \mid \gamma'[f \mapsto (\gamma', \mathbf{rec}\, f\, x.M), \\
& & \quad x \mapsto [\![W]\!]\gamma] \mid \kappa\rangle, \\
& & \text{if } [\![V]\!]\gamma = (\gamma', \mathbf{rec}\, f\, x.M) \\
\textsc{M-Const} & \langle V\ W \mid \gamma \mid \kappa\rangle \longrightarrow & \langle \mathbf{return}\ (\ulcorner c \urcorner\, ([\![W]\!]\gamma)) \mid \gamma \mid \kappa\rangle, \\
& & \text{if } [\![V]\!]\gamma = c \\
\textsc{M-Split} & \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, \\
& & \text{if } [\![V]\!]\gamma = \langle v, w\rangle \\
\textsc{M-CaseMatch} & \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, \\
& & \text{if } [\![V]\!]\gamma = \mathbf{inj}_\ell\, v \\
\textsc{M-CaseDef} & \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, \\
& & \text{if } [\![V]\!]\gamma = \mathbf{inj}_{\ell'}\, v \text{ and } \ell \neq \ell' \\
\textsc{M-Let} & \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 \\
\textsc{M-RetCont} & \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
\end{array}
$$

$$
\begin{array}{lrl}
\textsc{M-NewCont} & \langle \mathbf{newcont}\ V \mid \gamma \mid \kappa\rangle \longrightarrow & \langle \mathbf{return}\ x \mid \gamma[x \mapsto ([], [(\gamma, y, V\ y)])] \mid \kappa\rangle \\
\textsc{M-Resume} & \langle \mathbf{resume}\ \mathcal{L}\ V\ W \mid \gamma \mid \kappa\rangle \longrightarrow & \langle \mathbf{return}\ W \mid \gamma \mid \kappa' \mathbin{+\!\!+} [(\sigma', \mathcal{L})] \mathbin{+\!\!+} \kappa\rangle, \\
& & \text{if } [\![V]\!]\gamma = (\kappa', \sigma') \\
\textsc{M-Perform} & \langle \mathbf{perform}\ \ell\ V \mid \gamma \mid \kappa\rangle \longrightarrow & \langle \mathbf{return}\ (\mathbf{inj}_\ell\ \langle V, x\rangle) \mid \gamma[x \mapsto (\kappa', \sigma')] \mid \kappa''\rangle \\
& & \text{if } \kappa \text{ handles } \ell \text{ at } ((\kappa', \sigma'), \kappa'') \\
\textsc{M-RetHandler} & \langle \mathbf{return}\ V \mid \gamma \mid ([], \mathcal{L}) :: \kappa\rangle \longrightarrow & \langle \mathbf{return}\ (\mathbf{inj}_{\mathsf{ret}}\ V) \mid \gamma \mid \kappa\rangle
\end{array}
$$

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 $\kappa' \mathbin{+\!\!+} [(\sigma', X)]$ 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. [10] includes both deep and shallow handlers ours provides hybrid sheep handlers [24]. 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. [10] 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**.