Concurrent & Multicore OCaml: A deep dive

KC Sivaramakrishnan[1] & Stephen Dolan[1]

Leo White[2], Jeremy Yallop[1,3], Armaël Guéneau[4], Anil Madhavapeddy[1,3]

1 

2 

3 

4 

Concurrency ≠ Parallelism

- Concurrency
  - Programming technique
  - **Overlapped** execution of processes
- Parallelism
  - (*Extreme*) Performance hack
  - **Simultaneous** execution of computations

Concurrency ≠ Parallelism

- Concurrency
  - Programming technique
  - **Overlapped** execution of processes
- Parallelism
  - (*Extreme*) Performance hack
  - **Simultaneous** execution of computations

Concurrency ∩ Parallelism ➔ *Scalable Concurrency*

Concurrency ≠ Parallelism

- Concurrency
  - Programming technique
  - **Overlapped** execution of processes
- Parallelism
  - (*Extreme*) Performance hack
  - **Simultaneous** execution of computations

Concurrency ∩ Parallelism ➔ *Scalable Concurrency*
(***Fibers***) (***Domains***)

Schedulers

- Multiplexing fibers over domain(s)
  - Bake scheduler into the runtime system (GHC)

Schedulers

- Multiplexing fibers over domain(s)
  - Bake scheduler into the runtime system (GHC)
- Allow programmers to describe schedulers!
  - Parallel search —> LIFO work-stealing
  - Web-server —> FIFO runqueue
  - Data parallel —> Gang scheduling

Schedulers

- Multiplexing fibers over domain(s)
  - Bake scheduler into the runtime system (GHC)
- Allow programmers to describe schedulers!
  - Parallel search —> LIFO work-stealing
  - Web-server —> FIFO runqueue
  - Data parallel —> Gang scheduling
- ***Algebraic Effects and Handlers***

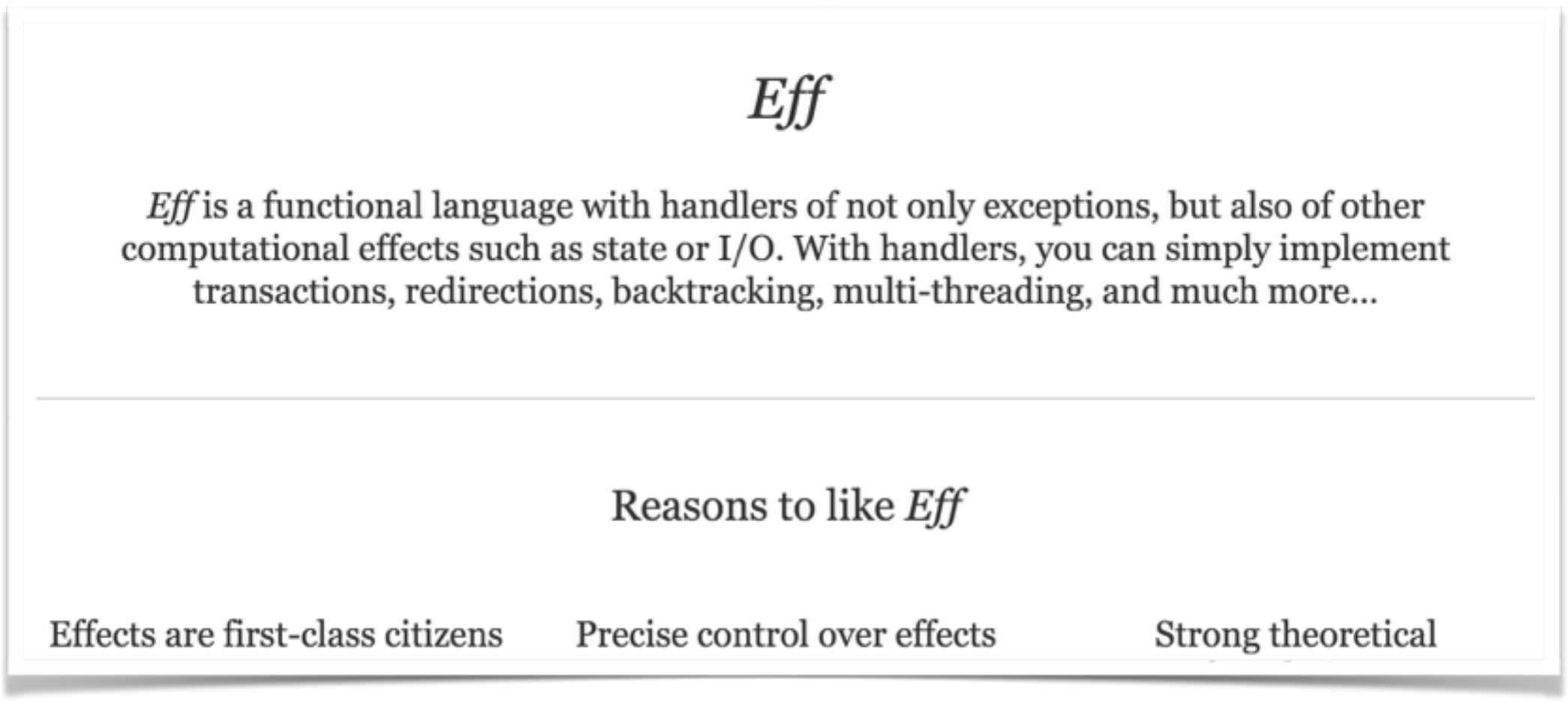
Algebraic effects & handlers

Algebraic effects & handlers

- Programming and reasoning about computational effects in a pure setting.
  - Cf. Monads

Algebraic effects & handlers

- Programming and reasoning about computational effects in a pure setting.
  - Cf. Monads
- *Eff* — **http://www.eff-lang.org/**



Eff

Eff is a functional language with handlers of not only exceptions, but also of other computational effects such as state or I/O. With handlers, you can simply implement transactions, redirections, backtracking, multi-threading, and much more...

Reasons to like *Eff*

Effects are first-class citizens

Precise control over effects

Strong theoretical

Algebraic Effects: Example

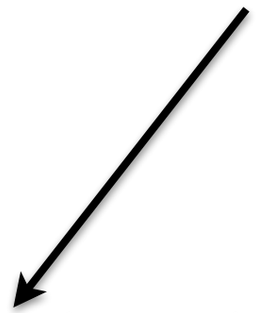
```
exception Foo of int

let f () = 1 + (raise (Foo 3))

let r =
  try
    f ()
  with Foo i -> i + 1
```

Algebraic Effects: Example

```
exception Foo of int

let f () = 1 + (raise (Foo 3))

let r =
  try
    f ()
  with Foo i -> i + 1
```

Algebraic Effects: Example

```
exception Foo of int

let f () = 1 + (raise (Foo 3))

let r =
  try
    f ()
  with Foo i -> i + 1
```



$$\text{val r : int} = 4$$

Algebraic Effects: Example

```
exception Foo of int

let f () = 1 + (raise (Foo 3))

let r =
  try
    f ()
  with Foo i -> i + 1
```

$$\text{val r : int} = 4$$

```
effect Foo : int -> int

let f () = 1 + (perform (Foo 3))

let r =
  try
    f ()
  with effect (Foo i) k ->
         continue k (i + 1)
```

Algebraic Effects: Example

```
exception Foo of int

let f () = 1 + (raise (Foo 3))

let r =
  try
    f ()
  with Foo i -> i + 1
```

$$\text{val r : int} = 4$$

```
effect Foo : int -> int

let f () = 1 + (perform (Foo 3))

let r =
  try
    f ()
  with effect (Foo i) k ->
         continue k (i + 1)
```

Algebraic Effects: Example

```
exception Foo of int

let f () = 1 + (raise (Foo 3))

let r =
    try
      f ()
    with Foo i -> i + 1
```

val r : int = 4

```
effect Foo : int -> int

let f () = 1 + (perform (Foo 3))

let r =
  try
    f ()
  with effect (Foo i) k ->
         continue k (i + 1)
```

Algebraic Effects: Example

```
exception Foo of int

let f () = 1 + (raise (Foo 3))

let r =
    try
      f ()
with Foo i -> i + 1
```

$$\text{val r : int} = 4$$

```
effect Foo : int -> int

let f () = 1 + (perform (Foo 3)) 4

let r =
  try
    f ()
with effect (Foo i) k ->
       continue k (i + 1)
```

Algebraic Effects: Example

```
exception Foo of int

let f () = 1 + (raise (Foo 3))

let r =
    try
      f ()
with Foo i -> i + 1
```

val r : int = 4

```
effect Foo : int -> int

let f () = 1 + (perform (Foo 3)) 4

let r =
  try
    f ()
with effect (Foo i) k ->
       continue k (i + 1)
```

val r : int = 5

Algebraic Effects: Example

```
exception Foo of int

let f () = 1 + (raise (Foo 3))

let r =
    try
      f ()
with Foo i -> i + 1
```

val r : int = 4

```
effect Foo : int -> int

let f () = 1 + (perform (Foo 3)) 4

let r =
  try
    f ()
with effect (Foo i) k ->
       continue k (i + 1)
```

val r : int = 5

fiber — lightweight stack

Scheduler Demo[1]

[1] https://github.com/kayceesrk/ocaml15-eff/tree/master/chameneos-redux

Implementation

- Fibers: Heap allocated, dynamically resized stacks
  - ~10s of bytes
  - No unnecessary closure allocation costs unlike CPS

