Concurrent and Parallel Programming with OCaml 5

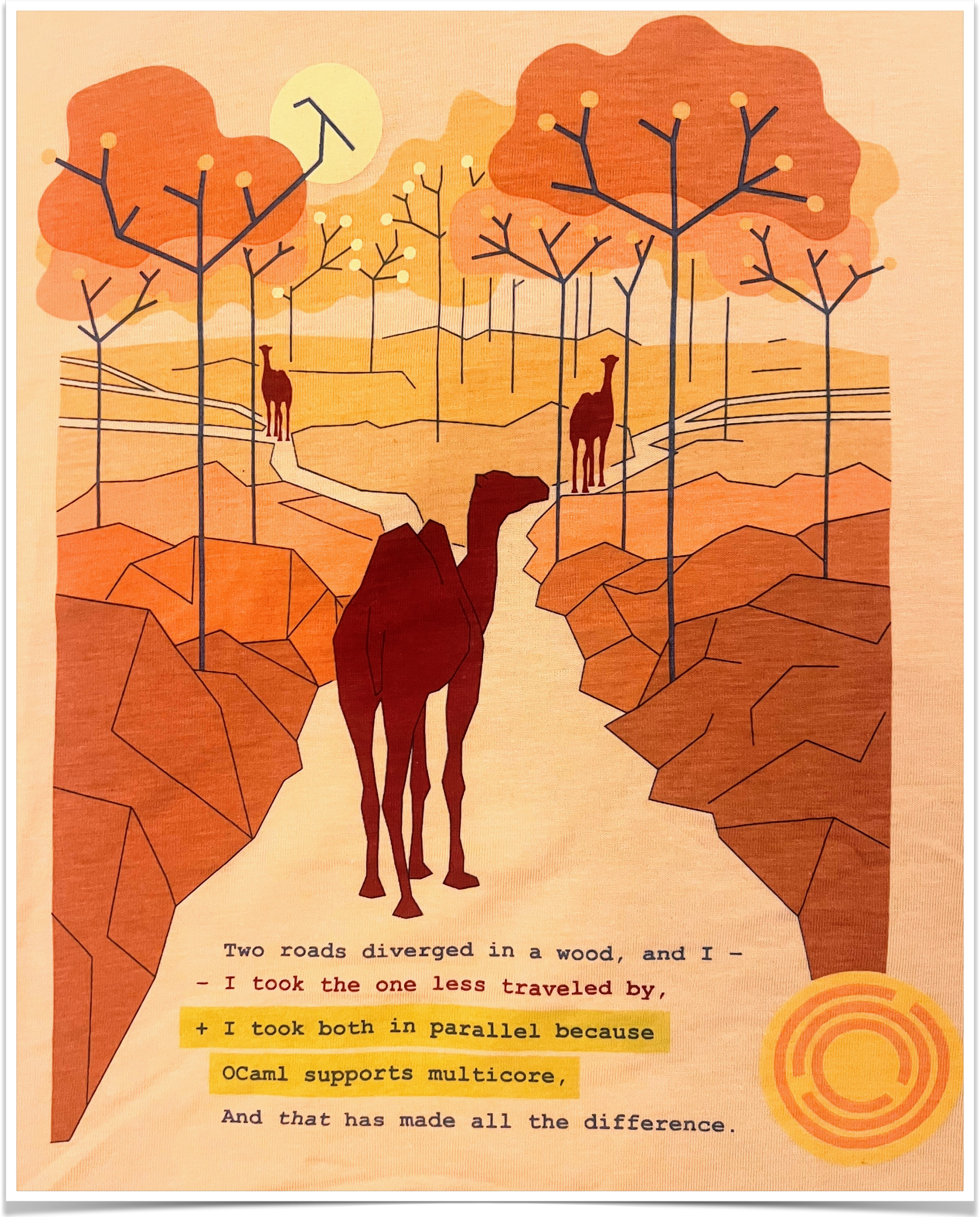
“KC” Sivaramakrishnan

IIT MADRAS

Tarides

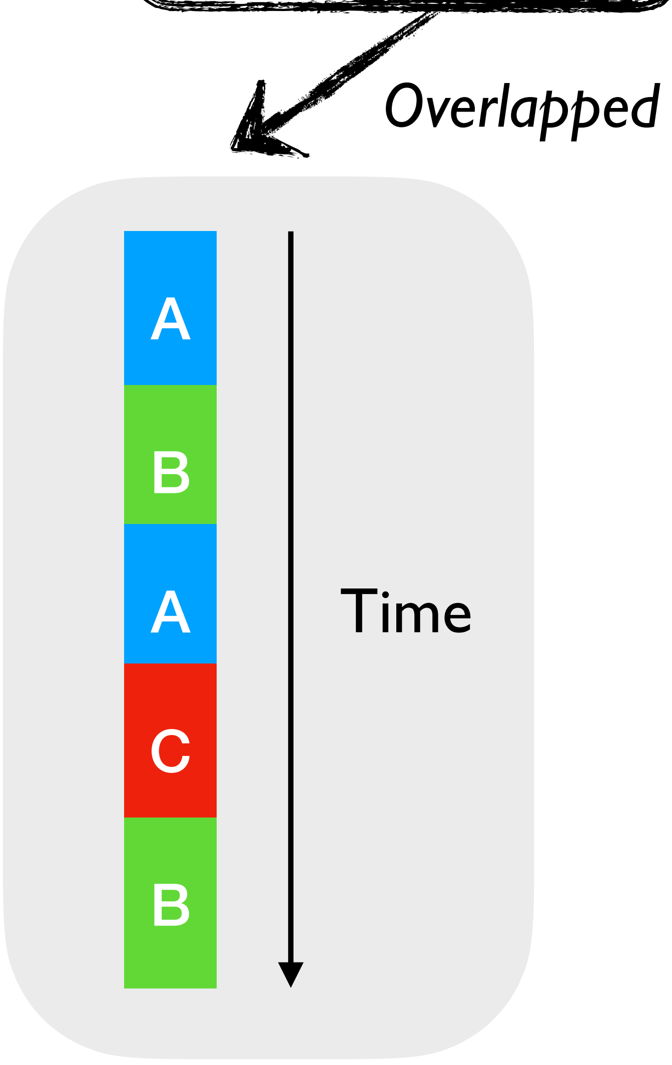
OCaml 5

- Native-support for ***concurrency*** and ***parallelism*** to OCaml
- Started in 2014 as “Multicore OCaml” project
