01.Nov 2013

Effect Handlers in Low-Level Languages
Challenges and Opportunities

Programming Language Team in Edinburgh

www.huawei.com

Using effect handlers from C

- **Why?**
  - Effect handlers provide: green threads, actors, generators, exceptions
  - C: only **modern** language missing **all** of these features
  - Therefore: C stands to benefit **the most**!

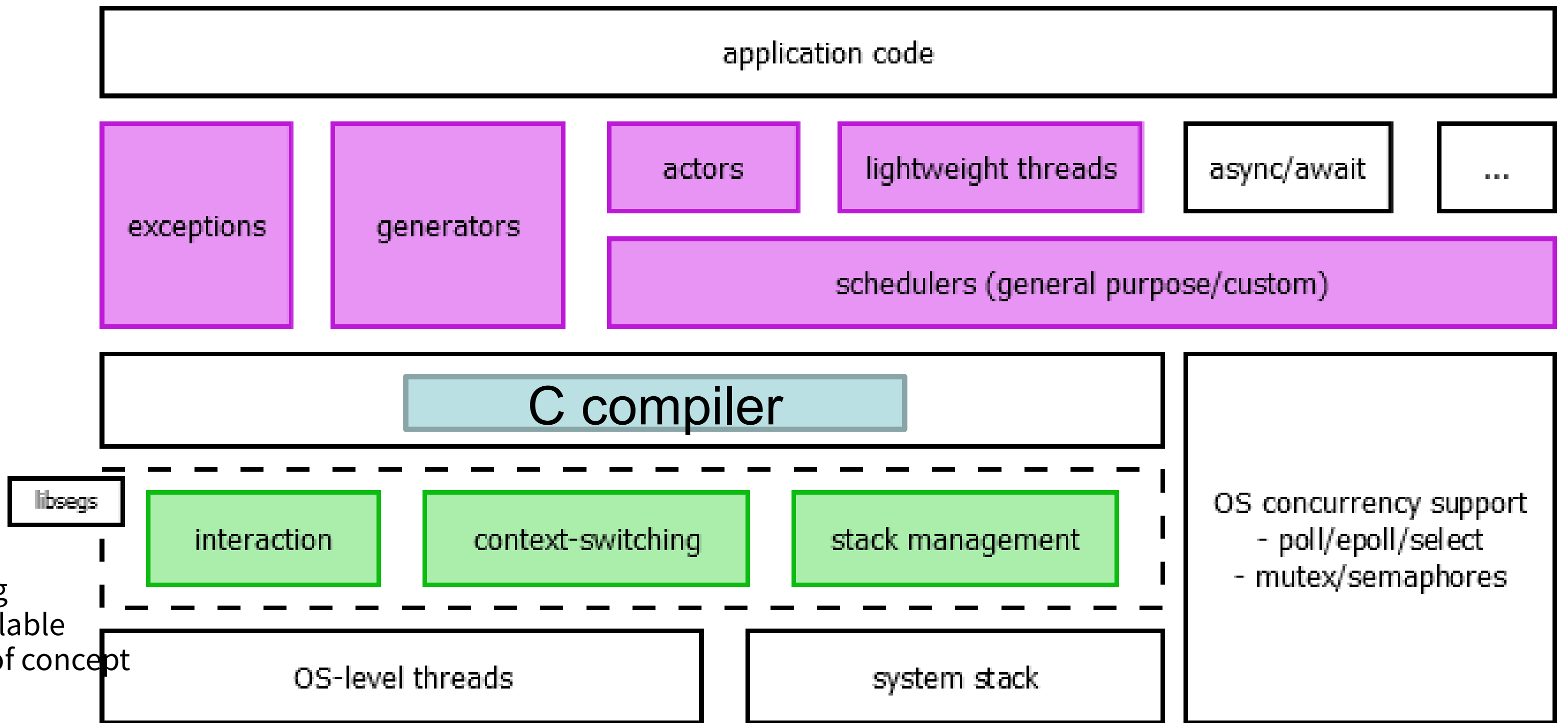
Using effect handlers from C

- **Why?**
  - Effect handlers provide: green threads, actors, generators, exceptions
  - C: only **modern** language missing **all** of these features
  - Therefore: C stands to benefit **the most**!

- **Ok, but why, really?**
  - Tons of C code in use at Huawei
  - Many projects re-invent concurrency! (Coroutines/actors built on setjmp/longjmp)
  - Main goal: use effect handlers to provide **lightweight, modular** concurrency features for C
  - Main goal: effect handlers should be **compatible with every C feature** (stack stability)
  - Main goal: effect handlers should be **ergonomic** to use **by hand**
  - Non-goal: use effect handlers to structure effectful computation
  - Non-goal (for now): statically enforce runtime safety

Stackful coroutines in C

- Offer coroutine support through **`libseff`** library (https://gitee.com/marioalvarezpicallo/libseff)
- Prototype implementation in major compilers gcc & clang
- Compiler can provide **extra support, optimizations** & **better syntax**
- Small asm part needs to be ported to different architectures, **rest is architecture-independent**
- **Effects** = **stackful coroutines + dynamic binding** (corollary: C programmers are **not scared**)



Green: working prototype available
Purple: proof of concept available

Effect example – Defining and Performing

```
DEFINE_EFFECT(print, 0, void, { char *str; });

DEFINE_EFFECT(get, 1, int64_t, {});
DEFINE_EFFECT(put, 2, void, { int64_t value; });

void *effectful(int64_t N) {
    for (int64_t i = 0; i < N; i++) {
        int64_t state = PERFORM(get);
        char str[256];
        sprintf(str, "State is %ld", state);
        PERFORM(print, str);
        state = state * state + 1;
        PERFORM(put, state);
    }
    return NULL;
}
```

Effect example – Defining and Performing

```
DEFINE_EFFECT(print, 0, void, { char *str; })

DEFINE_EFFECT(get, 1, int64_t, {});
DEFINE_EFFECT(put, 2, void, { int64_t value; });

void *effectful(int64_t N) {
    for (int64_t i = 0; i < N; i++) {
        int64_t state = PERFORM(get);
        char str[256];
        sprintf(str, "State is %ld", state);
        PERFORM(print, str);
        state = state * state + 1;
        PERFORM(put, state);
    }
    return NULL;
}
```

DEFINE_EFFECT macro defines a single operation, a la OCaml

Effect example – Defining and Performing

```c
DEFINE_EFFECT(print, 0, void, { char *str; });

DEFINE_EFFECT(get, 1, int64_t, {});
DEFINE_EFFECT(put, 2, void, { int64_t value; });

void *effectful(int64_t N) {
    for (int64_        < N; i++) {
        int64_t              ERFORM(get
        char str[256];
        sprintf(str, "State is %ld", state);
        PERFORM(print, str);
        state = state * state + 1;
        PERFORM(put, state);
    }
    return NULL;
}
```

Return type

Parameter types
(expressed as a struct)

Effect example – Defining and Performing

Programmer must specify effect ID (0 – 63)

```
DEFINE_EFFECT(print, 0, void, { char *str; });

DEFINE_EFFECT(get, 1, int64_t, {});
DEFINE_EFFECT(put, 2, void, { int64_t value; });

void *effectful(int64_t N) {
    for (int64_t i = 0; i < N; i++) {
        int64_t state = PERFORM(get);
        char str[256];
        sprintf(str, "State is %ld", state);
        PERFORM(print, str);
        state = state * state + 1;
        PERFORM(put, state);
    }
    return NULL;
}
```

Effect example – Defining and Performing

```c
DEFINE_EFFECT(print, 0, void, { char *str; });

DEFINE_EFFECT(get, 1, int64_t, {});
DEFINE_EFFECT(put, 2, void, { int64_t value; });

void *effectful(int64_t N) {
    for (int64_t i = 0; i < N; i++) {
        int64_t state = PERFORM(get);
        char str[256];
        sprintf(str, "State is %ld", state);
        PERFORM(print, str);
        state = state * state + 1;
        PERFORM(put, state);
    }
    return NULL;
}
```

Effects are performed via PERFORM macro, this is **relatively** type-safe

Pass pointer to stack-allocated `str` to effect handler – this is sound because of **stack stability**

Effect example – Handling

```
void *handle_state(closure *clo) {
    seff_coroutine_t *k = seff_coroutine_new(clo->fn, clo->arg);
    effect_set handled = HANDLES(get) | HANDLES(put);
    int64_t state = 0;
    seff_request_t req = seff_handle(k, NULL, handled);
    while (true) {
        switch (req.effect) {
            CASE_EFFECT(req, get, {
                req = seff_handle(k, (void *)state, handled);
                break;
            });
            CASE_EFFECT(req, put, {
                state = payload.value;
                req = seff_handle(k, NULL, handled);
                break;
            });
            CASE_RETURN(req, {
                seff_coroutine_delete(k);
                return payload.result;
            });
        }
    }
}
```

