# Effect handlers in OCaml

**KC Sivaramakrishnan[1] & Stephen Dolan[1]**

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

1 UNIVERSITY OF CAMBRIDGE

2 Jane Street

3 docker

4 ENS DE LYON

# 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)
- Allow programmers to describe schedulers!
  - Parallel search —> LIFO work-stealing
  - Web-server —> FIFO runqueue
  - Data parallel —> Gang scheduling
- ***Algebraic Effects and Handlers***

# Algebraic Effects: Example

```
effect Foo : int -> int

let f () = (perform (Foo 3)) (* 3 + 1 *)
           + (perform (Foo 3)) (* 3 + 1 *)
let r =
  try
    f ()
  with effect (Foo i) k ->
    continue k (i + 1)
```

val r : int = 8

# Dynamic wind

```
let dynamic_wind before_thunk thunk after_thunk =
  before_thunk ();
  let res =
    match thunk () with
    | v -> v
    | exception e -> after_thunk (); raise e
    | effect e k ->
        after_thunk ();
        let res' = perform e in
        before_thunk ();
        continue k res'
  in
  after_thunk ();
  res
```

# Effect systems and modularity

- Right now, we type effects like ML exceptions
  - *(we're in the market for an effect system, if anyone has one lying around...)*
- We need modularity, because effects can:
  - be local, dynamic and fresh
  - be abstracted, renamed and reuse
- We don't know statically whether two effects are the same