  - OCaml 5.0 released in Dec 2022
  - 5.1 — Sep 2023; 5.2 — May 2024; 5.3 — Jan 2025
- This talk
  - Concurrency
  - Parallelism
  - Experience porting from multi-process to multi-core



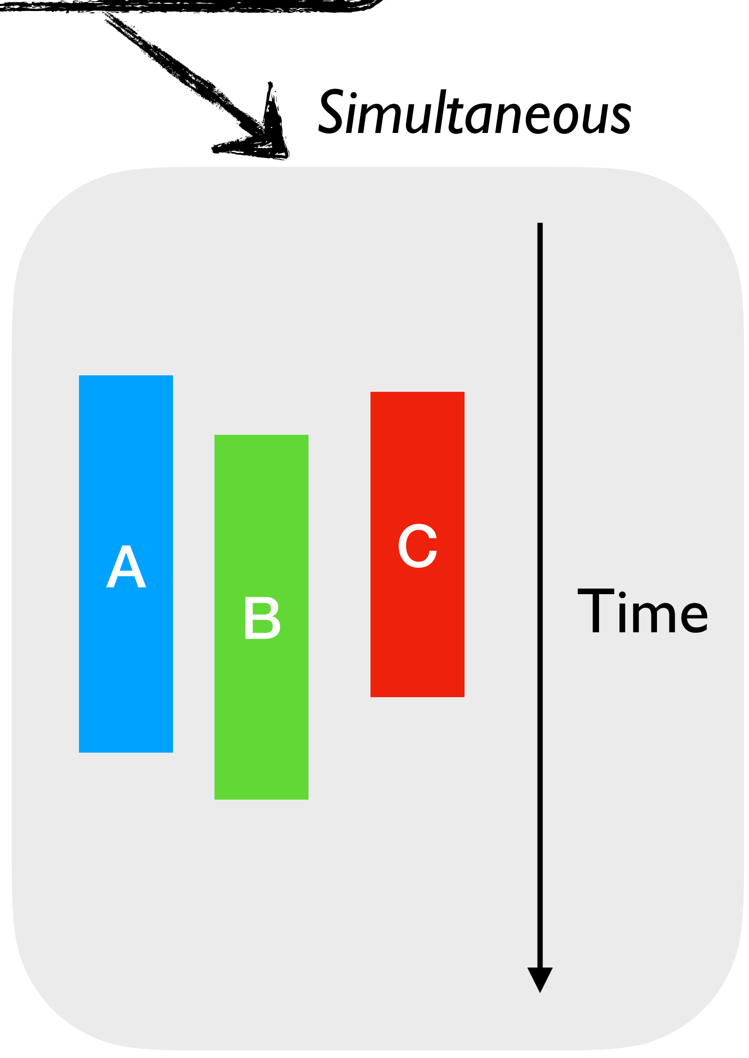
OCaml 5

- Native-support for **concurrency** and **parallelism** to OCaml programming language