- We often abstract handling to a handle_XYZ function
- Function runs a main loop dispatching on effect
- C lacks closures, we use ad-hoc closure types

Effect example – Handling

```
void *handle_state(closure *clo) {
    seff_coroutine_t *k = seff_coroutine_new(clo->fn, clo->arg);
    effect_set handled = HANDLES(get) | HANDLES(put);
    int64_t state = 0;
    seff_request_t req = seff_handle(k, NULL, handled);
    while (true) {
        switch (req.effect) {
            CASE_EFFECT(req, get, {
                req = seff_handle(k, (void *)state, handled);
                break;
            });
            CASE_EFFECT(req, put, {
                state = payload.value;
                req = seff_handle(k, NULL, handled);
                break;
            });
            CASE_RETURN(req, {
                seff_coroutine_delete(k);
                return payload.result;
            });
        }
    }
}
```

Effectful functions must be instantiated as **coroutines**: allocate stack space, metadata. Lifetime often managed by handler

Effect example – Handling

```
void *handle_state(closure *clo) {
    seff_coroutine_t *k = seff_coroutine_new(clo->fn, clo->arg);
    effect_set handled = HANDLES(get) | HANDLES(put);
    int64_t state = 0;
    seff_request_t req = seff_handle(k, NULL, handled);
    while (true) {
        switch (req.effect) {
            CASE_EFFECT(req, get, {
                req = seff_handle(k, (void *)state, handled);
                break;
            });
            CASE_EFFECT(req, put, {
                state = payload.value;
                req = seff_handle(k, NULL, handled);
                break;
            });
            CASE_RETURN(req, {
                seff_coroutine_delete(k);
                return payload.result;
            });
        }
    }
}
```

`seff_handle` starts/resumes a coroutine, returns a reified request object (can specify return value or performed command)

`CASE_EFFECT` macro checks effect tag, unpacks request into (typed) payload variable

Coroutine `return` treated as a special effect with a single `result` argument

Effect example – Handling

```
void *handle_print(closure *clo) {
    seff_coroutine_t *k = seff_coroutine_new(clo->fn, clo->arg);
    effect_set handled = HANDLES(print);
    seff_request_t req = seff_handle(k, NULL, handled);
    while (true) {
        switch (req.effect) {
            CASE_EFFECT(req, print, {
                puts(payload.str);
                req = seff_handle(k, NULL, handled);
                break;
            });
            CASE_RETURN(req, {
                seff_coroutine_delete(k);
                return payload.result;
            });
        }
    }
}

int main(void) {
    closure closure_1 = (closure){effectful, (void *)10};
    closure closure_2 = (closure){handle_state, (void *)&closure_1};
    handle_print(&closure_2);
}
```

Lack of closures makes composition awkward. Code is equivalent to

```
handle_print(() => {
    handle_state(() => { effectful(10) })
})
```

Effect example

Coroutine API

- The **coroutine** is the fundamental abstraction of **`libseff (no resumptions/continuations)`**
- The stack frame of any function executing inside the coroutine lives in the coroutine's memory
- Once created, a coroutine may be **resumed**
- Inside a running coroutine, any function may **yield** and provide some information to the context
- Coroutines are **thread-safe** and can be sent between threads to achieve e.g. work-stealing

```
typedef struct { ... } coroutine_t;

typedef void *seff_start_fun_t(void *);

seff_coroutine_t *seff_coroutine_new(seff_start_fun_t *, void
*);
void seff_coroutine_delete(seff_coroutine_t *);

bool seff_coroutine_init(seff_coroutine_t *, fun_t *, void *);
bool seff_coroutine_release(seff_coroutine_t *);
```

Coroutine API

- The **coroutine** is the fundamental abstraction of **`libseff (no resumptions/continuations)`**
- The stack frame of any function executing inside the coroutine lives in the coroutine's memory
- Once created, a coroutine may be **resumed**
- Inside a running coroutine, any function may **yield** and provide some information to the context
- Coroutines are **thread-safe** and can be sent between threads to achieve e.g. work-stealing

```
typedef struct { ... } coroutine_t;

typedef void *seff_start_fun_t(void *);

seff_coroutine_t *seff_coroutine_new(seff_start_fun_t *, void
*);
void seff_coroutine_delete(seff_coroutine_t *);

bool seff_coroutine_init(seff_coroutine_t *, fun_t *, void *);
bool seff_coroutine_release(seff_coroutine_t *);
```

- Coroutine & stacklet can be dynamically allocated or programmer can provide memory block
- Implementation is **untyped** (input/return is `void *`)

Coroutine API

- The **coroutine** is the fundamental abstraction of **`libseff (no resumptions/continuations)`**
- The stack frame of any function executing inside the coroutine lives in the coroutine's memory
- Once created, a coroutine may be **resumed**
- Inside a running coroutine, any function may **yield** and provide some information to the context
- Coroutines are **thread-safe** and can be sent between threads to achieve e.g. work-stealing

```
void *seff_yield(coroutine_t *, void *);
void *seff_resume(coroutine_t *, void *);

typedef struct {
    effect_id id;
    void *payload;
} seff_request_t;

void *seff_perform(effect_id, void *);
seff_request_t seff_handle(coroutine_t *, void *, effect_set);
```

  - Handlers are **shallow** (technically **sheep**) – see example later
  - Helper macros DEFINE_EFFECT, PERFORM, CASE_EFFECT buy us *some* type-safety/convenience

Coroutine API

- The **coroutine** is the fundamental abstraction of **`libseff (no resumptions/continuations)`**
- The stack frame of any function executing inside the coroutine lives in the coroutine's memory
- Once created, a coroutine may be **resumed**
- Inside a running coroutine, any function may **yield** and provide some information to the context
- Coroutines are **thread-safe** and can be sent between threads to achieve e.g. work-stealing

```
void *seff_yield(coroutine_t *, void *);
void *seff_resume(coroutine_t *, void *);

typedef struct {
    effect_id id;
    void *payload;
} seff_request_t;

void *seff_perform(effect_id, void *);
seff_request_t seff_handle(coroutine_t *, void *, effect_set);
```

Coroutine-like API (similar to e.g. libco), can yield to any "parent" coroutine (not checked)

Effects are **reified** as request objects, fit in 2 registers

We use 64-bit bitsets for effects, max 64 definable commands

Effect-like API: do not yield to specific coroutine, instead search active coroutine stack for installed handler

- Handlers are **shallow** (technically **sheep**) – see example later
- Helper macros `DEFINE_EFFECT`, `PERFORM`, `CASE_EFFECT` buy us *some* type-safety/convenience

Coroutine API

- The **coroutine** is the fundamental abstraction of **`libseff (no resumptions/continuations)`**
- The stack frame of any function executing inside the coroutine lives in the coroutine's memory
- Once created, a coroutine may be **resumed**
- Inside a running coroutine, any function may **yield** and provide some information to the context
- Coroutines are **thread-safe** and can be sent between threads to achieve e.g. work-stealing

```
void *seff_yield(seff_coroutine_t *, effect_id, void *);
seff_request_t seff_resume(seff_coroutine_t *, void *);

typedef struct {
    effect_id id;
    void *payload;
} seff_request_t;
```

Coroutine API actually uses effect request structs, programmer can pass additional data in effect field

```
void *seff_perform(effect_id, void *);
seff_request_t seff_handle(coroutine_t *, void *, effect_set);
```

- Handlers are **shallow** (technically **sheep**) – see example later
- Helper macros `DEFINE_EFFECT`, `PERFORM`, `CASE_EFFECT` buy us *some* type-safety/convenience

Context Switching

```
DEFINE_EFFECT(eff1,   , void, {});
DEFINE_EFFECT(eff2,   , void, {});
void *g(void *arg) { PERFORM(eff1); PERFORM(eff2); }
void *f(void *arg) {
    seff_coroutine_t *k2 = seff_coroutine_new(g, NULL);
    seff_request_t req1 = seff_handle(k2, NULL, HANDLES(eff2));
    seff_request_t req2 = seff_handle(k2, NULL, HANDLES(eff2));
}
void main() {
    seff_coroutine_t *k1 = seff_coroutine_new(f, NULL);
    seff_request_t req1 = seff_handle(k1, NULL, HANDLES(eff1) | HANDLES(eff2));
    seff_request_t req2 = seff_handle(k1, NULL, HANDLES(eff1) | HANDLES(eff2));
}
```