Implementation

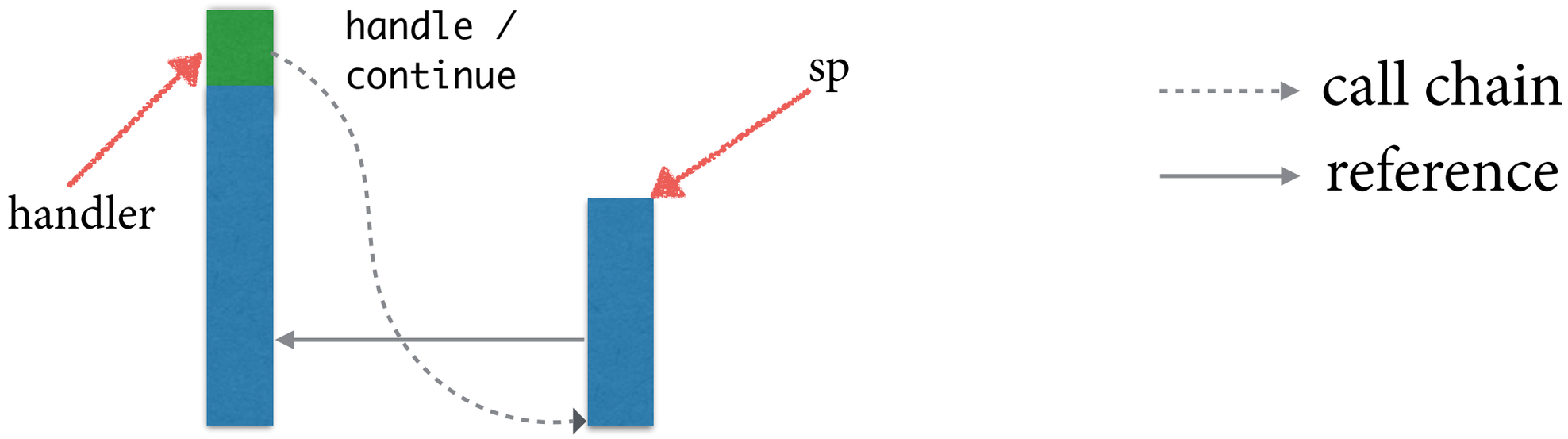
- Fibers: Heap allocated, dynamically resized stacks
  - ~10s of bytes
  - No unnecessary closure allocation costs unlike CPS
- One-shot delimited continuations
  - Simplifies reasoning about resources - sockets, locks, etc.

Implementation

- Fibers: Heap allocated, dynamically resized stacks
  - ~10s of bytes
  - No unnecessary closure allocation costs unlike CPS
- One-shot delimited continuations
  - Simplifies reasoning about resources - sockets, locks, etc.
- Handlers —> Linked-list of fibers

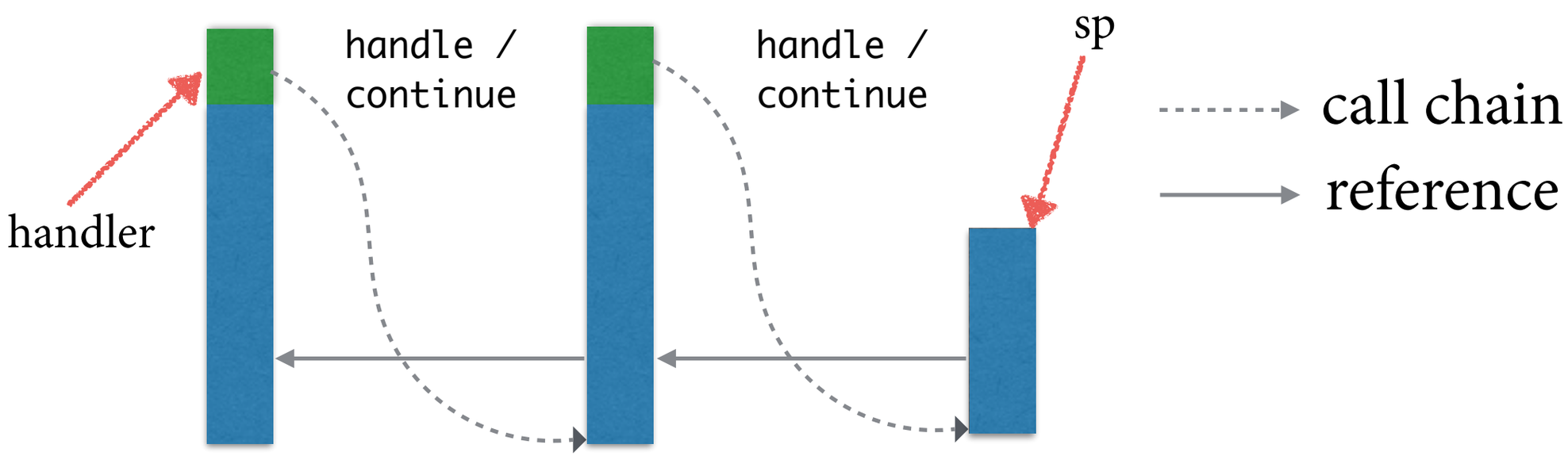
Implementation

- Fibers: Heap allocated, dynamically resized stacks
  - ~10s of bytes
  - No unnecessary closure allocation costs unlike CPS
- One-shot delimited continuations
  - Simplifies reasoning about resources - sockets, locks, etc.
- Handlers —> Linked-list of fibers



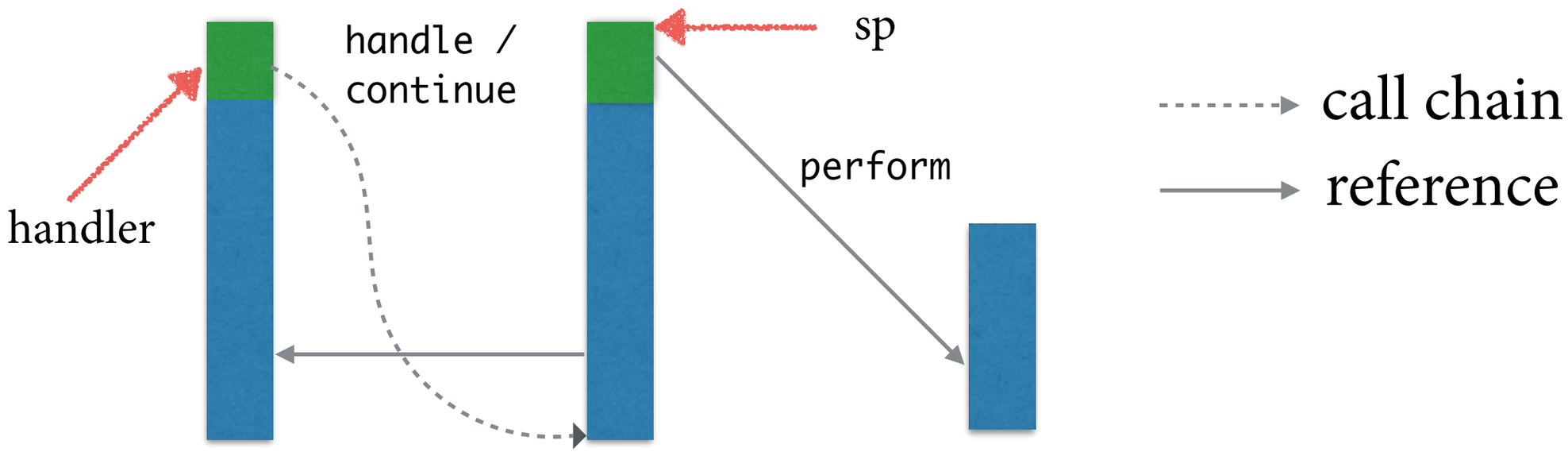
Implementation

- Fibers: Heap allocated, dynamically resized stacks
  - ~10s of bytes
  - No unnecessary closure allocation costs unlike CPS
- One-shot delimited continuations
  - Simplifies reasoning about resources - sockets, locks, etc.
- Handlers —> Linked-list of fibers



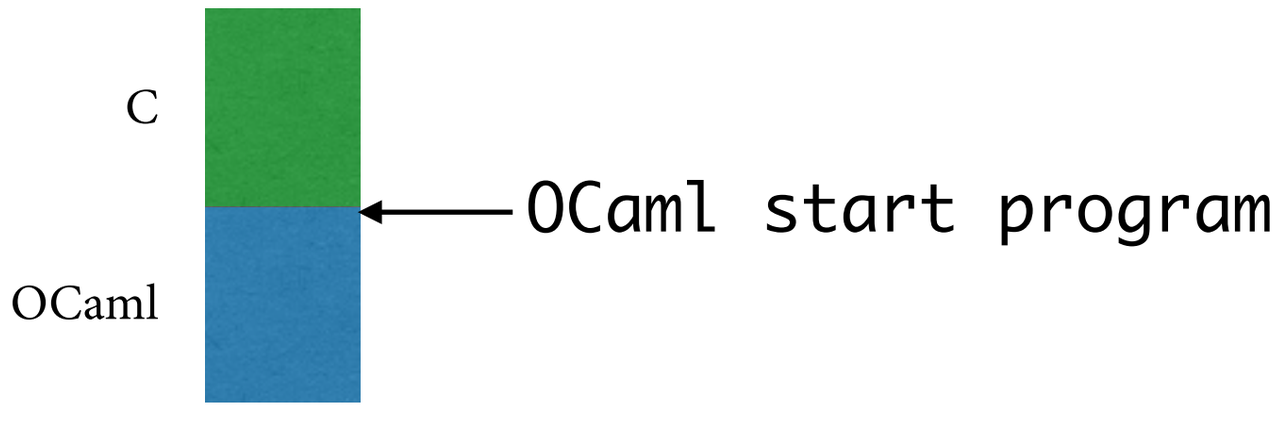
Implementation

- Fibers: Heap allocated, dynamically resized stacks
  - ~10s of bytes
  - No unnecessary closure allocation costs unlike CPS
- One-shot delimited continuations
  - Simplifies reasoning about resources - sockets, locks, etc.
- Handlers —> Linked-list of fibers