# Scheduler Demo[1]

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

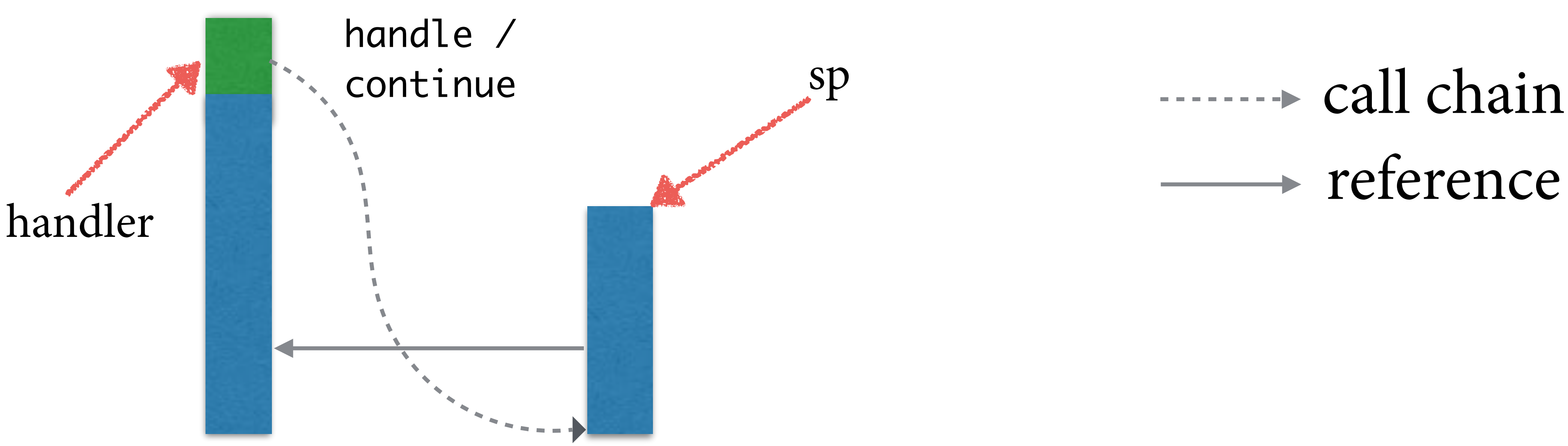
# Generator from Iterator[1]

```
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

(* 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