Effect Handlers

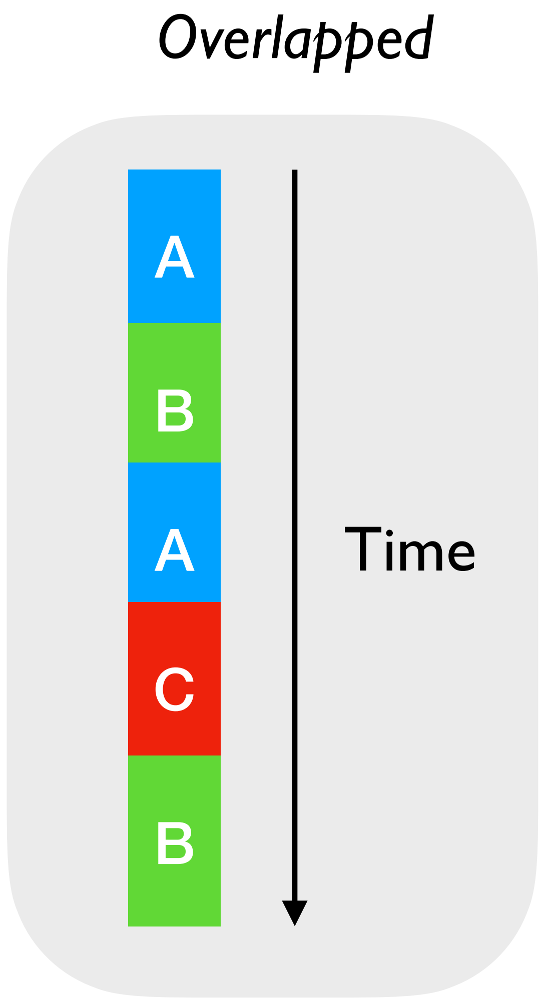
"Retrofitting Effect Handlers onto OCaml", PLDI 2021