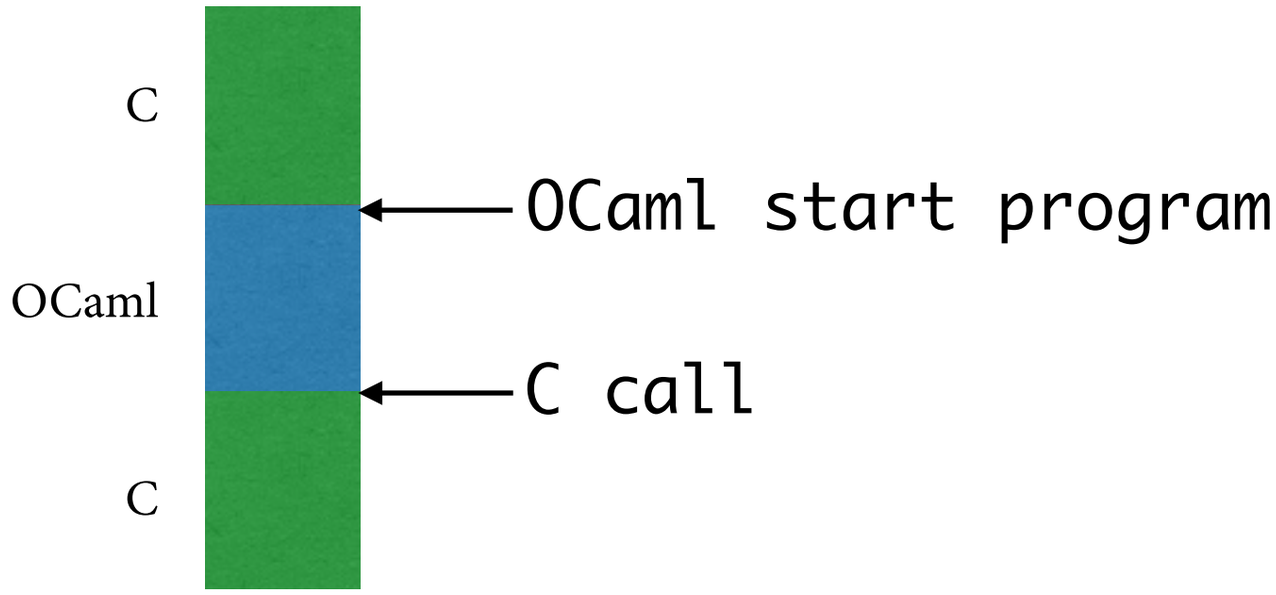


Native-code fibers — Vanilla

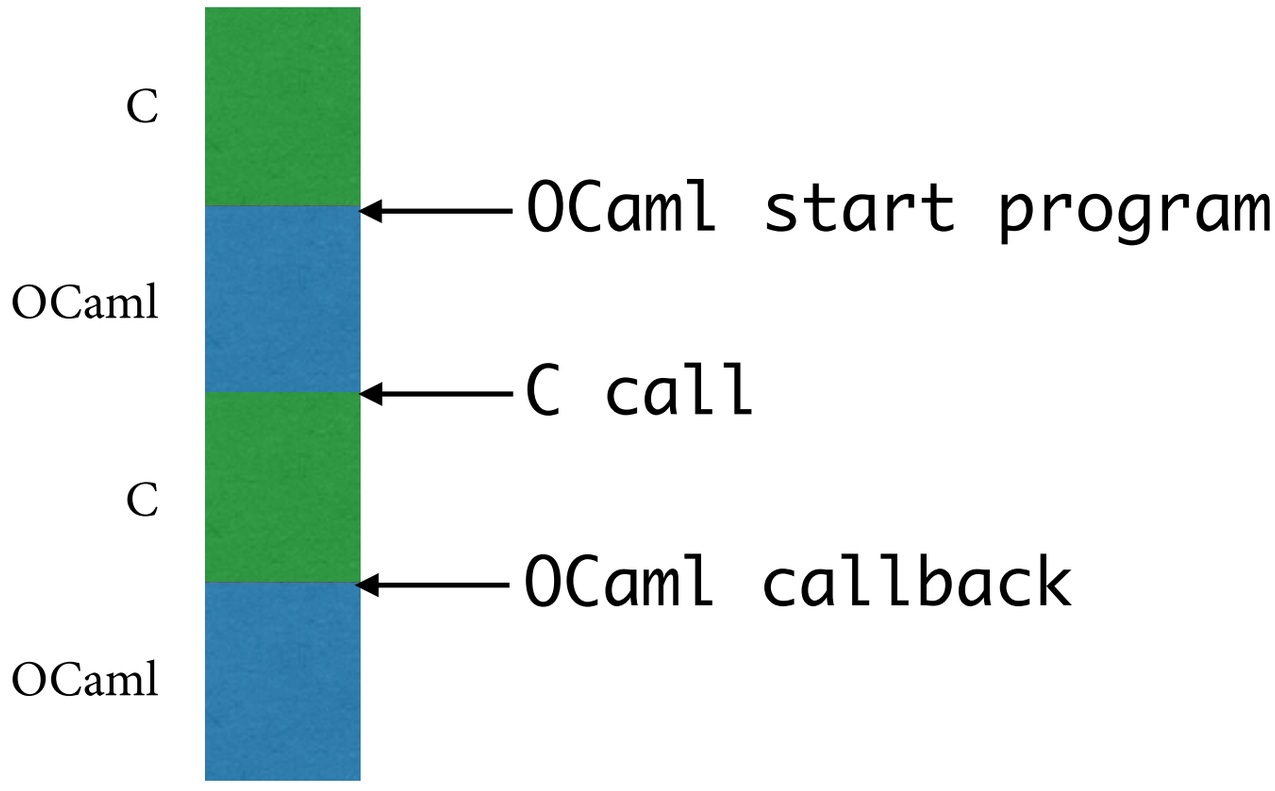
Native-code fibers — Vanilla



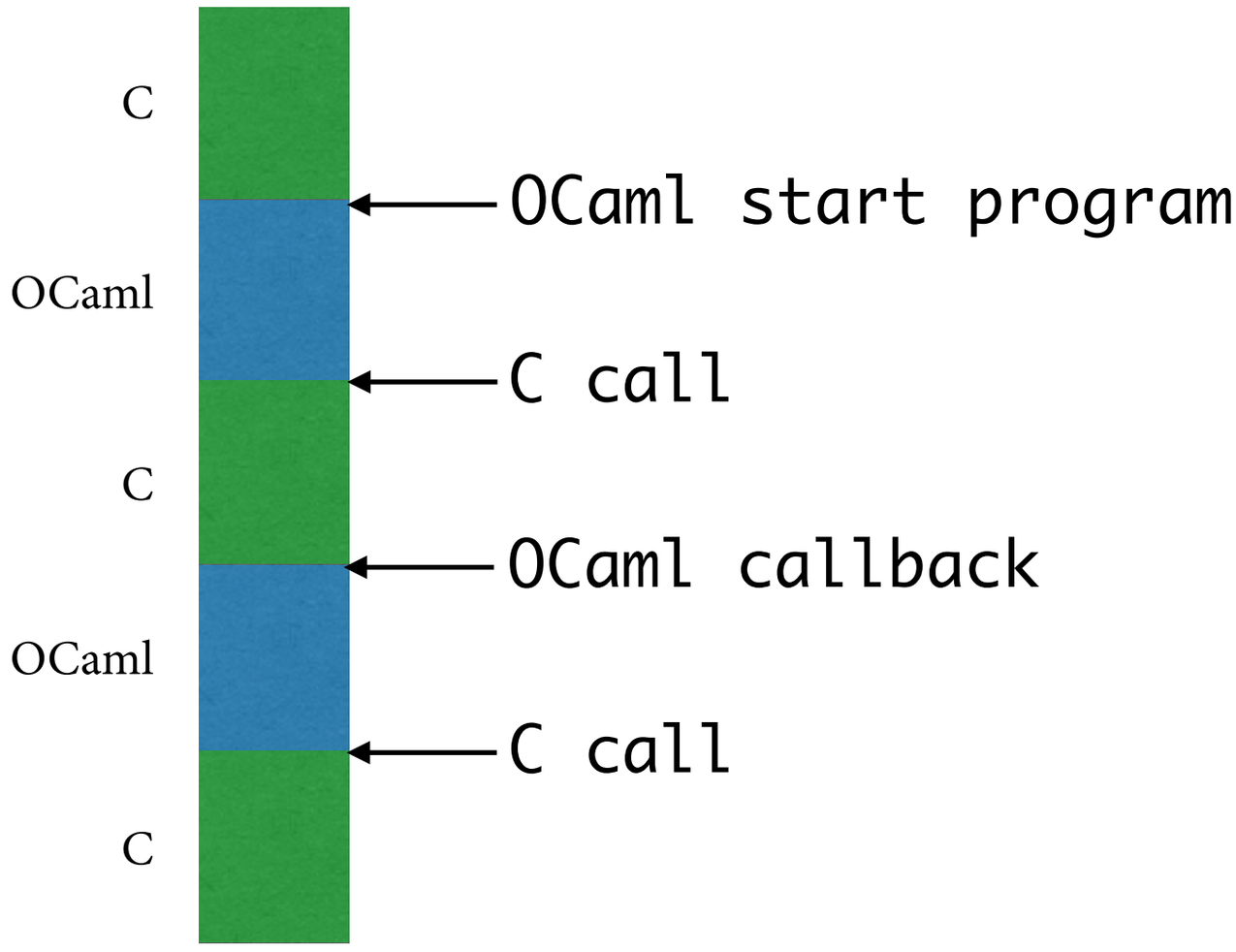
Native-code fibers — Vanilla



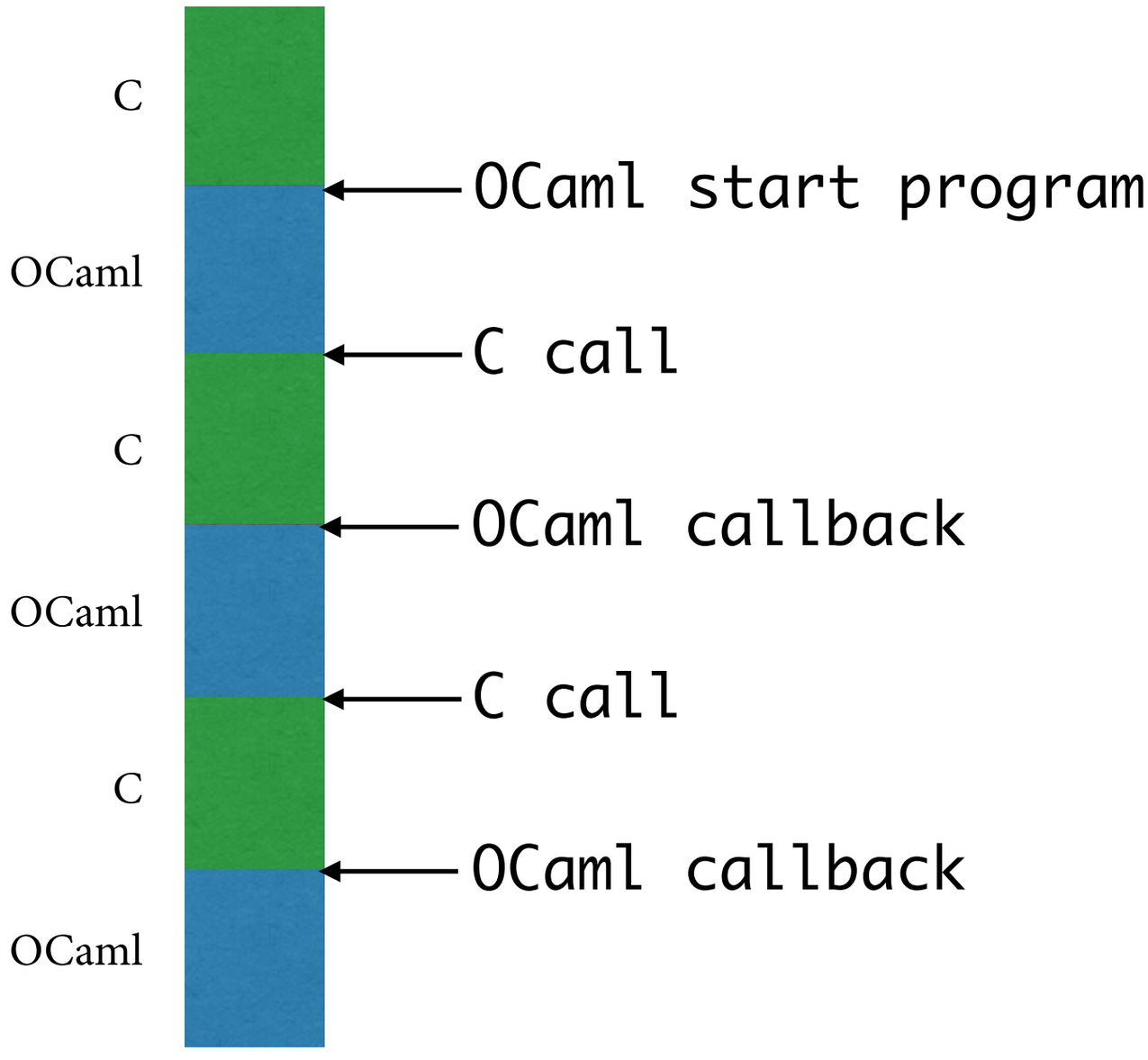
Native-code fibers — Vanilla