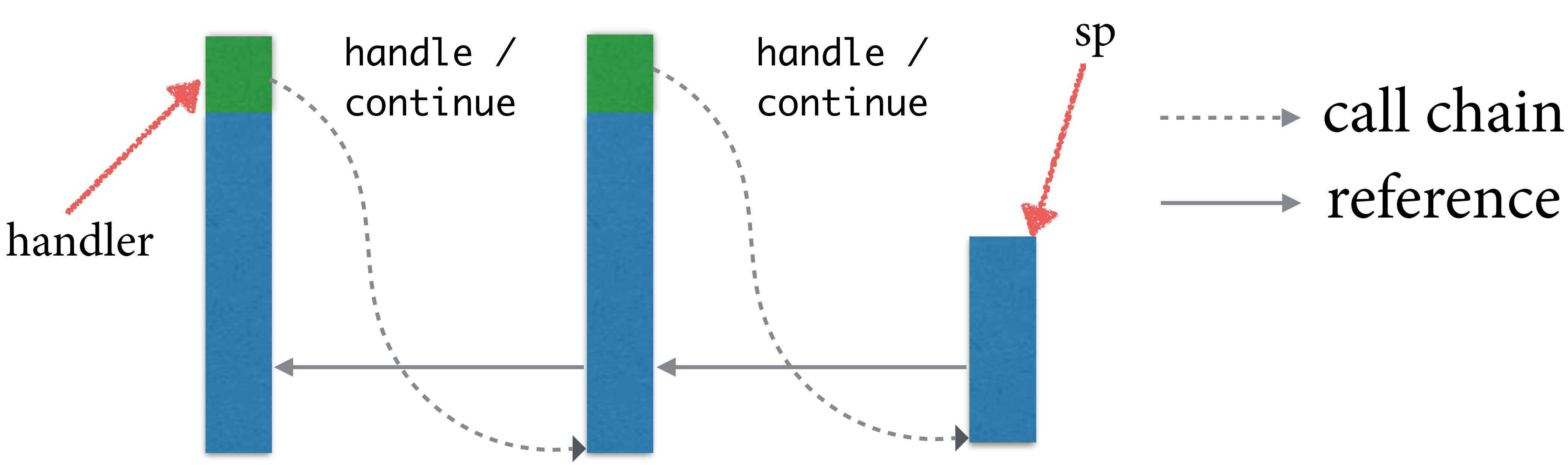
# Implementation

- 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



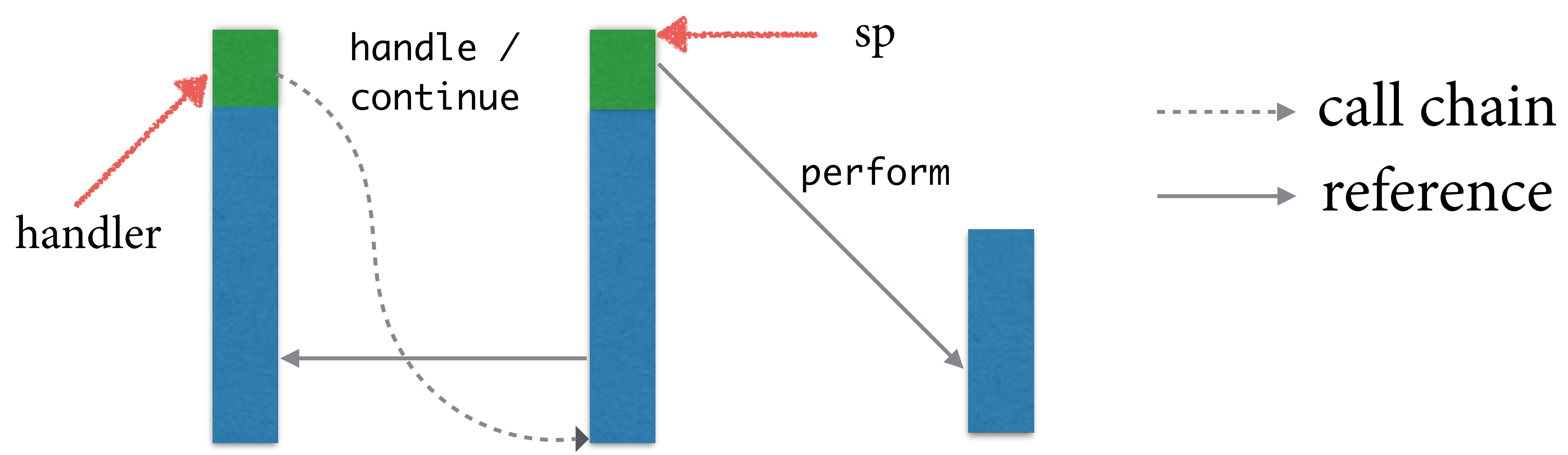
# Implementation

- 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

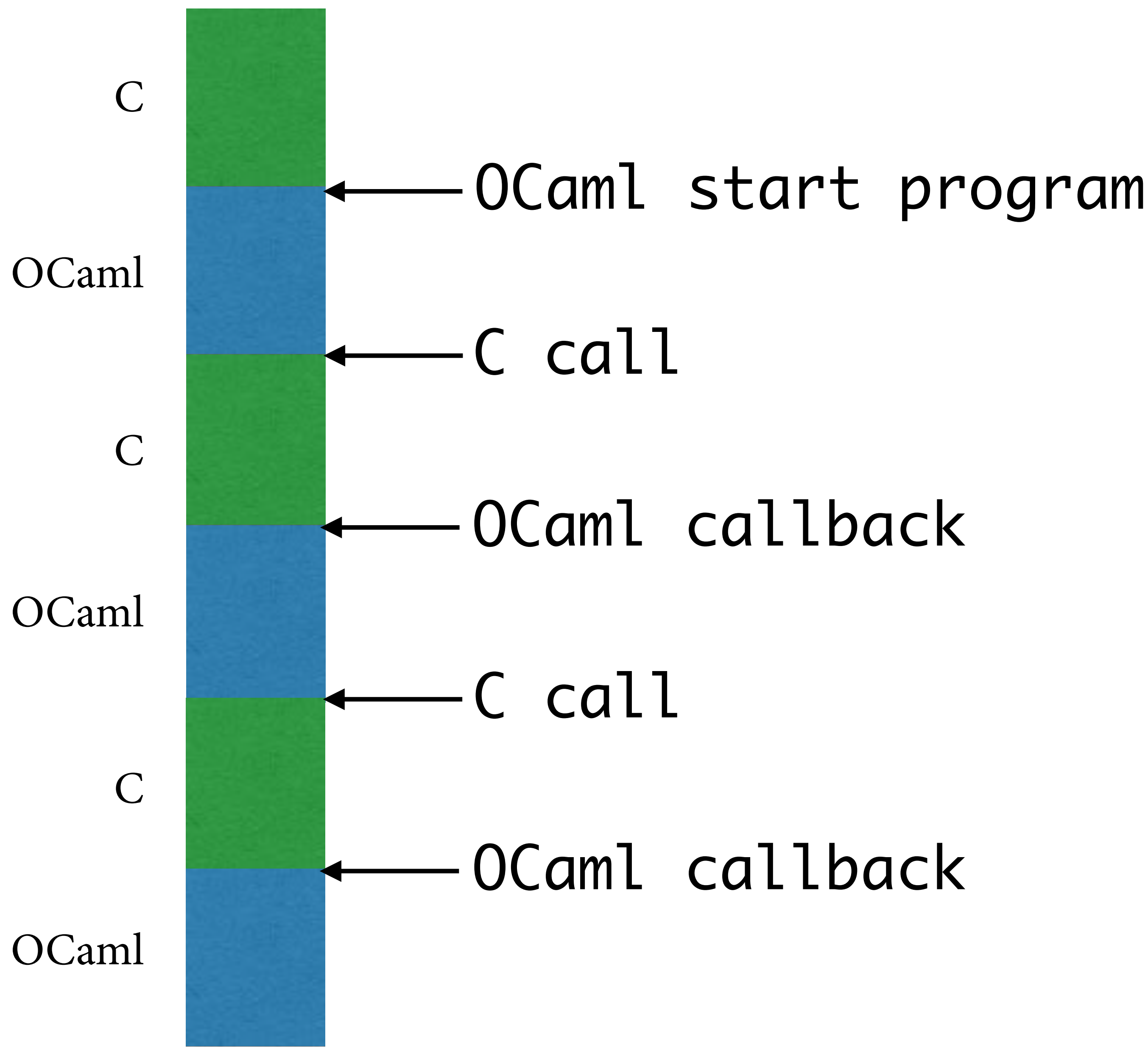