Domains

"Retrofitting Parallelism onto OCaml", ICFP 2020

Concurrency

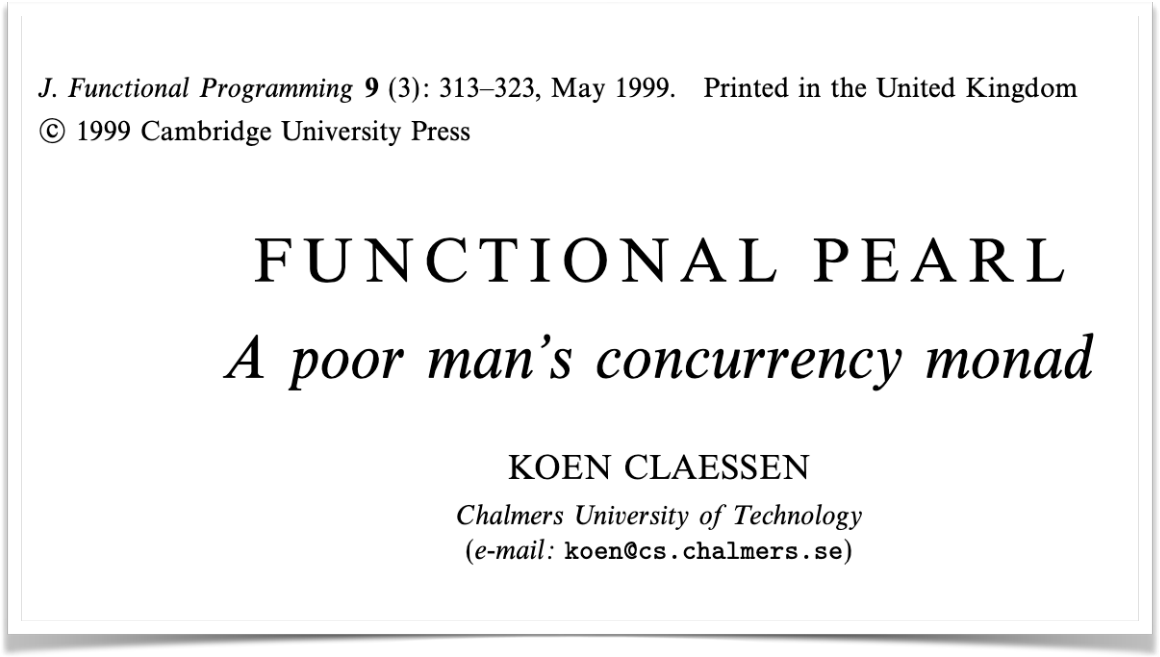


Concurrent Programming

- Computations may be *suspended* and *resumed* later
- Many languages provide concurrent programming mechanisms as *primitives*
    - ✦ **async/await** — JavaScript, Python, Rust, C# 5.0, F#, Swift, …
    - ✦ **generators** — Python, Javascript, …
    - ✦ **coroutines** — C++, Kotlin, Lua, …
    - ✦ **futures & promises** — JavaScript, Swift, …
    - ✦ **Lightweight threads/processes** — Haskell, Go, Erlang
- *Often include many different primitives in the same language!*
    - ✦ JavaScript has async/await, generators, promises, and callbacks

Concurrent Programming in OCaml 4

- No *primitive* support for concurrent programming
- **Lwt** and **Async** - concurrent programming *libraries* in OCaml
  - Callback-oriented programming with *monadic* syntax


J. Functional Programming **9** (3): 313–323, May 1999. Printed in the United Kingdom

© 1999 Cambridge University Press

FUNCTIONAL PEARL

A poor man's concurrency monad

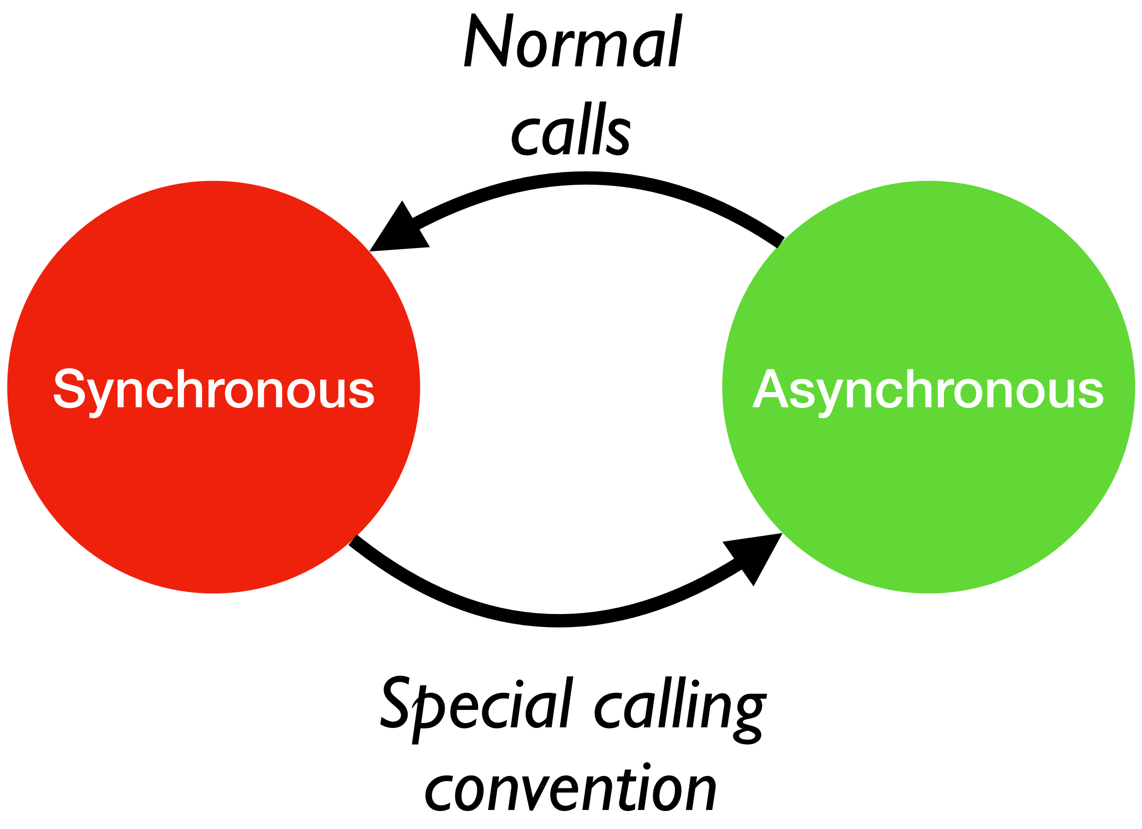
KOEN CLAESSEN

Chalmers University of Technology

(*e-mail:* koen@cs.chalmers.se)

Concurrent Programming in OCaml 4

- No *primitive* support for concurrent programming
- **Lwt** and **Async** - concurrent programming *libraries* in OCaml
  - Callback-oriented programming with *monadic* syntax
- Suffers the pitfalls of callback-orinted programming
  - Incomprehensible (*"callback hell"*), no backtraces, poor performance, function colouring
- **Don't want a zoo of primitives, but need expressivity!**
  - Add the *smallest* primitive that captures *many* concurrent programming patterns