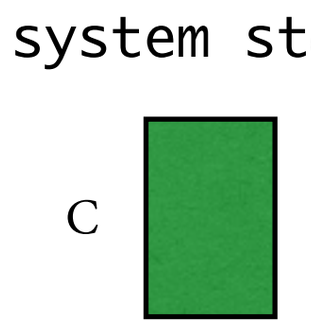
Native-code fibers — Vanilla



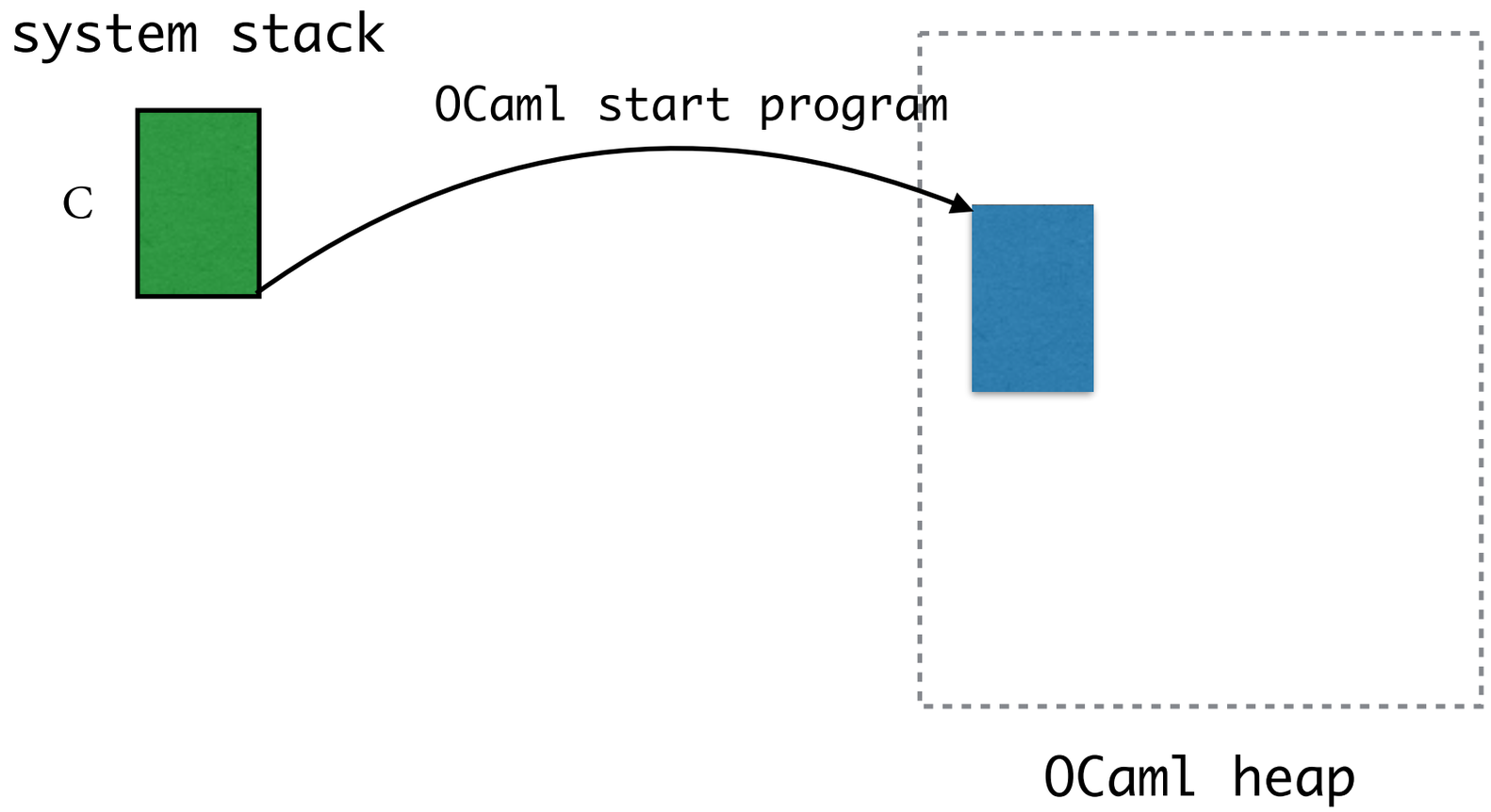
Native-code fibers — Vanilla



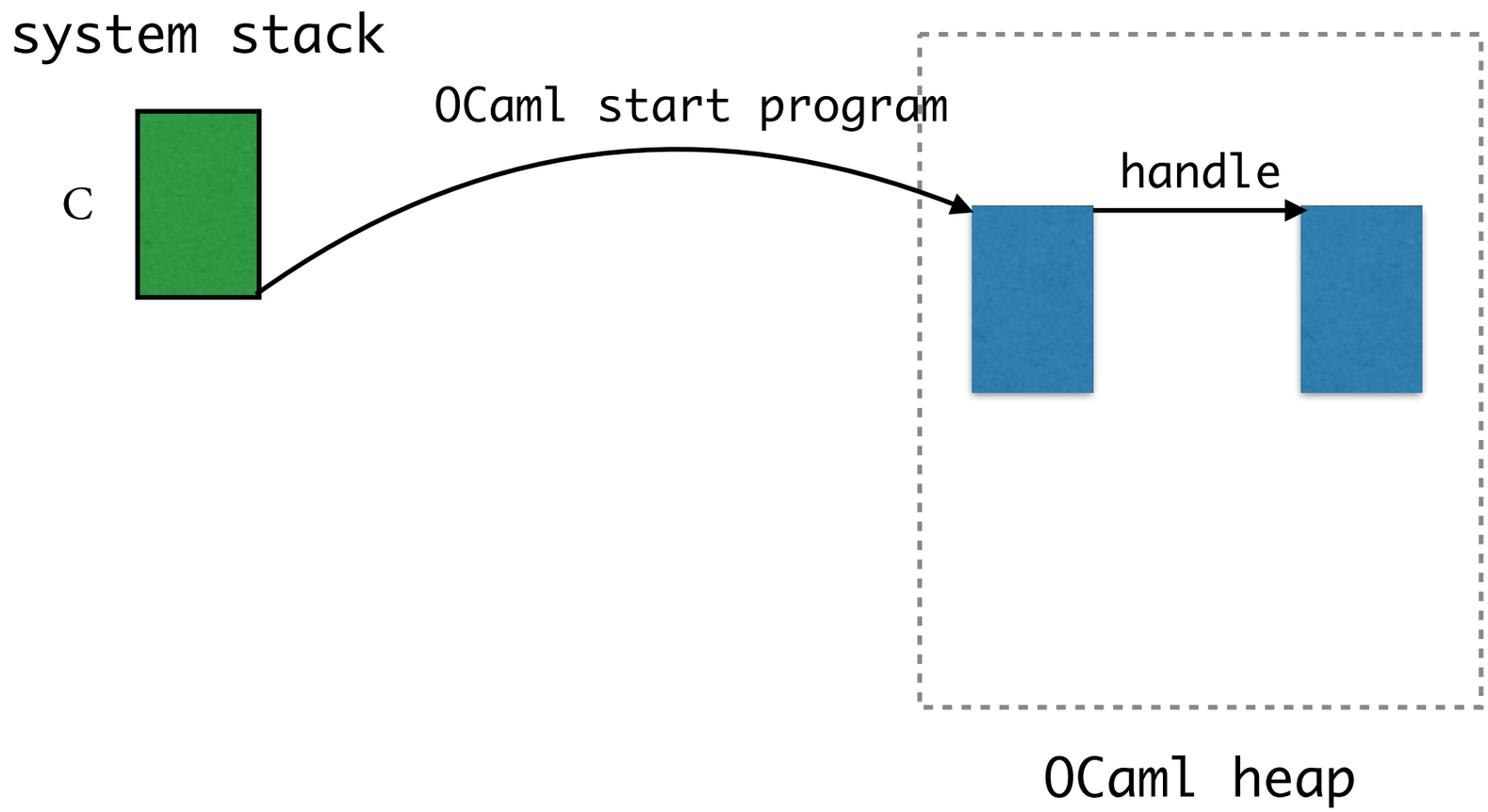
Native-code fibers — Effects



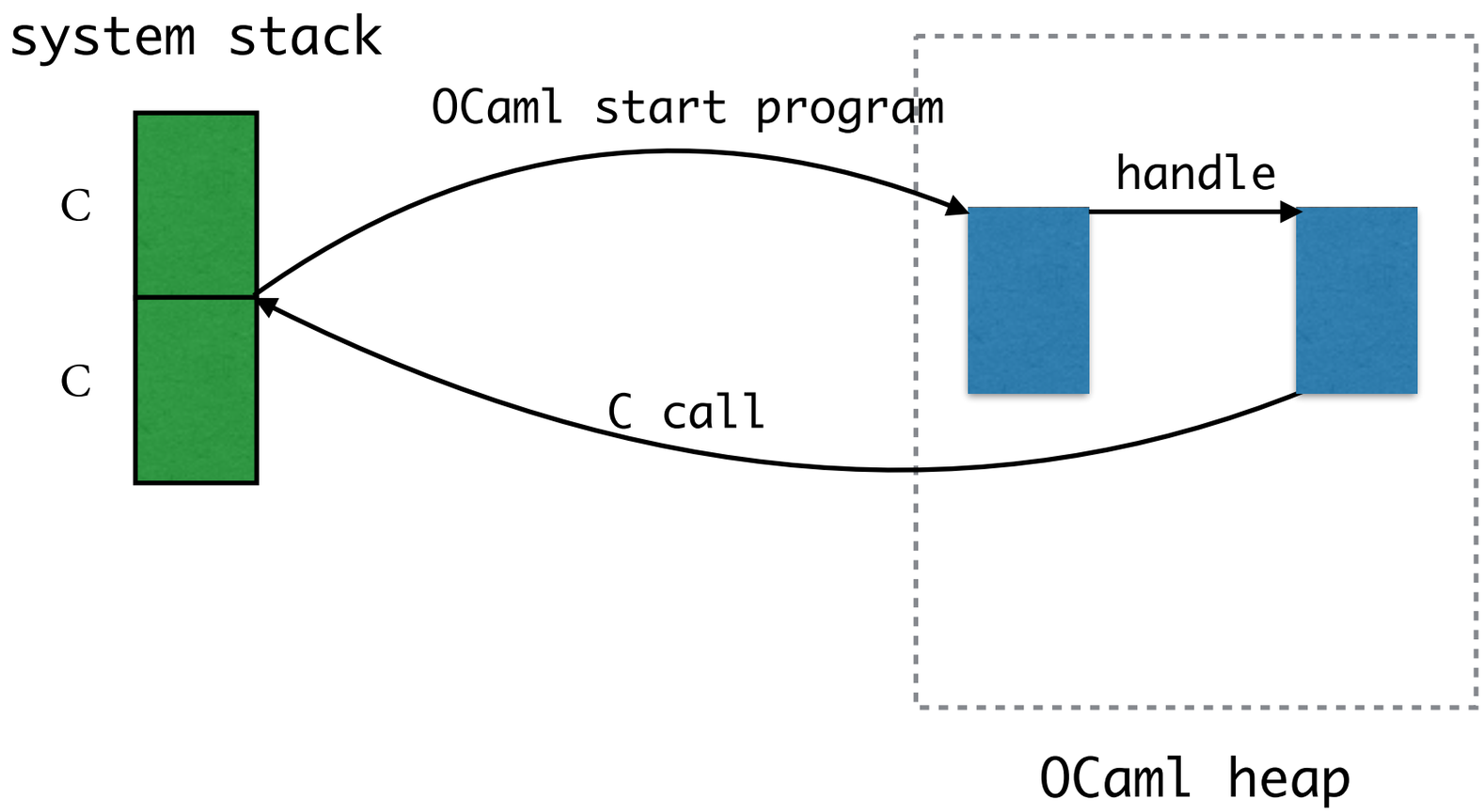