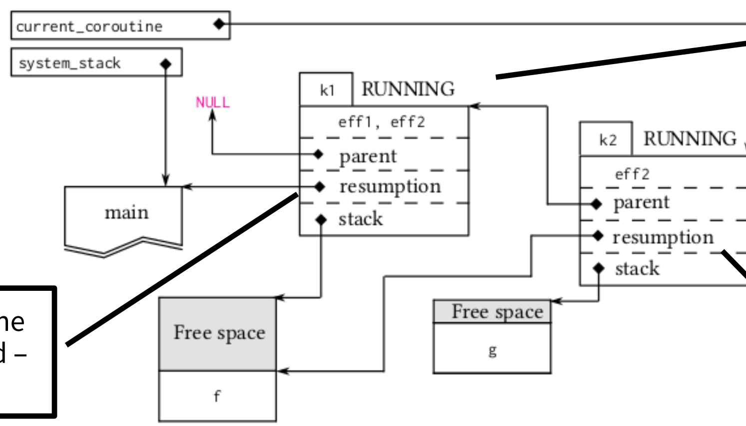
A coroutine contains

- A pointer to the **stack frame**
- A pointer to a **parent coroutine**
- A **resumption** (saved register state)
- Bitset of **handled effects**

Thread-local metadata

- Address of **top of system stack**
- Pointer to **currently executing coroutine**

Note that both coroutines are RUNNING since technically k1 never yielded



Resumptions are stored in the coroutine header instead of being passed around – helps avoid allocation

Fig. 1. Before PERFORM(eff1)

In a running coroutine the resumption indicates **where to yield**

Context Switching

```
DEFINE_EFFECT(eff1, void, {});
DEFINE_EFFECT(eff2, void, {});
void *g(void *arg) { PERFORM(eff1); PERFORM(eff2); }
void *f(void *arg) {
    seff_coroutine_t *k2 = seff_coroutine_new(g, NULL);
    seff_request_t req1 = seff_handle(k2, NULL, HANDLES(eff2));
    seff_request_t req2 = seff_handle(k2, NULL, HANDLES(eff2));
}
void main() {
    seff_coroutine_t *k1 = seff_coroutine_new(f, NULL);
    seff_request_t req1 = seff_handle(k1, NULL, HANDLES(eff1) | HANDLES(eff2));
    seff_request_t req2 = seff_handle(k1, NULL, HANDLES(eff1) | HANDLES(eff2));
}
```

A coroutine contains

- A pointer to the **stack frame**
- A pointer to a **parent coroutine**
- A **resumption** (saved register state)
- Bitset of **handled effects**

Thread-local metadata

- Address of **top of system stack**
- Pointer to **currently executing coroutine**

Outside the scope of any handler, `current_coroutine` is NULL

K1 is suspended, its resumption now indicates **where to resume computation**

Only k1 can handle `eff1`, so we suspend it and return the request to its handler

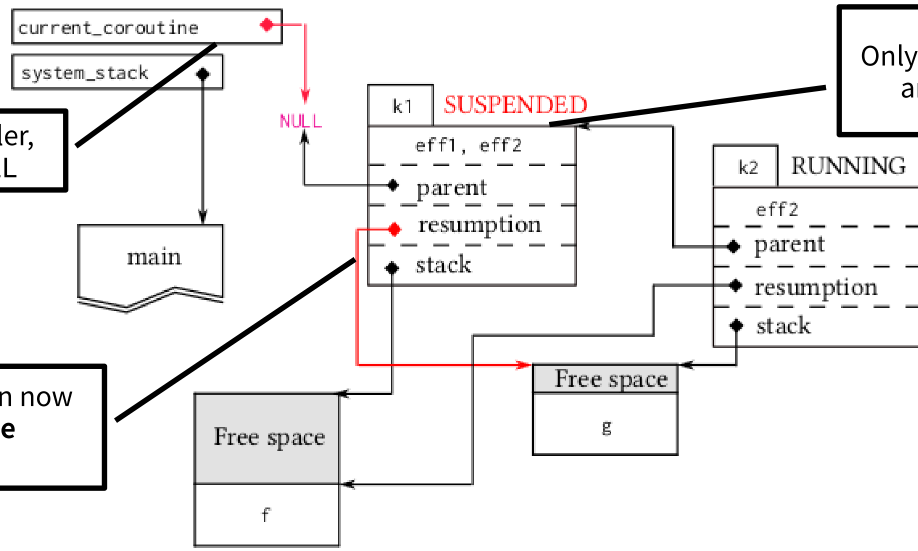


Fig. 2. After PERFORM(eff1)

Context Switching

```c
DEFINE_EFFECT(eff1, 0, void, {});
DEFINE_EFFECT(eff2, 1, void, {});
void *g(void *arg) { PERFORM(eff1); PERFORM(eff2); }
void *f(void *arg) {
    seff_coroutine_t *k2 = seff_coroutine_new(g, NULL);
    seff_request_t req1 = seff_handle(k2, NULL, HANDLES(eff2));
    seff_request_t req2 = seff_handle(k2, NULL, HANDLES(eff2));
}
void main() {
    seff_coroutine_t *k1 = seff_coroutine_new(f, NULL);
    seff_request_t req1 = seff_handle(k1, NULL, HANDLES(eff1) | HANDLES(eff2));
    seff_request_t req2 = seff_handle(k1, NULL, HANDLES(eff1) | HANDLES(eff2));
}
```

A coroutine contains

- A pointer to the **stack frame**
- A pointer to a **parent coroutine**
- A **resumption** (saved register state)
- Bitset of **handled effects**

Thread-local metadata

- Address of **top of system stack**
- Pointer to **currently executing coroutine**

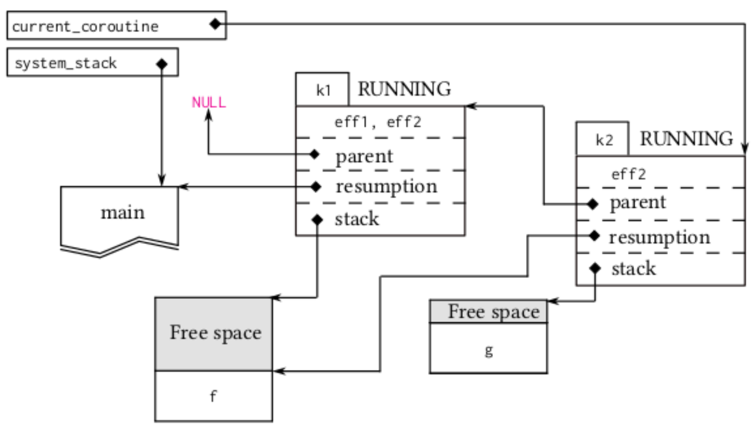


Fig. 3 Before PERFORM(eff 2

Context Switching

```
DEFINE_EFFECT(eff1, 0, void, {});
DEFINE_EFFECT(eff2, 1, void, {});
void *g(void *arg) { PERFORM(eff1); PERFORM(eff2); }
void *f(void *arg) {
    seff_coroutine_t *k2 = seff_coroutine_new(g, NULL);
    seff_request_t req1 = seff_handle(k2, NULL, HANDLES(eff2));
    seff_request_t req2 = seff_handle(k2, NULL, HANDLES(eff2));
}
void main() {
    seff_coroutine_t *k1 = seff_coroutine_new(f, NULL);
    seff_request_t req1 = seff_handle(k1, NULL, HANDLES(eff1) | HANDLES(eff2));
    seff_request_t req2 = seff_handle(k1, NULL, HANDLES(eff1) | HANDLES(eff2));
}
```

A coroutine contains

- A pointer to the **stack frame**
- A pointer to a **parent coroutine**
- A **resumption** (saved register state)
- Bitset of **handled effects**

Thread-local metadata

- Address of **top of system stack**
- Pointer to **currently executing coroutine**