Effect handlers

- A mechanism for programming with *user-defined effects*
- *Modular* and *composable* basis of non-local control-flow mechanisms
  - ✦ Exceptions, generators, lightweight threads, promises, asynchronous IO, coroutines as *libraries*
- Effect handlers ~= *first-class, restartable exceptions*
  - ✦ Structured programming with *delimited continuations*

`https://github.com/ocaml-multicore/effects-examples`

- Direct-style asynchronous I/O
- Generators
- Resumable parsers
- Probabilistic Programming
- Reactive UIs
- ....

Effect handlers

```
type _ eff += E : string eff

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  try
    comp ()
  with effect E, k ->
    print_string "1 ";
    continue k "2 ";
    print_string "4 "
```

effect declaration

suspends current computation

computation

delimited continuation

handler

resume suspended computation

Stepping through the example

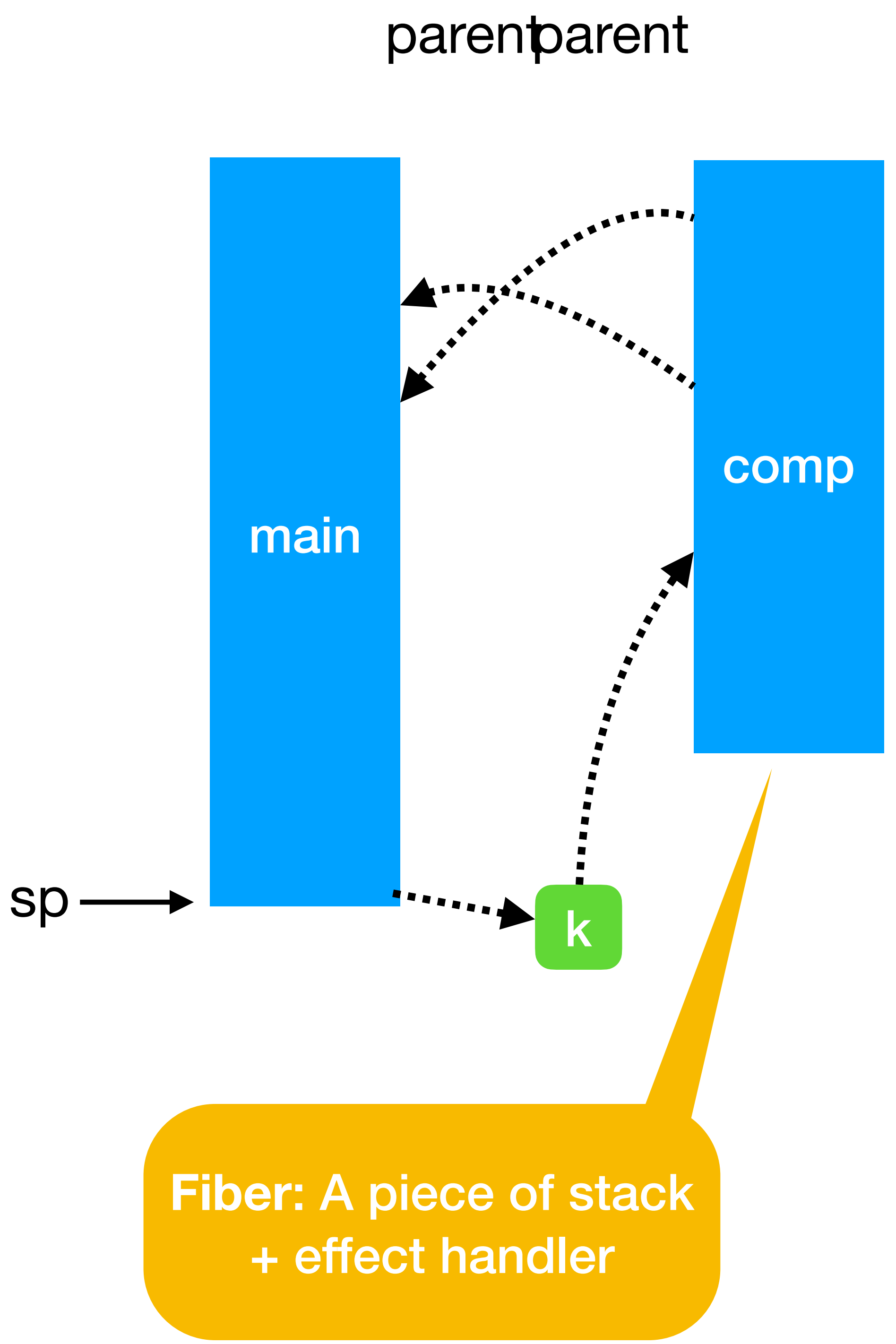
```
type 'a eff += E : string eff

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  try
    comp ()
  with effect E, k ->
    print_string "1 ";
    continue k "2 ";
    print_string "4 "
```

pc → (at `try`)