Native-code fibers — Effects



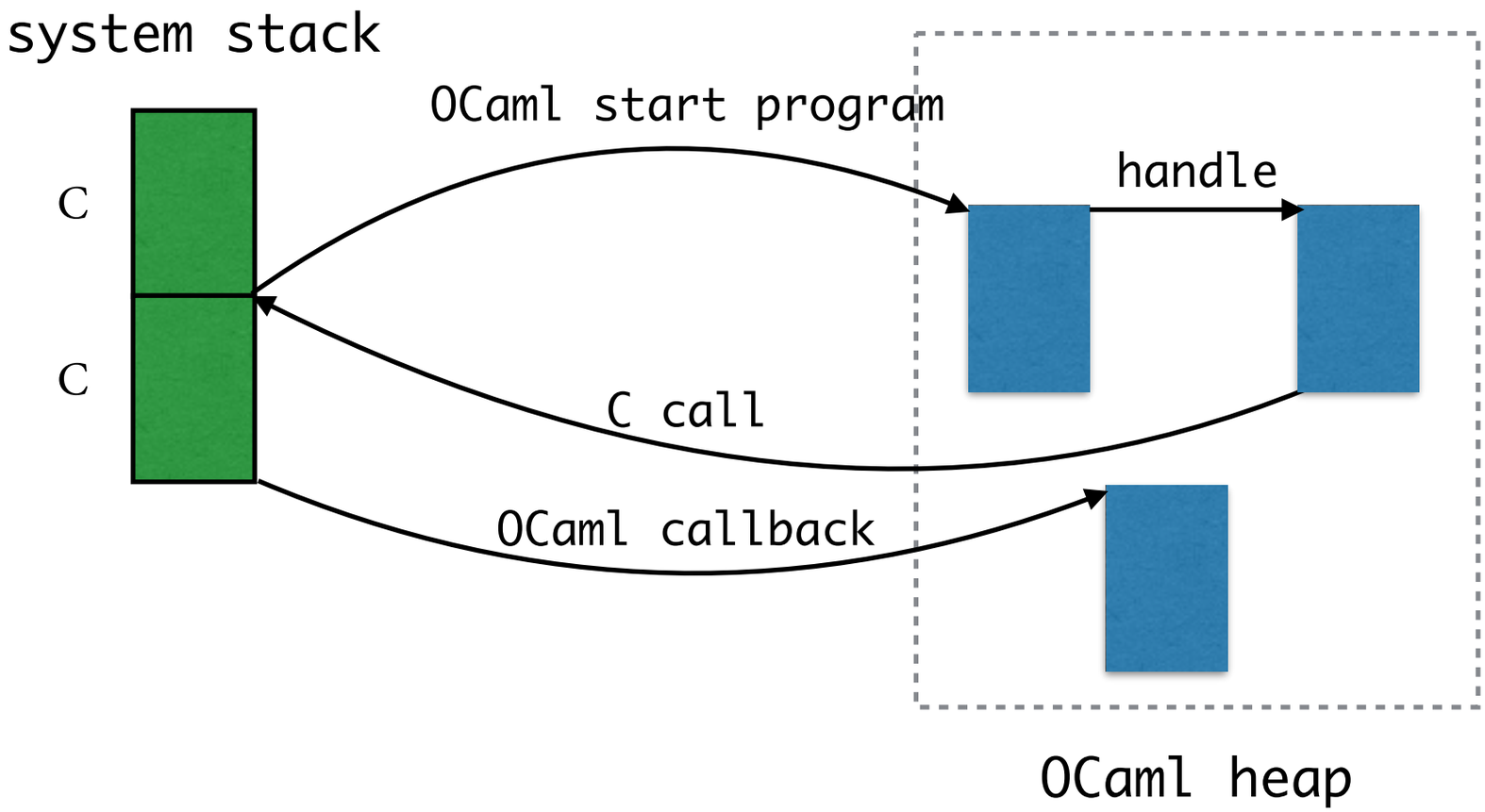
Native-code fibers — Effects



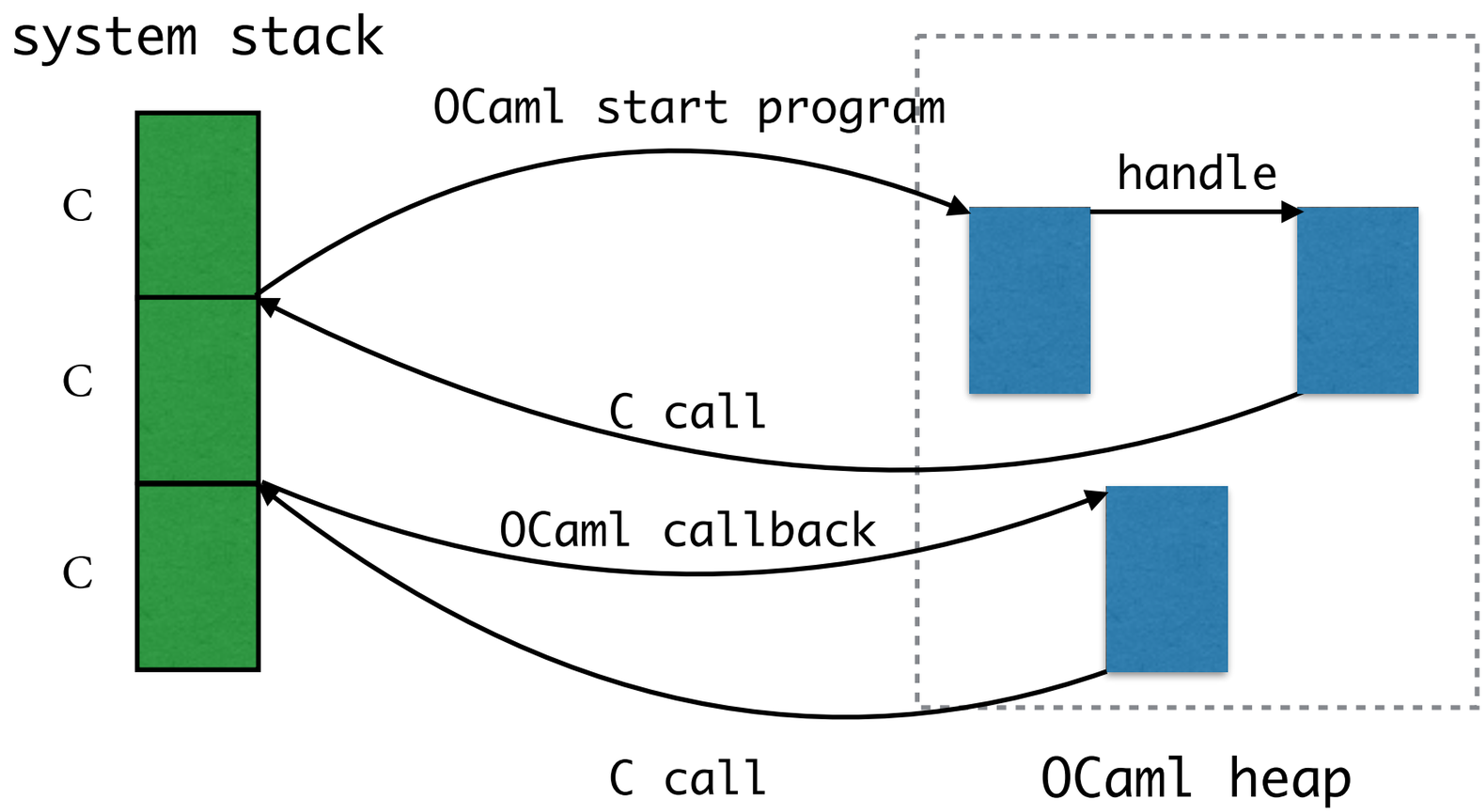
Native-code fibers — Effects



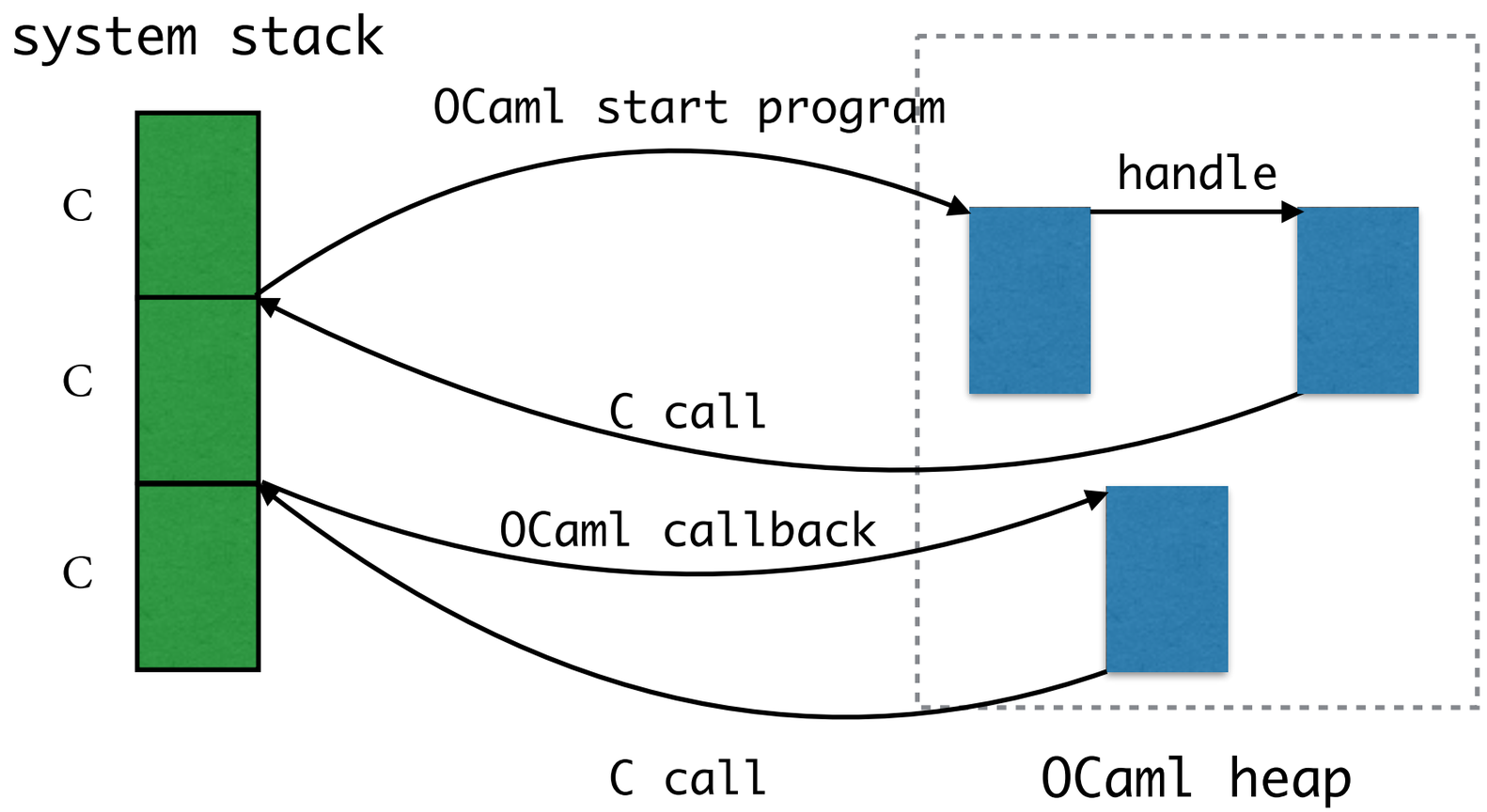
Native-code fibers — Effects



Native-code fibers — Effects