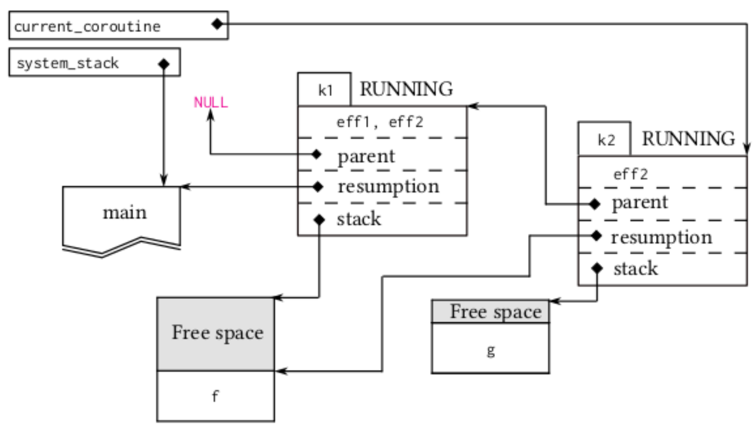


Fig. 3 Before PERFORM(eff 2

Context Switching

```c
DEFINE_EFFECT(eff1, 0, void, {});
DEFINE_EFFECT(eff2, 1, void, {});
void *g(void *arg) { PERFORM(eff1); PERFORM(eff2); }
void *f(void *arg) {
    seff_coroutine_t *k2 = seff_coroutine_new(g, NULL);
    seff_request_t req1 = seff_handle(k2, NULL, HANDLES(eff2));
    seff_request_t req2 = seff_handle(k2, NULL, HANDLES(eff2));
}
void main() {
    seff_coroutine_t *k1 = seff_coroutine_new(f, NULL);
    seff_request_t req1 = seff_handle(k1, NULL, HANDLES(eff1) | HANDLES(eff2));
    seff_request_t req2 = seff_handle(k1, NULL, HANDLES(eff1) | HANDLES(eff2));
}
```

A coroutine contains

- A pointer to the **stack frame**
- A pointer to a **parent coroutine**
- A **resumption** (saved register state)
- Bitset of **handled effects**

Thread-local metadata

- Address of **top of system stack**
- Pointer to **currently executing coroutine**

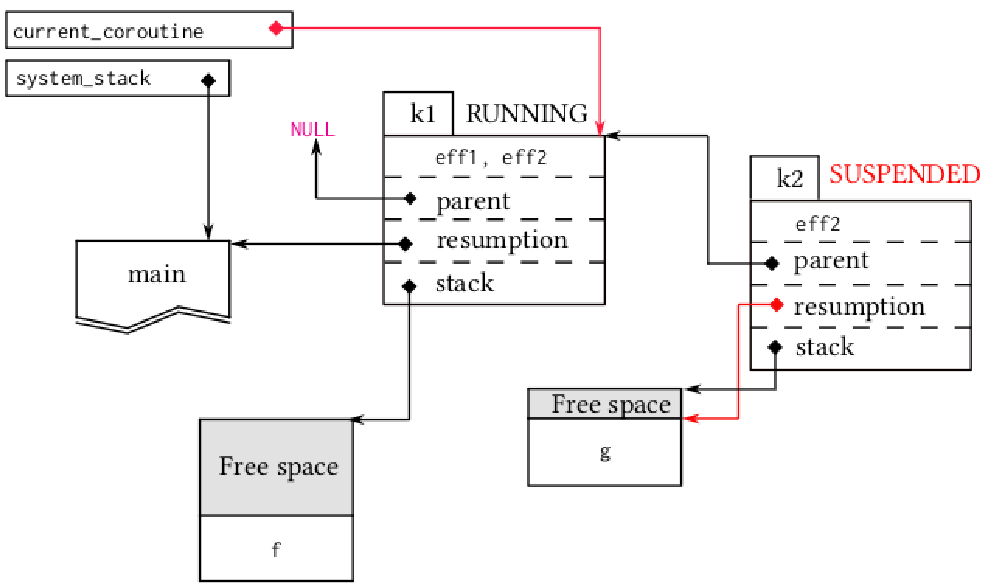


Fig. 3. After PERFORM(eff2)

Context Switching

```c
DEFINE_EFFECT(eff1, 0, void, {});
DEFINE_EFFECT(eff2, 1, void, {});
void *g(void *arg) { PERFORM(eff1); PERFORM(eff2); }
void *f(void *arg) {
    seff_coroutine_t *k2 = seff_coroutine_new(g, NULL);
    seff_request_t req1 = seff_handle(k2, NULL, HANDLES(eff2));
    seff_request_t req2 = seff_handle(k2, NULL, HANDLES(eff2));
}
void main() {
    seff_coroutine_t *k1 = seff_coroutine_new(f, NULL);
    seff_request_t req1 = seff_handle(k1, NULL, HANDLES(eff1) | HANDLES(eff2));
    seff_request_t req2 = seff_handle(k1, NULL, HANDLES(eff1) | HANDLES(eff2));
}
```

A coroutine contains

- A pointer to the **stack frame**
- A pointer to a **parent coroutine**
- A **resumption** (saved register state)
- Bitset of **handled effects**

Thread-local metadata

- Address of **top of system stack**
- Pointer to **currently executing coroutine**

- Single resumption rather than separate yield/resume points
  - Saves space on coroutine metadata at the cost of more time swapping registers – **actually no overhead**
- Effect payload is **allocated on the stack**
  - Perform/resume incur **zero allocations**

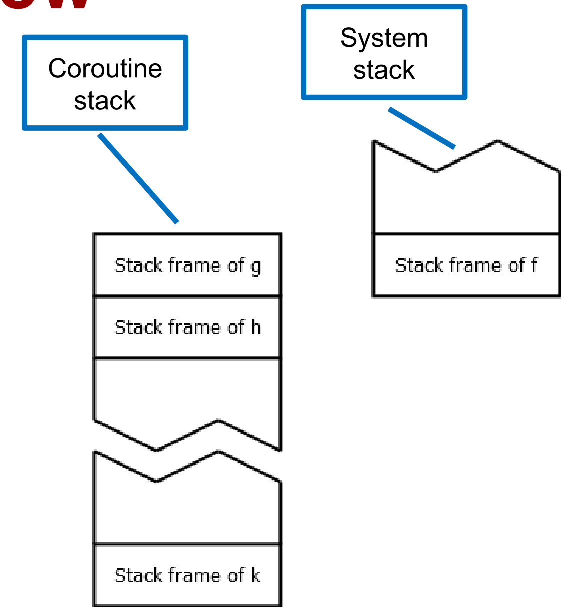
```c
PERFORM(put, 10);
```

```c
put_payload_t payload = (put_payload_t) { 10 };
seff_perform(put_eff_id, &payload);
```

  - Effect tag & payload pointer passed through

Stackful coroutines: an overview

- Commonly implemented with runtime support
  - Lua (coroutines, built-in)
  - Go (goroutines, built-in)
  - Java (virtual threads, built-in since Java 19)
  - C++ ([Boost::Coroutine](), implemented as a library)
  - Rust ([may](), implemented as a library)
  - Erlang (processes, built-in)
- Allocate **entire stack** (not just one frame) for coroutine
  - Stack space can be allocated in heap, global memory, or anywhere
- All calls inside coroutine use coroutine stack
- **Any** function within the coroutine may yield
- Can use **static-sized** stacks or **growable** stacks
  - Growable stacks need more runtime support
- No difference between sync/async functions
  - All functions can call async functions



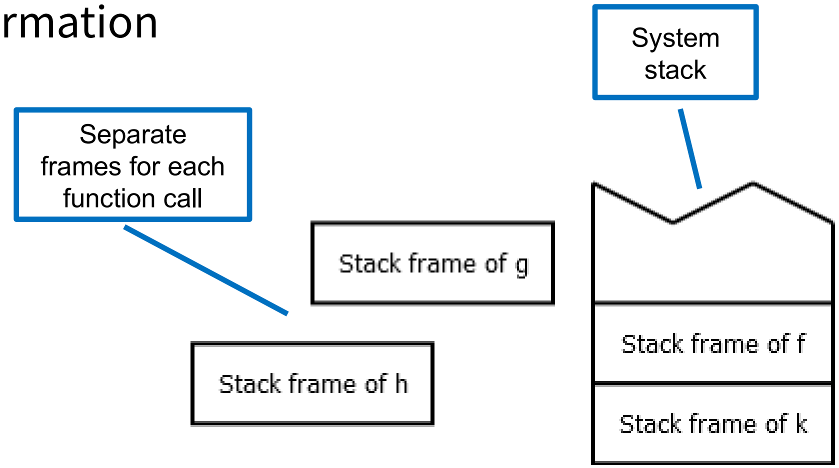
No distinction between sync/async

```
int k() { yield; return 1; }
int h() { yield; return k(); }
int g() { h(); }

void f() {
    coroutine* coro =
create_coroutine(g);
}
```