# Implementation

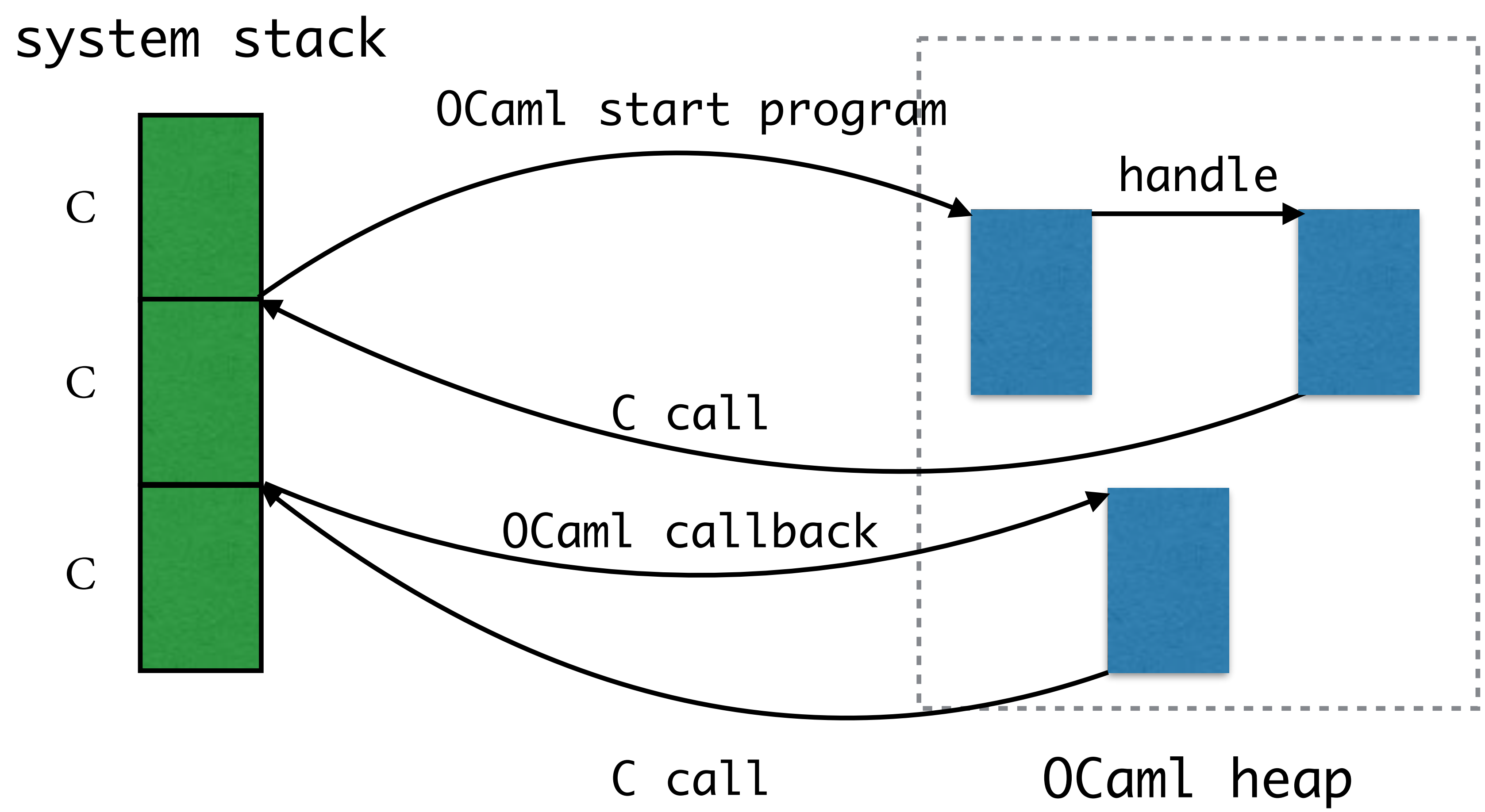
- Fibers: Heap allocated, dynamically resized stacks
  - ~10s of bytes
  - No unnecessary closure allocation costs unlike CPS
- One-shot delimited continuations
  - Simplifies reasoning about resources - sockets, locks, etc.