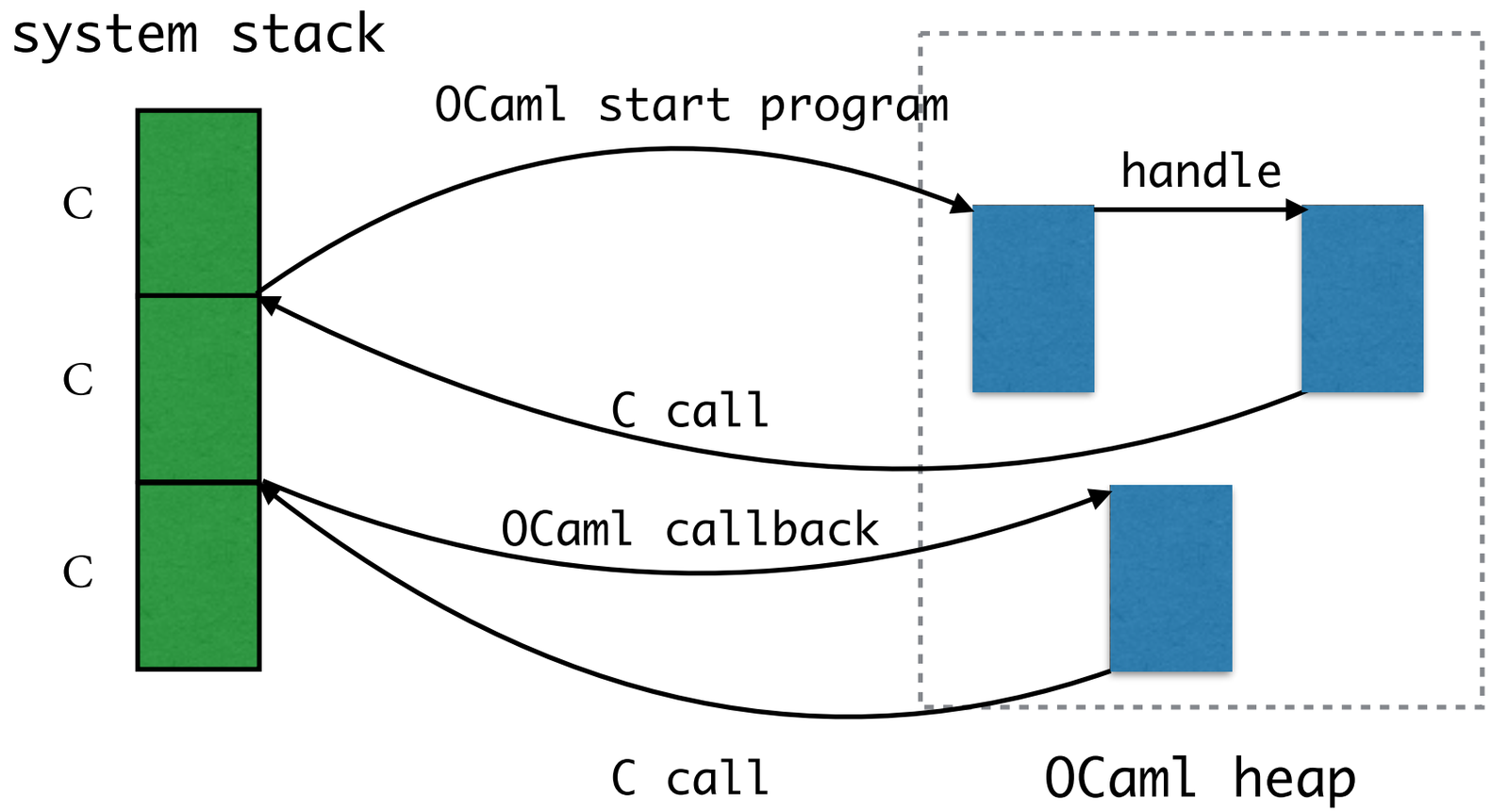


Native-code fibers — Effects



1. Stack overflow checks for OCaml functions
   - Simple static analysis eliminates many checks

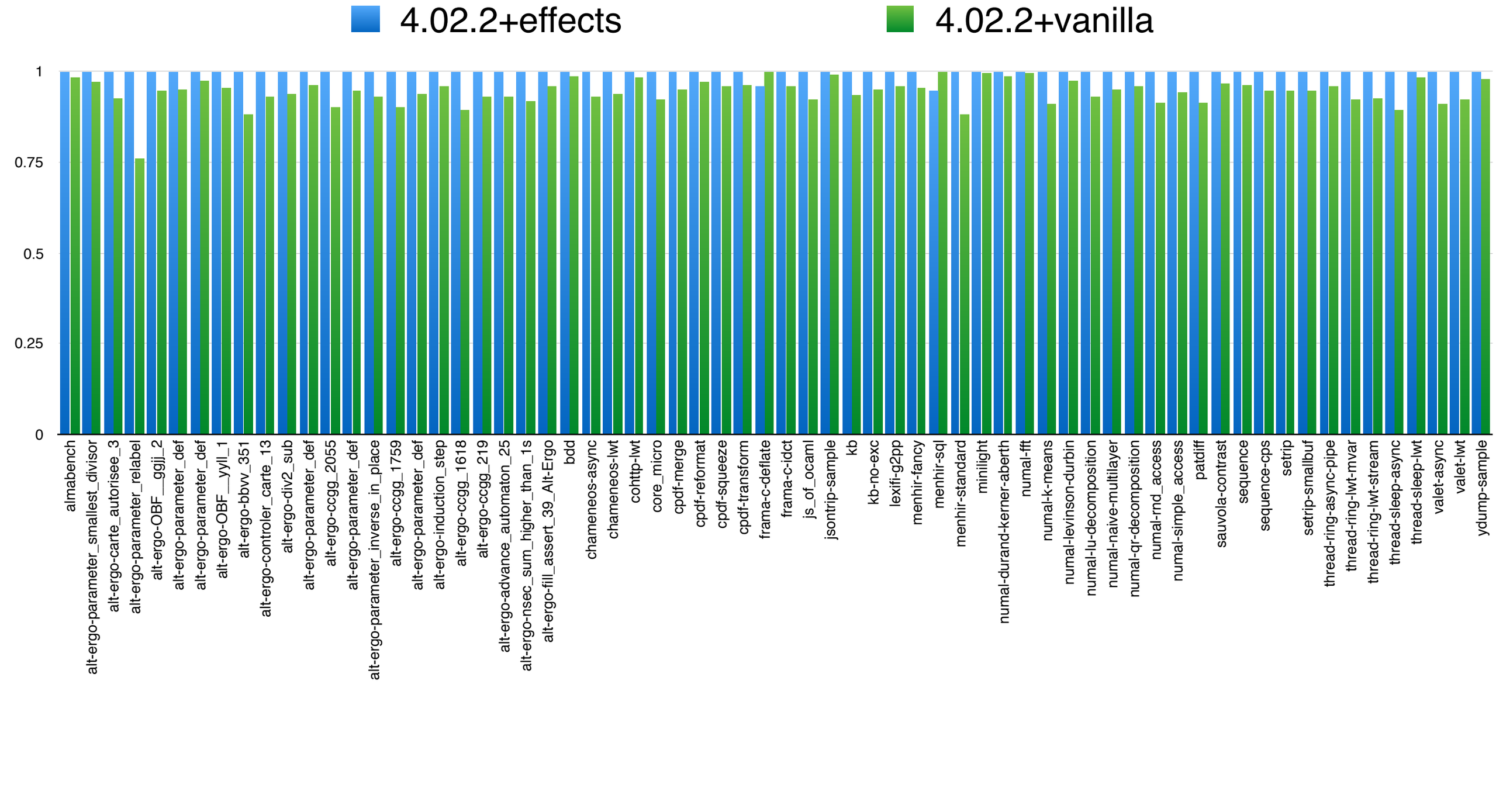
Native-code fibers — Effects



1. Stack overflow checks for OCaml functions