Stackless concurrency: an overview

- Commonly implemented via compiler transformation
  - C++ (C++20 coroutines/libcoro)
  - Rust
  - Kotlin
  - Swift
  - Javascript
- Create **single stack frame** for coroutine
  - Frame can be allocated anywhere
  - Function is transformed into state machine
- Calls inside coroutine use system stack
- Can only yield from top-level function
  - Can yield from nested coroutine with special **`await`** syntax
  - Without complex optimizations, nesting coroutines can be very expensive! (one allocation per coroutine call, chaining yields…)
- Async functions are **special**
  - E.g. cannot be used as function pointers



Can only use **`yield`** in **`async`** functions

```
int k() { yield; return 1; }
int h() async { yield; return k(); }
int g() async { await h(); }

void f() {
    coroutine* coro = g();
}
```

Hypothetical syntax for stackless coroutines in C

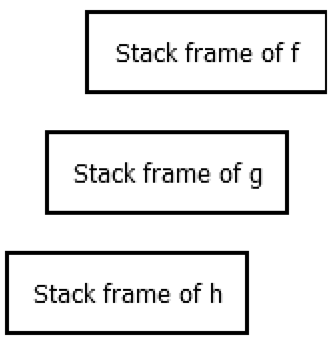
- `yield` *for pausing the current coroutine*
- `await` *for nesting coroutine calls*
- `async` *for marking coroutine functions*

Stackful challenges

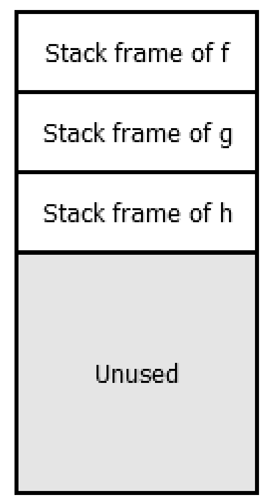
- Needs architecture-specific support (portable C library, binds to small platform-dependent asm)
- More complex stack management
  - Resizable stacks
    - Virtual mem
    - Stack copying

      Not suitable for low-level!
    - Segmented stacks — Complex, some runtime overhead
  - Fixed-size stacks — Some memory waste, no recursion
- Cost of context switch — 20~30 µinstructions
- **Less efficient use of memory**

Stackless:
- Each frame is allocated independently
- More allocations
- But less memory usage!




```
int h() async {
  yield; return 1;
}
int g() async {
  yield; await
h();
}
int f() async {
  await g();
}
```



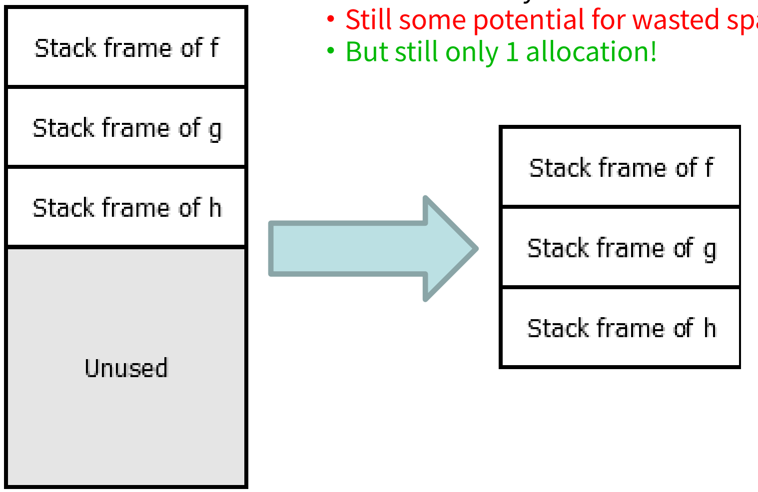
Stackful:
- All frames stored in a single memory block
- Some wasted space
- But only 1 allocation!

HUAWEI

Stackful challenges

- Needs architecture-specific support (portable C library, binds to small platform-dependent asm)
- More complex stack management
- Cost of context switch
- **Less efficient use of memory**
  - Many optimizations are possible for stackful

```
int h() {
    yield;
    return 1;
}
int g() {
    yield;
    h();
}
int f() {
    g();
}
```



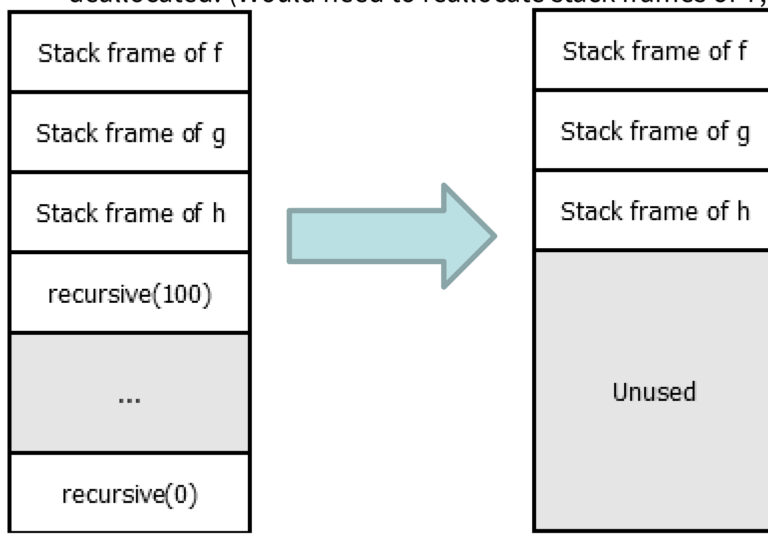
- When compiler can determine max stack frame size, can allocate exactly what is needed!
- Still some potential for wasted space
- But still only 1 allocation!

Stackful challenges

- Needs architecture-specific support (portable C library, binds to small platform-dependent asm)
- More complex stack management
- Cost of context switch
- **Less efficient use of memory**
  - Many optimizations are possible for stackful

- If compiler can determine max stack frame size, can allocate exactly what is needed!
- **Still some potential for wasted space**
  - After recursive function ends, stack frames are removed but memory cannot be easily deallocated! (Would need to reallocate stack frames of f, g, h)

```
int rec(int n) {
   ... rec(n-1);
}
int h() {
    yield;
    recursive(100);
    yield;
}
int g() {
    yield; h();
}
int f() {
    yield; g();
}
```



- Need to allocate stack space for recursive call
- After recursive call, space is not freed, but will not be used!

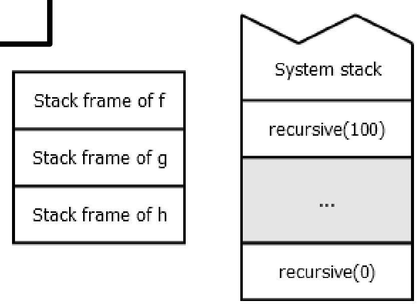
Stackful challenges

- Needs architecture-specific support (portable C library, binds to small platform-dependent asm)
- More complex stack management
- Cost of context switch
- **Less efficient use of memory**
  - Many optimizations are possible for stackful

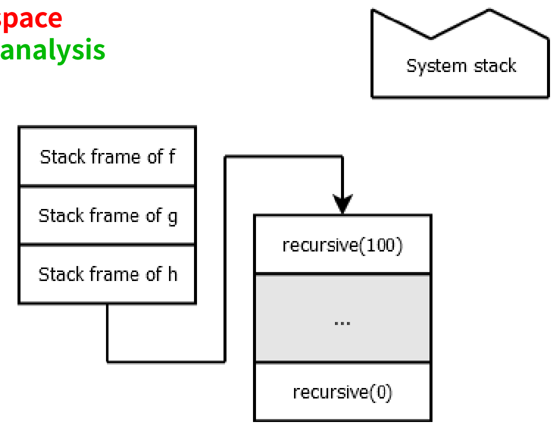
```
int recur(int n) {
    ... recur(n-1);
}
int h() {
    yield;
    recursive(100);
    yield;
}
int g() {
    yield; h();
}
int f() {
    yield; g();
}
```

No **yield** → can run on system stack

- If compiler can determine max stack frame size, can allocate exactly what is needed!
- **Still some potential for wasted space**
- **Can be mitigated with compiler analysis**



Run `recur` on the system stack
- No need for a large coroutine stack!
- But depends on compiler analysis



Use segmented stack
- Separate segments for different function calls
- Deallocate/reuse when finished
- Some runtime penalty