0 1 2 3 4



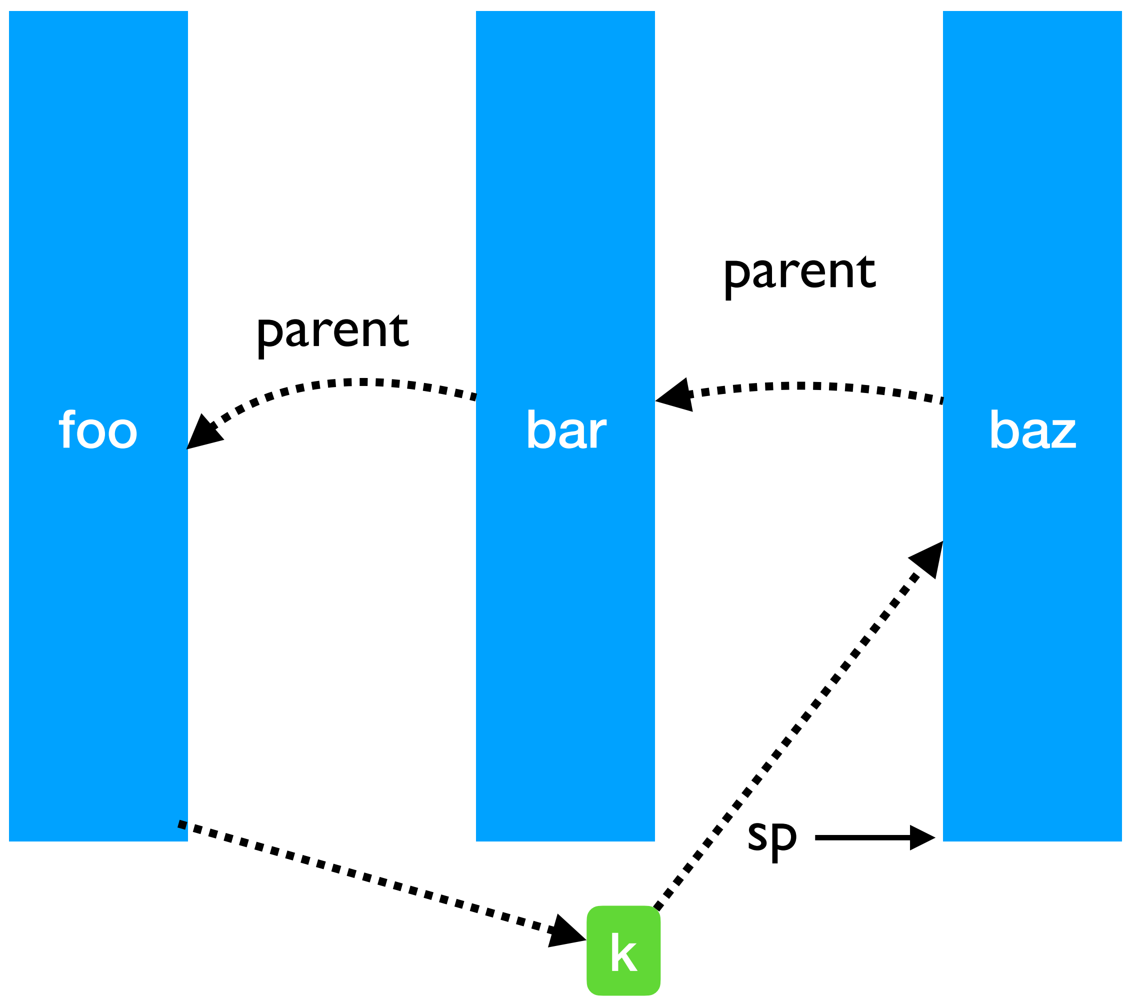
Handlers can be nested

```
type _ eff += A : unit eff
            | B : unit eff

let baz () =
  perform A

let bar () =
  try
    baz ()
  with effect B, k ->
    continue k ()

let foo () =
  try
    bar ()
  with effect A, k ->
    continue k ()
```