   - Simple static analysis eliminates many checks
2. FFI calls are more expensive due to stack switching
   - Specialise for calls which {allocate / pass arguments on stack / do neither}

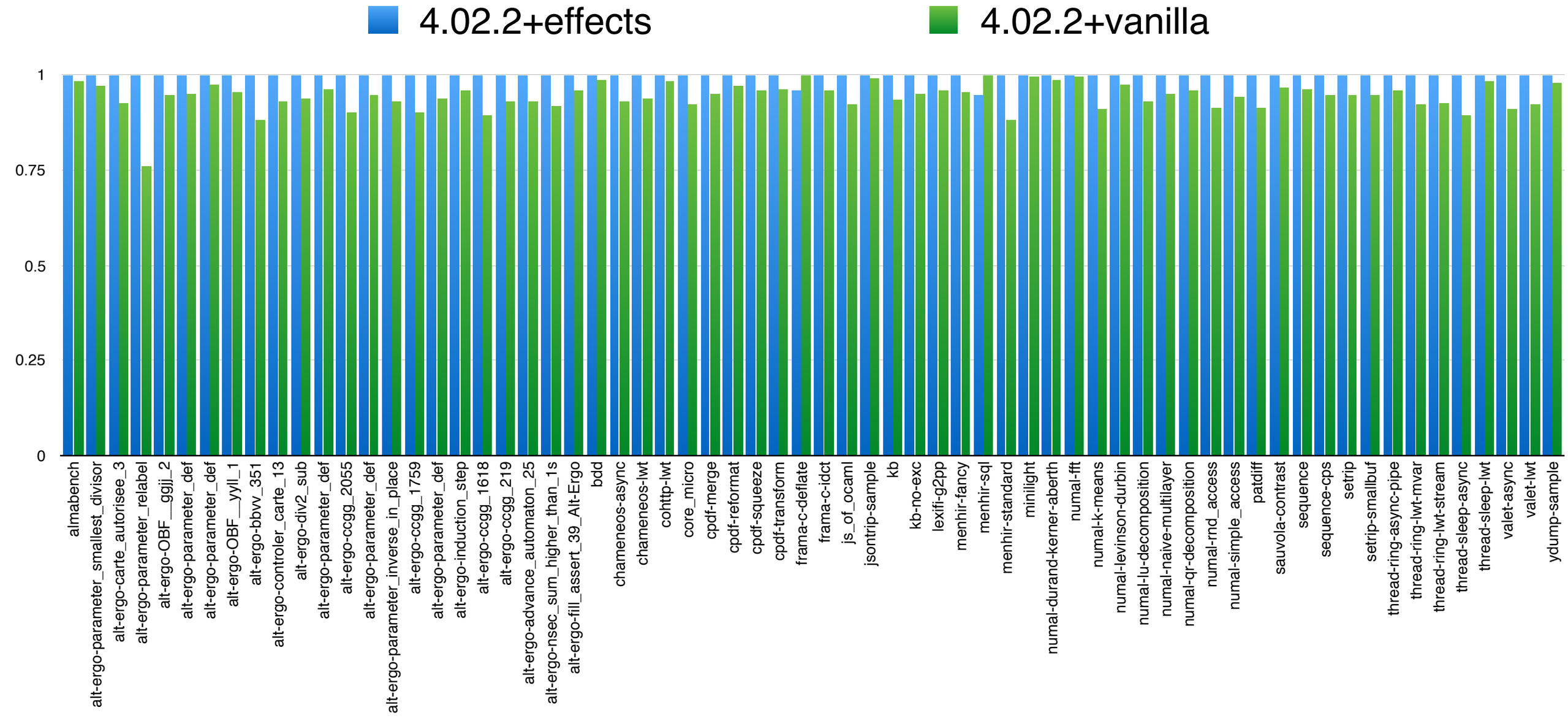
Performance : Vanilla OCaml

Normalised time (lower is better)