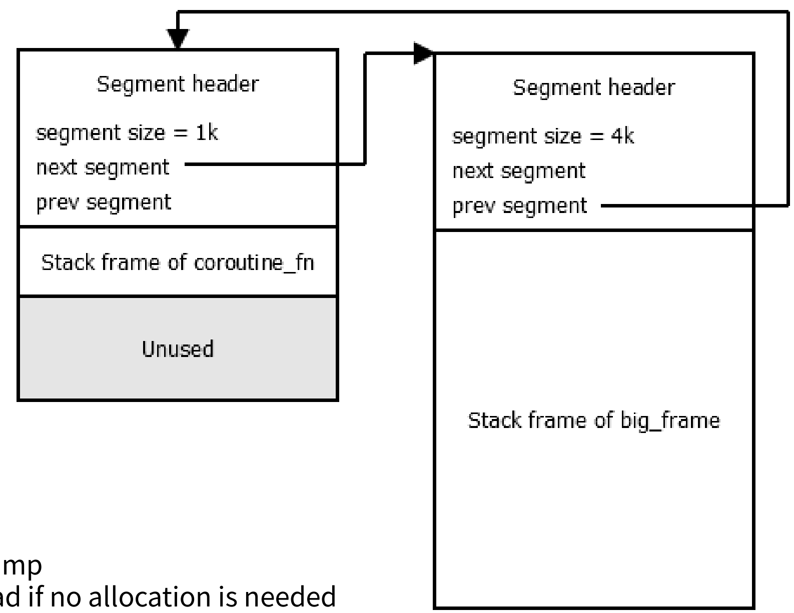
- Both approaches can be combined
- **Use compiler analysis to decide strategy**
- **If call tree is known at compile time, memory usage can be optimal**

Stack handling

- **`libseff`** uses **segmented stacks** for stack handling
  - But can easily be adapted to **stack copying** or **virtual memory** if the architecture supports it!
- Coroutines are given an **initial stack** (size can be chosen by the programmer)
- Every function call **checks available stack space** vs **function stack frame size**
  - If not enough available, **new segment is allocated**
  - The check and allocation are inserted automatically by compiler (clang & gcc –fsplit–stack support)

```
int big_frame() {
    int array[1024];
    ...
    return 0;
}
int coroutine_fn() {
    big_frame();
}
```

- The current stack segment is too small for the big array
- `big_frame` checks stack size in **function prelude**, creates new stack segment in doubly-linked list




```
big_frame():
        lea     r11, [rsp - 4104]
        cmp     r11, qword ptr fs:[112]
        ja      .LBB0_0
        mov     r10d, 4104
        mov     r11d, 0
        call    __morestack
        ret
.LBB0_0:
        …
        ret
```

Prelude: check stack size & allocate

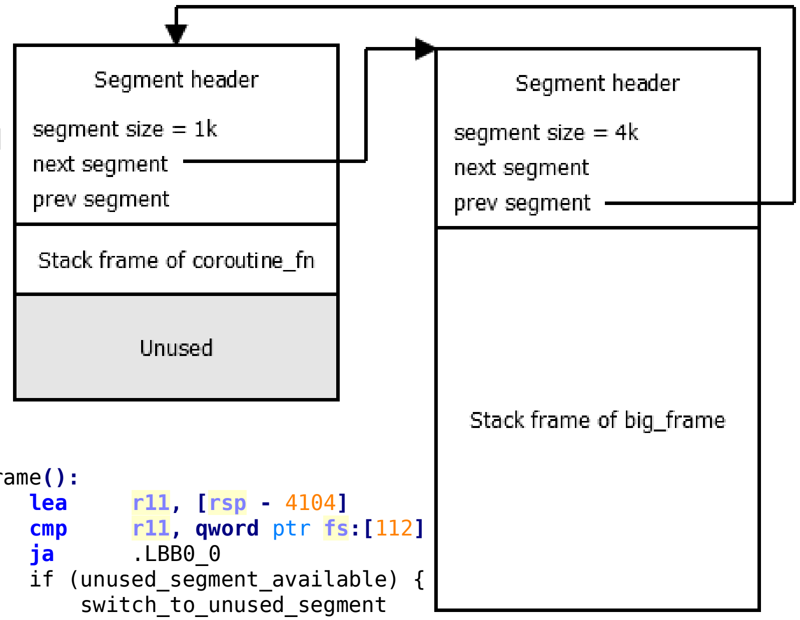
- Prelude is very cheap: load + cmp + jmp
- Branch predictor eliminates overhead if no allocation is needed
- Slow path only taken when stack needs resizing

Stack handling

- `libseff` uses **segmented stacks** for stack handling
- Coroutines are given an **initial stack** (size can be chosen by the programmer)
- Every function call **checks available stack space** vs **function stack frame size**
- Potential performance issue: **hot split problem**

```
int big_frame() {
    int array[1024];
    ...
    return 0;
}
int coroutine_fn() {
    for (int i = 0; i < 1000000; i++) {
        big_frame();
    }
}
```

- `big_frame` allocates new segment, but is deleted at end of function call
- Allocate and deallocate 1M segments?!



Can be solved with runtime support!

- Do not deallocate segment upon return, just change pointers
- No need to allocate new segment, just reuse old segment!
- Allocation is replaced by just switching stack ptr
- Change autogenerated function prelude to do check: minimal overhead

Can be solved with compiler analysis!

- Detect big allocation in `big_frame`, lift it to `coroutine_fn`
- Effectively: combine stack frames of `big_frame` and `coroutine_fn`

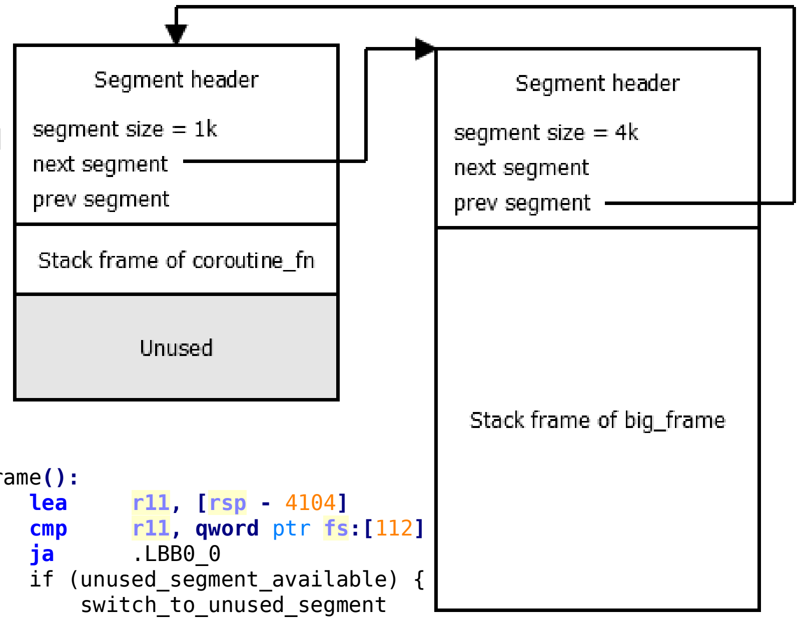
```
big_frame():
        lea     r11, [rsp - 4104]
        cmp     r11, qword ptr fs:[112]
        ja      .LBB0_0
        if (unused_segment_available) {
            switch_to_unused_segment
            ja      .LBB0_0
        }
        mov     r10d, 4104
        mov     r11d, 0
        call    __morestack
        ret
.LBB0_0:
        …
        ret
```

Stack handling

- **`libseff`** uses **segmented stacks** for stack handling
- Coroutines are given an **initial stack** (size can be chosen by the programmer)
- Every function call **checks available stack space** vs **function stack frame size**
- Potential performance issue: **hot split problem**

```
int big_frame() {
    int array[1024];
    ...
    return 0;
}
int coroutine_fn() {
    for (int i = 0; i < 1000000; i++) {
        big_frame();
    }
}
```

- `big_frame` allocates new segment, but is deleted at end of function call
- Allocate and deallocate 1M segments?!




```
big_frame():
        lea     r11, [rsp - 4104]
        cmp     r11, qword ptr fs:[112]
        ja      .LBB0_0
        if (unused_segment_available) {
            switch_to_unused_segment
            ja      .LBB0_0
        }
        mov     r10d, 4104
        mov     r11d, 0
        call    __morestack
        ret
.LBB0_0:
        …
        ret
```

Can be solved with runtime support!

- Do not deallocate segment upon return, just change pointers
- No need to allocate new segment, just reuse old segment!
- Allocation is replaced by just switching stack ptr
- Change autogenerated function prelude to do check: minimal overhead

Can be solved with compiler analysis!