- Linear search through handlers
  - *Handler stacks shallow in practice*

Lightweight threading

```
type _ eff += Fork  : (unit -> unit) -> unit eff
            | Yield : unit eff

let run main =
  ... (* assume queue of continuations *)
  let run_next () =
    match dequeue () with
    | Some k -> continue k ()
    | None -> ()
  in
  let rec spawn f =
    match f () with
    | () -> run_next () (* value case *)
    | effect Yield, k -> enqueue k; run_next ()
    | effect (Fork f), k -> enqueue k; spawn f
  in
  spawn main

let fork f = perform (Fork f)
let yield () = perform Yield
```

Lightweight threading

```
let main () =
  fork (fun _ ->
    print_endline "1.a";
    yield ();
    print_endline "1.b");
  fork (fun _ ->
    print_endline "2.a";
    yield ();
    print_endline “2.b")
;;
run main
```

```
1.a
2.a
1.b
2.b
```

Lightweight threading

```
let main () =
  fork (fun _ ->
    print_endline "1.a";
    yield ();
    print_endline "1.b");
  fork (fun _ ->
    print_endline "2.a";
    yield ();
    print_endline “2.b")
;;
run main
```

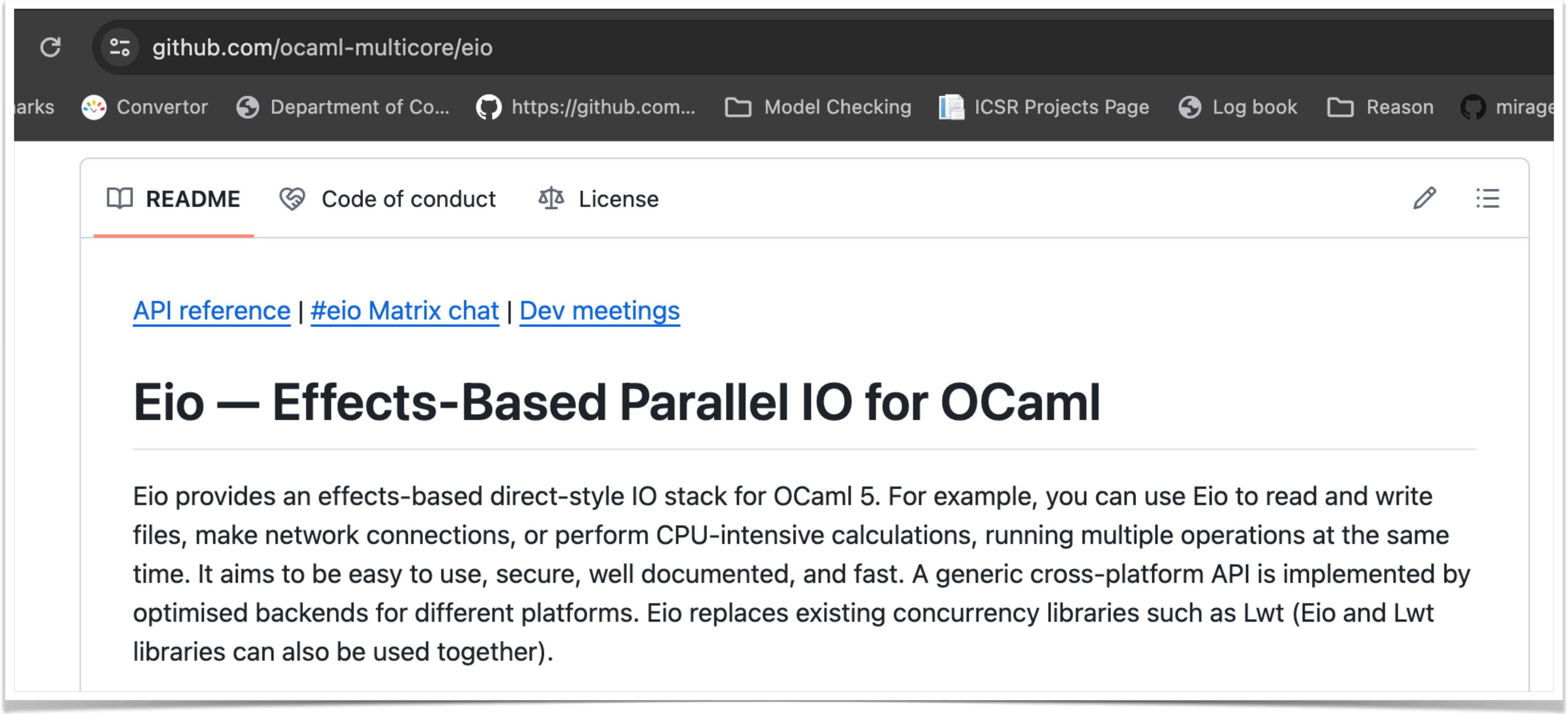
Ability to specialise scheduler unlike GHC Haskell / Go

```
1.a
2.a
1.b
2.b
```