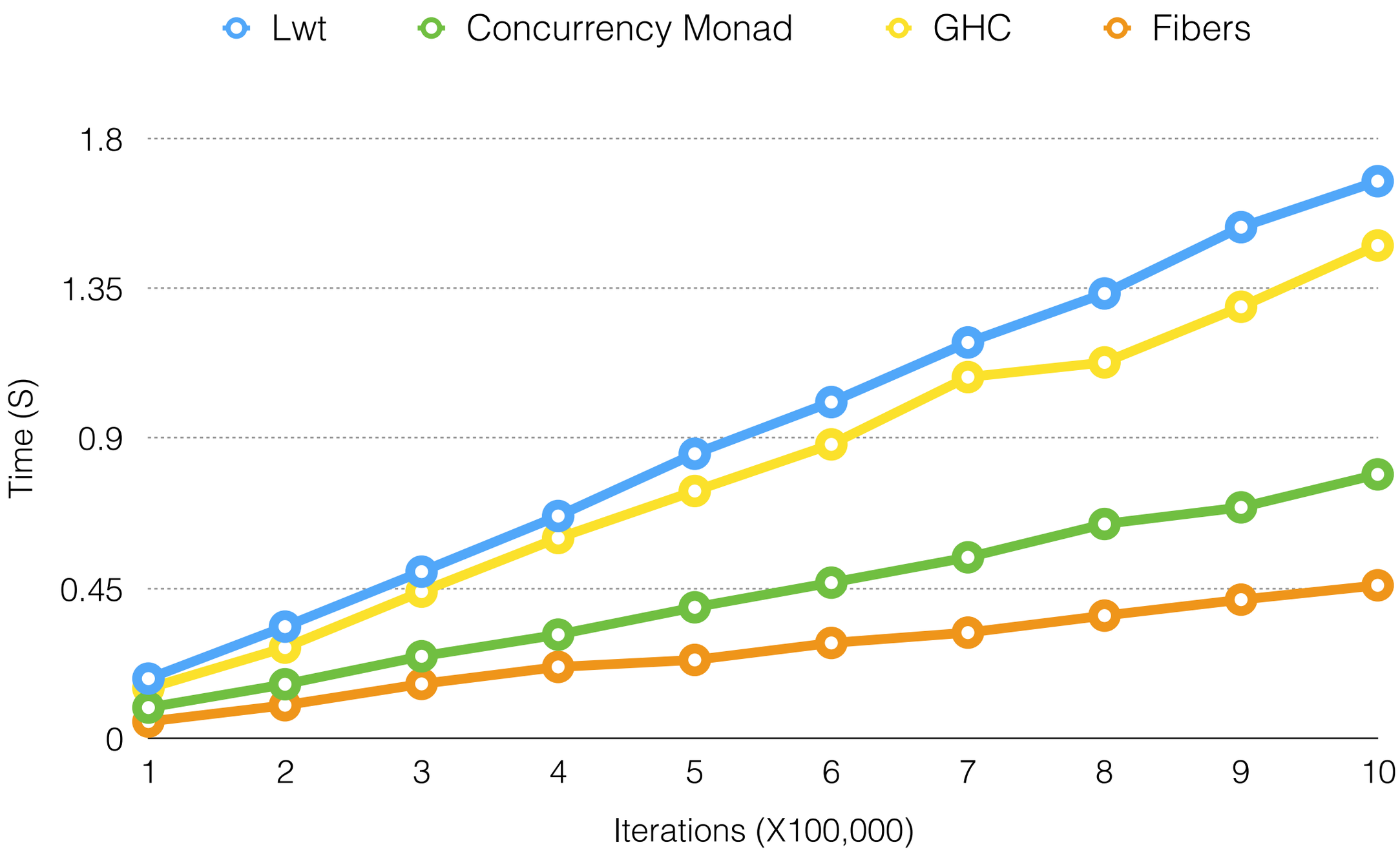
Performance : Vanilla OCaml

Normalised time (lower is better)



4.02.2+effects ~**5.4%** slower

Performance : Chameneos-Redux



Generator from Iterator[1]

```
type 'a t =
| Leaf
| Node of 'a t * 'a * 'a t

let rec iter f = function
  | Leaf -> ()
  | Node (l, x, r) -> iter f l; f x; iter f r
```

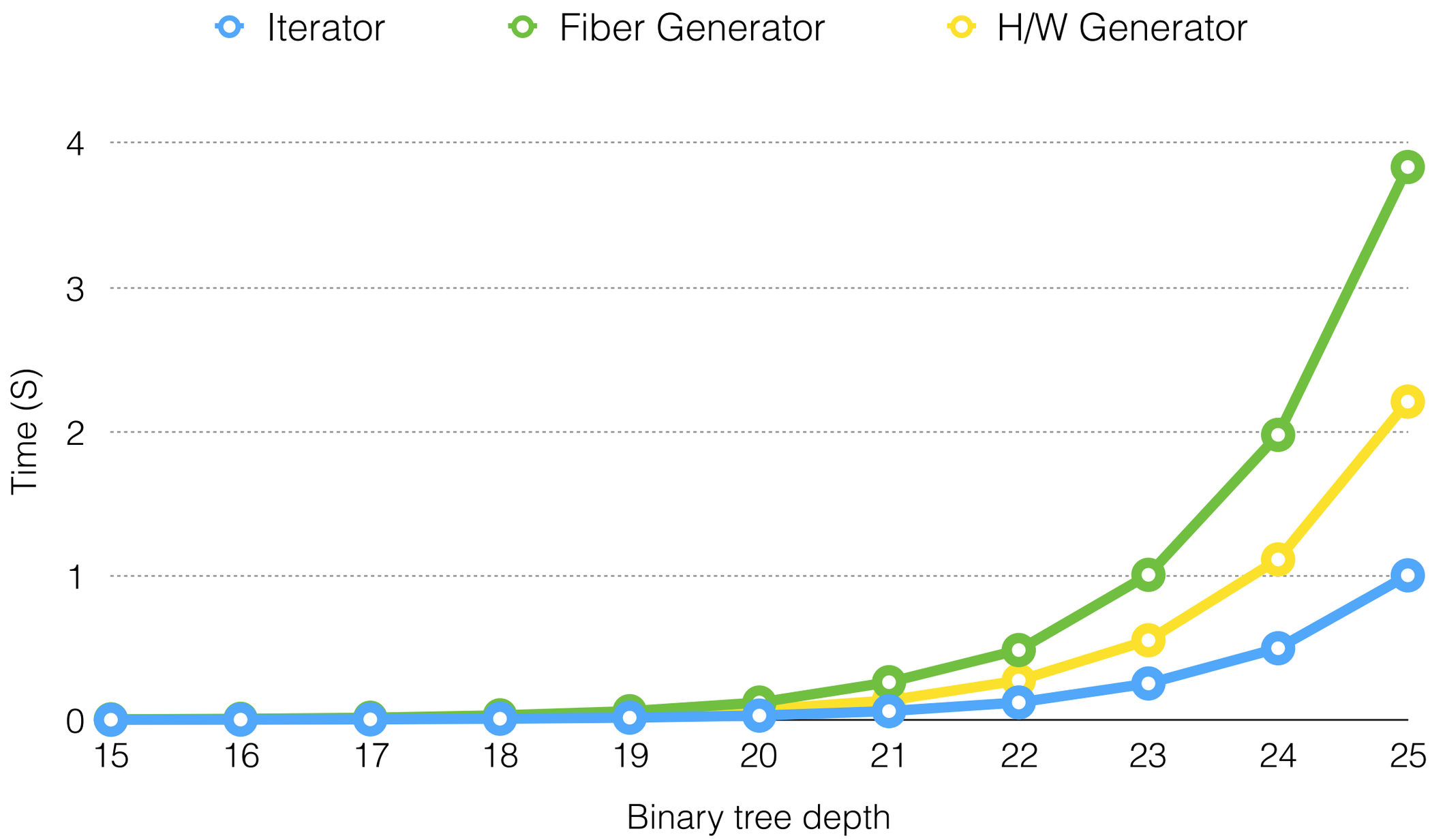
[1] https://github.com/kayceesrk/ocaml15-eff/blob/master/generator.ml

Generator from Iterator[1]

```
type 'a t =
| Leaf
| Node of 'a t * 'a * 'a t

let rec iter f = function
  | Leaf -> ()
  | Node (l, x, r) -> iter f l; f x; iter f r

(* val to_gen : 'a t -> (unit -> 'a option) *)
let to_gen (type a) (t : a t) =
  let module M = struct effect Next : a -> unit end in
  let open M in
  let step = ref (fun () -> assert false) in
  let first_step () =
    try
      iter (fun x -> perform (Next x)) t; None
    with effect (Next v) k ->
      step := continue k; Some v
  in
    step := first_step;
    fun () -> !step ()
```

[1] https://github.com/kayceesrk/ocaml15-eff/blob/master/generator.ml

Performance : Generator



Async I/O in *direct style*[1]

[1] https://github.com/kayceesrk/ocaml15-eff/tree/master/async-io

Async I/O in *direct style*[1]

~~Callback Hell~~

[1] https://github.com/kayceesrk/ocaml15-eff/tree/master/async-io

Javascript backend

- js_of_ocaml
  - OCaml bytecode —> Javascript

Javascript backend

- js_of_ocaml
  - OCaml bytecode —> Javascript
- js_of_ocaml compiler pass
  - Whole-program selective CPS transformation

Javascript backend

- js_of_ocaml
  - OCaml bytecode —> Javascript
- js_of_ocaml compiler pass
  - Whole-program selective CPS transformation
- Work-in-progress!
  - *Runs "hello-effects-world"!*

fin.

https://github.com/kayceesrk/ocaml-eff-example