- Detect big allocation in `big_frame`, lift it to `coroutine_fn`
- Effectively: combine stack frames of `big_frame` and `coroutine_fn`

Stack handling

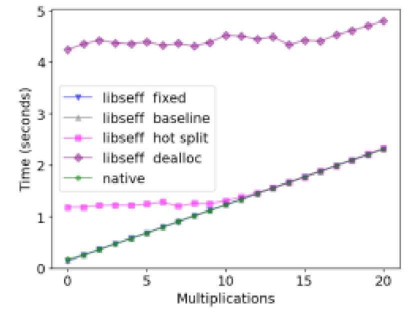
- **`libseff`** uses **segmented stacks** for stack handling
- Coroutines are given an **initial stack** (size can be chosen by the programmer)
- Every function call **checks available stack space** vs **function stack frame size**
- Potential performance issue: **hot split problem**

```c
int8_t *__attribute_noinline__ bottom() {
    volatile int8_t arr[BOTTOM_ARR];
    for (int i = 0; i < MULTS; ++i) {
        x = x * 3.0;
    }
    /* Avoids inlining */
    __asm__("" : "=o"(v) : "o"(v));
    return (void *)arr;
}

void *__attribute_noinline__ top(void *_arg) {
    for (int i = 0; i < REPS; ++i) {
        int8_t *a = bottom();
        /* So the result from bottom is actually "used"*/
        __asm__("" : "=r"(a) : "r"(a));
    }
}
```

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

Benchmark a hot-split call

- `bottom` has enough padding in the stack to require allocating a segment
- Perform variable number of multiplications
- Compare 5 versions
  - Native: no split stack
  - Libseff fixed: fixed-size stacks
  - Libseff baseline: large segments, no hot split
  - Libseff hot split: hot split with segment reuse
  - Libseff dealloc: hot split with eager segment deallocation

Conclusions

- Segmented stack prelude does not cause perceptible overhead by itself
- Eager deallocation causes unacceptable performance degradation
- Beyond 8 multiplications, segment recycling completely mitigates hot split
- Below 8 multiplications **`bottom` should be inlined anyways**
- Stack recycling "happy path" could be faster

Stack handling

- `libseff` uses **segmented stacks** for stack handling
- Coroutines are given an **initial stack** (size can be chosen by the programmer)
- Every function call **checks available stack space** vs **function stack frame size**
- Potential performance issue: **calling library code**

```
int coroutine_fn() {
    puts("hello");
}
```

- `printf` was compiled without support for segmented stacks
- GNU/Clang segmented stack approach: **conservatively reserve large segment**
- Alternative approach: **switch to system stack**

```
int puts(char*);

int __attribute__((no_split_stack)) puts_syscall_wrapper(char*);
__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;"
)
```

- `libseff` can generate wrapper via macro but:
  - Must be requested manually
  - Programmer must choose which version of the function to call
  - This breaks the promise that stickful concurrency is transparent
- Compiler support can eliminate this need
  - Compiler autogenerates wrappers
  - Or just compile everything with segmented stacks!

Split stack overhead

- **`libseff`** uses **segmented stacks** for stack handling
- Coroutines are given an **initial stack** (size can be chosen by the programmer)
- Every function call **checks available stack space** vs **function stack frame size**
- Potential performance issue: **calling library code**

```
int coroutine_fn() {
    puts("hello");
}
```

- `printf` was compiled without support for segmented stacks
- GNU/Clang segmented stack approach: **conservatively reserve large segment**
- Alternative approach: **switch to system stack**

```
int puts(char*);

int __attribute__((no_split_stack)) puts_syscall_wrapper(char*);
__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;"
)
```

- **`libseff`** can generate wrapper via macro but:
  - Must be requested manually
  - Programmer must choose which version of the function to call
  - This breaks the promise that stickful concurrency is transparent
- Compiler support can eliminate this need
  - Compiler autogenerates wrappers
  - Or just compile everything with segmented stacks!

Benchmarks: context-switching

- We compare **`libseff`**, **`libco`** (Tencent's stackful coroutine library) and C++ coroutines (with **`cppcoro`**)
  - **`libco`** is used in **real-world applications** (currently in WeChat backend!)
- All benchmarks running on clang 10.0.0 at optimization level 3
- Context-switching: create a coroutine and resume/yield n times
- We control three different variables: number of yield/resume, depth of the stack, and size of each stack frame

```
void *deep_coroutine(seff_coroutine_t *self, void *arg) {
    char arr[padding];
    int64_t depth = (int64_t)arg;
    if (depth == 0) {
        volatile bool loop = true;
        while (loop) {
            seff_yield(self, nullptr);
        }
        return arr;
    } else {
        deep_coroutine(self, (void *)(depth - 1));
        return arr;
    }
}
```

Frames padded with uninitialized data

Infinite loop, volatile to avoid optimizations

No tail call to avoid optimizations

`libseff` version

```
seff_coroutine_t *k1 = seff_coroutine_new(fn, (void *)depth);
seff_coroutine_t *k2 = seff_coroutine_new(fn, (void *)depth);
for (size_t i = 0; i < iterations / 2; i++) {
    seff_resume(k1, nullptr);
    seff_resume(k2, nullptr);
}
seff_coroutine_delete(k1);
seff_coroutine_delete(k2);
```

Driver code interleaves execution of 2 coroutines `iterations/2` times each

Benchmarks: context-switching

- We compare **`libseff`**, **`libco`** (Tencent's stackful coroutine library) and C++ coroutines (with **`cppcoro`**)
- All benchmarks running on clang 10.0.0 at optimization level 3
- Context-switching: create a coroutine and resume/yield n times
- We control three different variables: number of yield/resume, depth of the stack, and size of each stack frame

```
void *deep_coroutine(void *arg) {
    char arr[padding];
    int64_t depth = (int64_t)arg;
    if (depth == 0) {
        volatile bool loop = true;
        while (loop) {
            co_yield_ct();
        }
        return arr;
    } else {
        deep_coroutine((void *)(depth - 1));
        return arr;
    }
}
```

Frames padded with uninitialized data

Infinite loop, volatile to avoid optimizations

No tail call to avoid optimizations

`libco` version

API is almost identical to **`libseff`**

```
stCoRoutine_t *k1;
stCoRoutine_t *k2;
stShareStack_t *share_stack
    = co_alloc_sharestack(1, 1024 * 128);
stCoRoutineAttr_t attr;
attr.stack_size = 0;
attr.share_stack = share_stack;
co_create(&k1, &attr, fn, (void *)depth);
co_create(&k2, &attr, fn, (void *)depth);
for (size_t i = 0; i < iterations / 2; i++) {
    co_resume(k1);
    co_resume(k2);
}
co_release(k1);
co_release(k2);
```

Set `share_stack` on (for resizable coroutines, otherwise stack size is fixed)

Benchmarks: context-switching

- We compare **`libseff`**, **`libco`** (Tencent's stackful coroutine library) and C++ coroutines (with **`cppcoro`**)
- All benchmarks running on clang 10.0.0 at optimization level 3
- Context-switching: create a coroutine and resume/yield n times
- We control three different variables: number of yield/resume, depth of the stack, and size of each stack frame

```
cppcoro::recursive_generator<char *>
deep_coroutine(int64_t depth) {
    char arr[padding];
    if (depth == 0) {
        volatile bool loop = true;
        while (loop) {
            co_yield arr;
        }
    } else {
        co_yield deep_coroutine_rec (depth - 1);
        co_yield arr;
    }
}
```

Frames padded with uninitialized data

Infinite loop, volatile to avoid optimizations

No tail call to avoid optimizations

`cppcoro` version

`cppcoro` api does not allow for an exact comparison. We use `recursive_generator` here because it is more optimized, but **`cppcoro`** recursive generators are more limited than coroutines (no async).

```
cppcoro::recursive_generator<char*> k1
    = coroutine_fn(depth);
cppcoro::recursive_generator<char*> k2
    = coroutine_fn(depth);
cppcoro::recursive_generator<char*>
    ::iterator k1_iter = k1.begin();
cppcoro::recursive_generator<char*>
    ::iterator k2_iter = k2.begin();
for (auto i = 0; i < iterations / 2; i++) {
    k1_iter++;
    k2_iter++;
}
```

`cppcoro` coroutines are heap-allocated, but RAII so there is no explicit deallocation