- Handlers —> Linked-list of fibers



# Native-code fibers — Vanilla



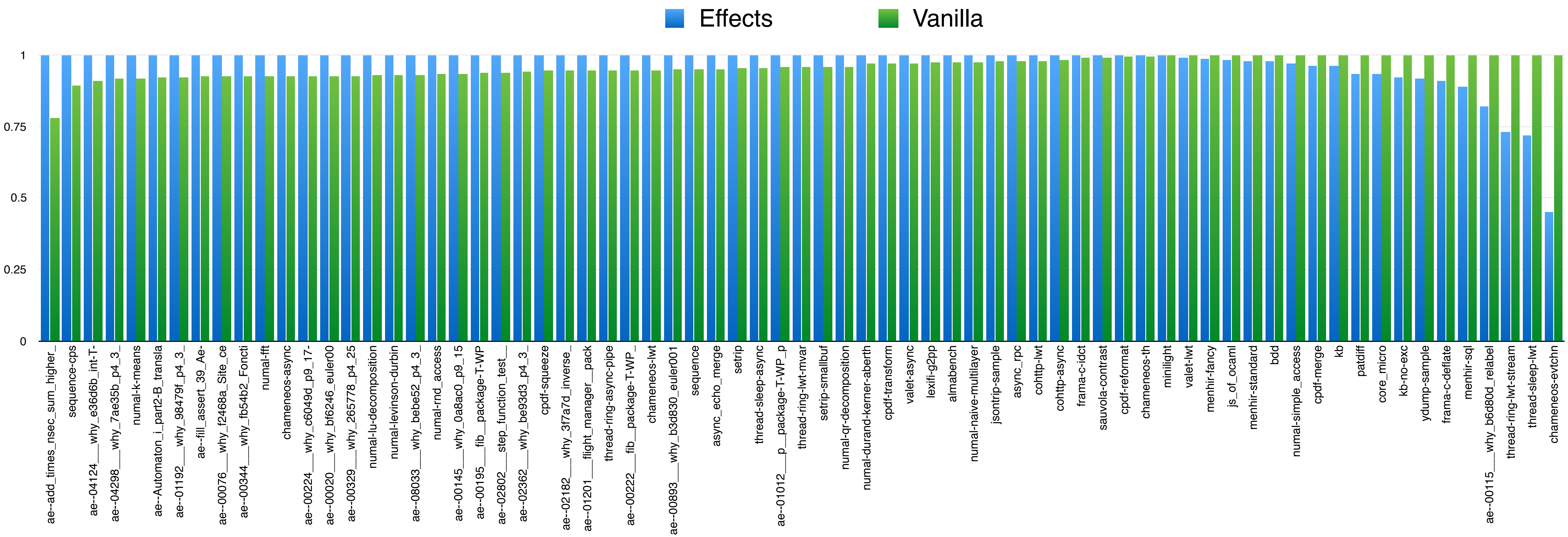
# Native-code fibers — Effects




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}

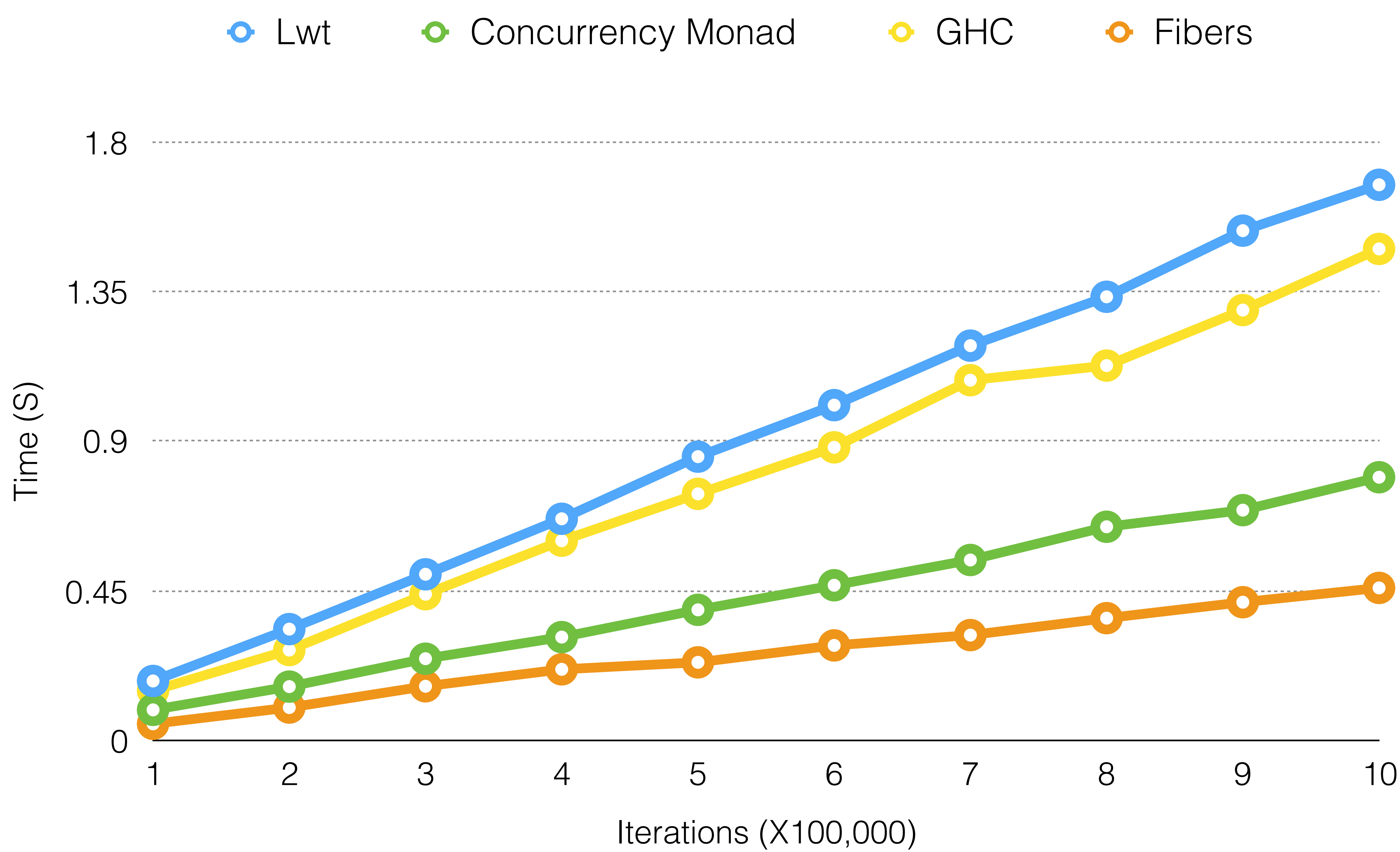
# Performance : Vanilla OCaml

**Normalised time** (lower is better)

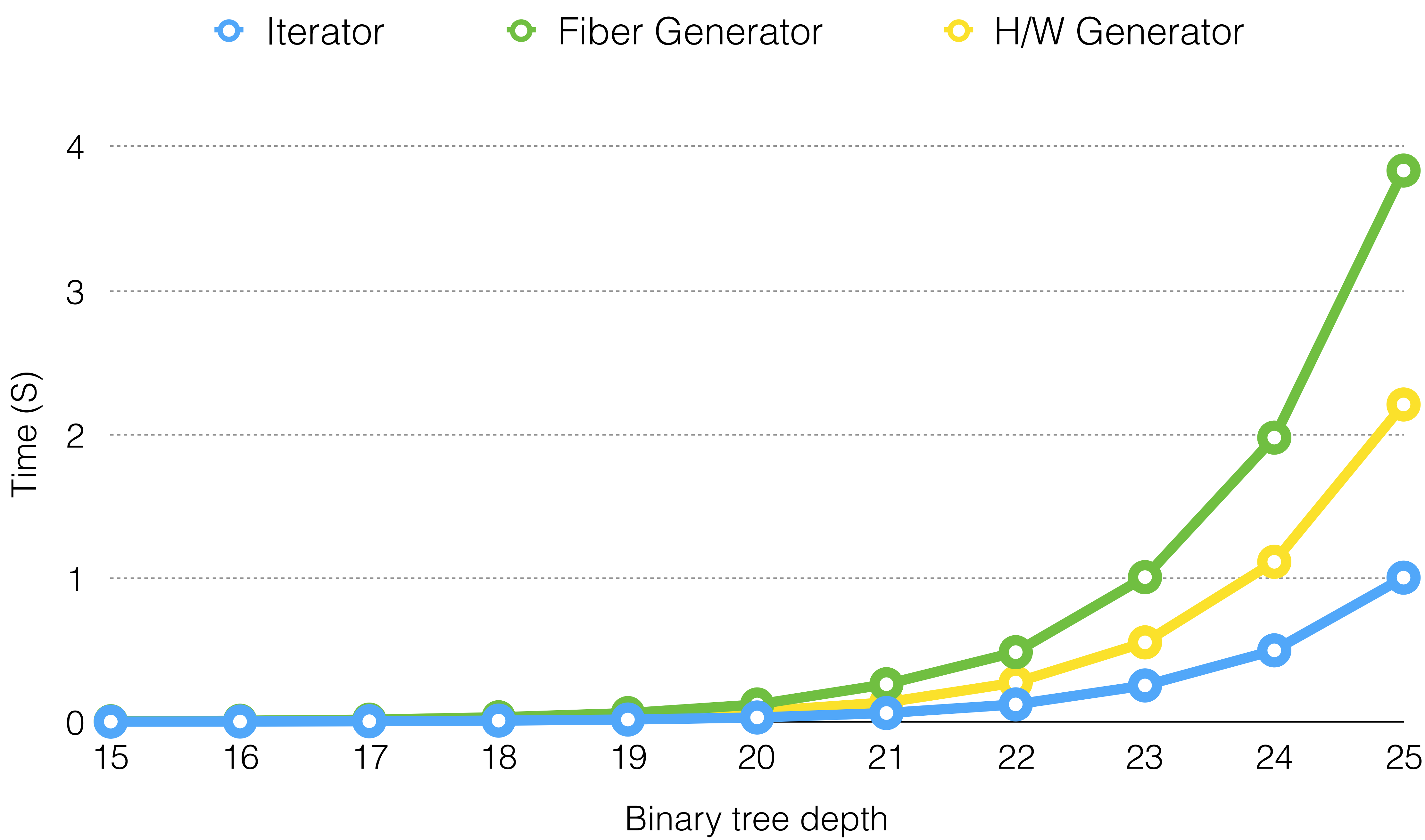



Effects **~0.9%** slower

# Performance : Chameneos-Redux



# Performance : Generator



# Javascript backend

- js_of_ocaml
  - OCaml bytecode —> Javascript
- js_of_ocaml compiler pass
  - Whole-program selective CPS transformation
- Work-in-progress!
  - Selective CPS translation

# *fin.*

**Multicore OCaml:** https://github.com/ocamllabs/ocaml-multicore

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