Benchmarks: context-switching

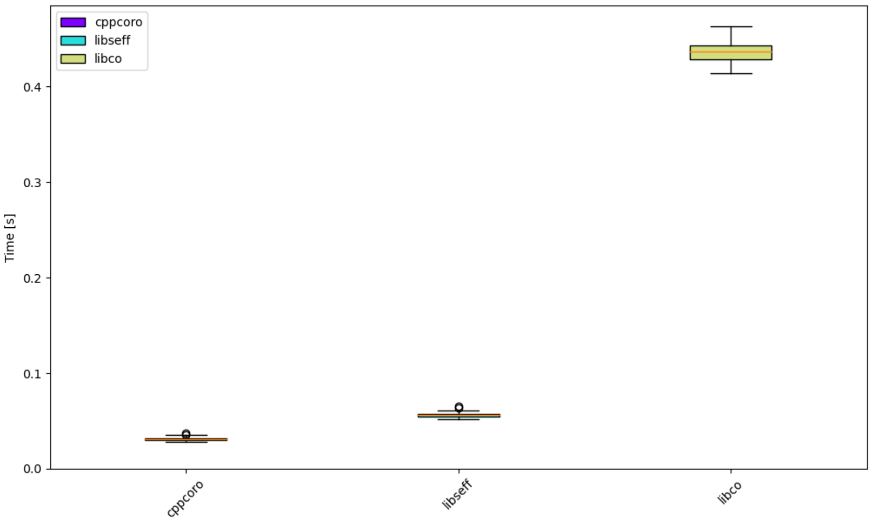
- We compare **`libseff`**, **`libco`** (Tencent's stackful coroutine library) and C++ coroutines (with **`cppcoro`**)
- All benchmarks running on clang 15.0.0 at optimization level 3
- Context-switching: create a coroutine and resume/yield n times
- We control three different variables: number of yield/resume, depth of the stack, and size of each stack frame

If –O0 we're actually 4.3x FASTER than cppcoro!

Simple example

- 10,000,000 iterations (resume + yield)
- Call stack has depth 0
- Stack frames have no padding

Framework	Mean time (ms)	Relative
`libco`	473.4 ± 33.5	14.00
`libseff`	60.9 ± 2.4	1.80
`cppcoro`	33.8 ± 3.0	1.00



Conclusions

- **`libseff`** is much more efficient than **`libco`**, due to using split stacks instead of stack copying
- **`cppcoro`** is faster, but less flexible (benchmark code could not be extended with async)

Benchmarks: context-switching

- We compare **`libseff`**, **`libco`** (Tencent's stackful coroutine library) and C++ coroutines (with **`cppcoro`**)
- All benchmarks running on clang 15.0.0 at optimization level 3
- Context-switching: create a coroutine and resume/yield n times
- We control three different variables: number of yield/resume, depth of the stack, and size of each stack frame

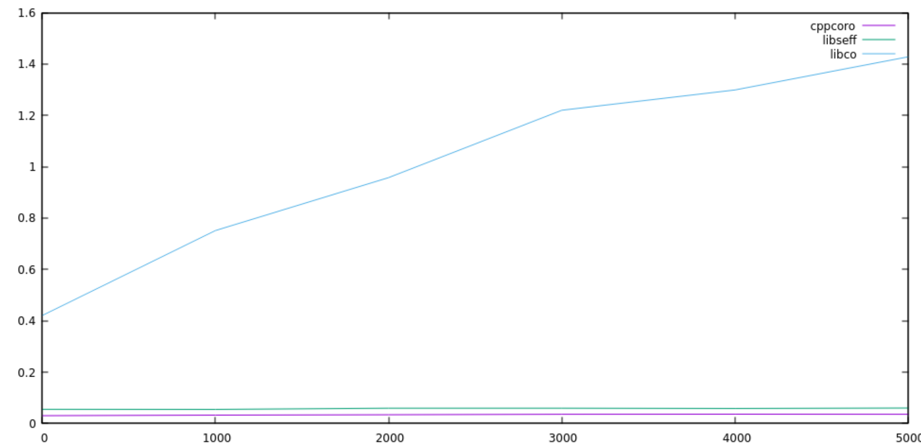
Stack size scaling

- 10,000,000 iterations (resume + yield)
- Call stack has depth 0
- Stack frames have 0-5kb of padding

Size = 5000

Framework	Mean time (ms)	Relative
`libco`	1,428.01	39.63
`libseff`	60.36	1.67
`cppcoro`	36.03	1.00

Time (s)

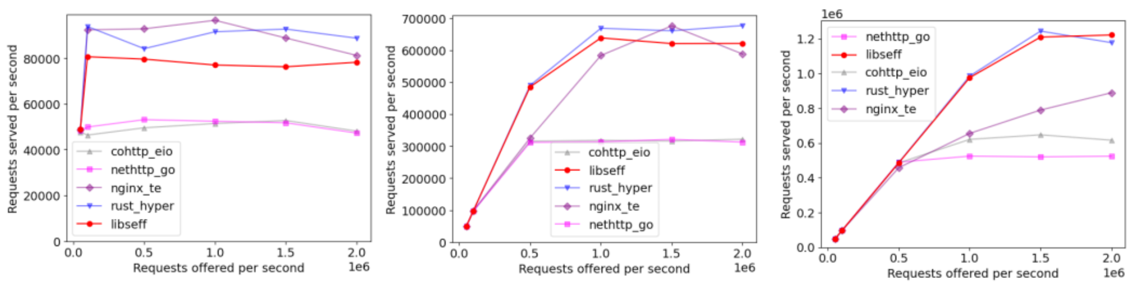


Stack padding (bytes)

Conclusions
- As expected, **`libco`** scales linearly with stack size due to stack copying
- Performance of **`libseff`** and **`cppcoro`** is independent of stack size

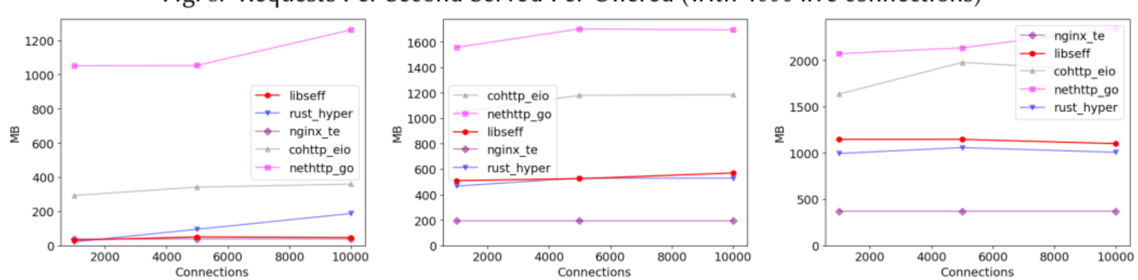
Case study

- Goal 1: showcase performance of **`libseff`** features in “realistic” application
- Goal 2: show how to write applications and schedulers using **`libseff`**
    1. “Proof of concept” multi-threaded scheduler with async capabilities based on epoll (can easily be adapted to poll/select)
    2. Echo server built on example scheduler, using “listen-accept-fork” approach with coroutines
    3. Benchmark single-threaded performance & multi-threaded scaling
    4. Compare against nginx, hyper (Rust+tokio), OCaml (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 1000000 requests offered per second)

- Toy implementation but performance is competitive
- Scheduler is extremely naïve, uses simple work-stealing & lock-free datastructures
- Higher memory consumption than Rust & C due to memory wastage, not significant

Learned lessons

- **Segmented stacks work great, actually!**
  - "Hot-split" problem not so problematic
  - Cost could be reduced further with more ASM
- **Stack allocation + fitting everything into registers**
  - Biggest advantage over other EH libraries: zero heap allocation. **Needs stack stability!**
- **Group commands into effects**
  - Solves 64 bit-set problem (though so far not a real problem)
  - No need for user to specify effect ID (can use ptrs to per-effect globals)
  - Lost flexibility not really an issue
- **Resumption-based API would be possible**
  - Avoiding heap allocation of resumptions is important, use pointer to coroutine
  - Hard to ensure linear usage in presence of race conditions (have to use atomic, cost is prohibitive)
  - Some type-safety gains: can type `resume` properly (C programmers tend not to care)

Conclusions

- **Enormous potential for effects in C**
  - Can be ergonomic & efficient **without** compiler support!
  - But **lots of low-hanging fruit** for compiler support
    - Type-safety, optimizations
- **Major pain point: segmented stacks**
  - **No real alternative:** virtual memory/stack copying **unworkable**
  - Opportunities for optimization
  - Gets better with proper effect typing/"purity" tracking!
- **API differences from high-level languages**
  - No try/handle blocks, continuations not exposed, coroutines as only visible abstraction
  - **Session types** obvious candidate for typing coroutines, add extra safety
- **It is worth doing!**
  - Massive gains in programmer productivity even from a minimal prototype
  - Few sharp edges, usable by non-experts!

Thank You

www.huawei.com