- Direct-style (no monads)
- User-code need not be aware of effects
- No Async vs Sync distinction

Lightweight threading

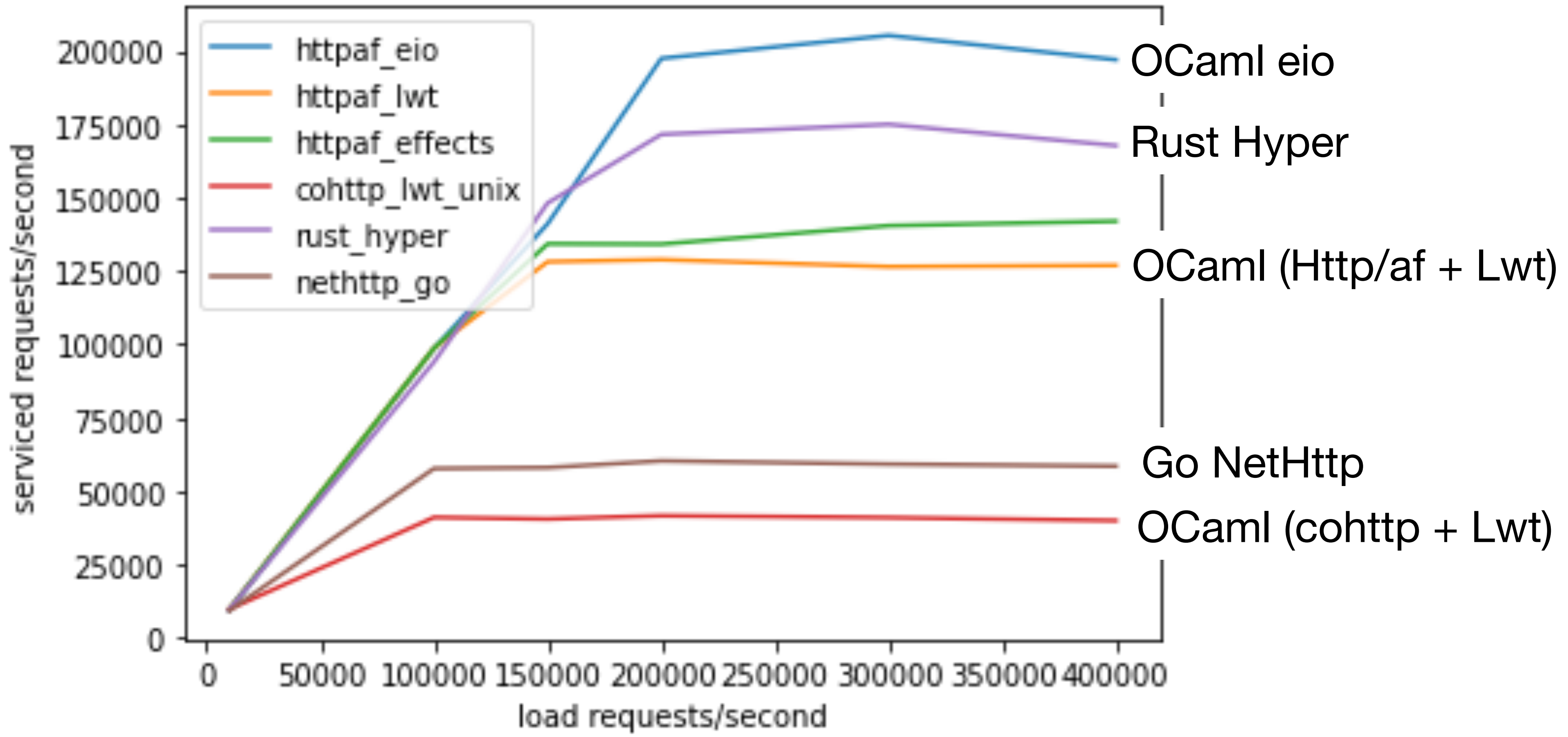
- **eio**: effects-based direct-style I/O
  - ✦ Multiple backends — epoll, select, ***io_uring*** *(new async io in Linux kernel)*



https://github.com/ocaml-multicore/eio

Lightweight threading

- **eio**: effects-based direct-style I/O
  - ✦ Multiple backends — epoll, select, ***io_uring*** *(new async io in Linux kernel)*



100 open connections, 60 seconds w/ io_uring

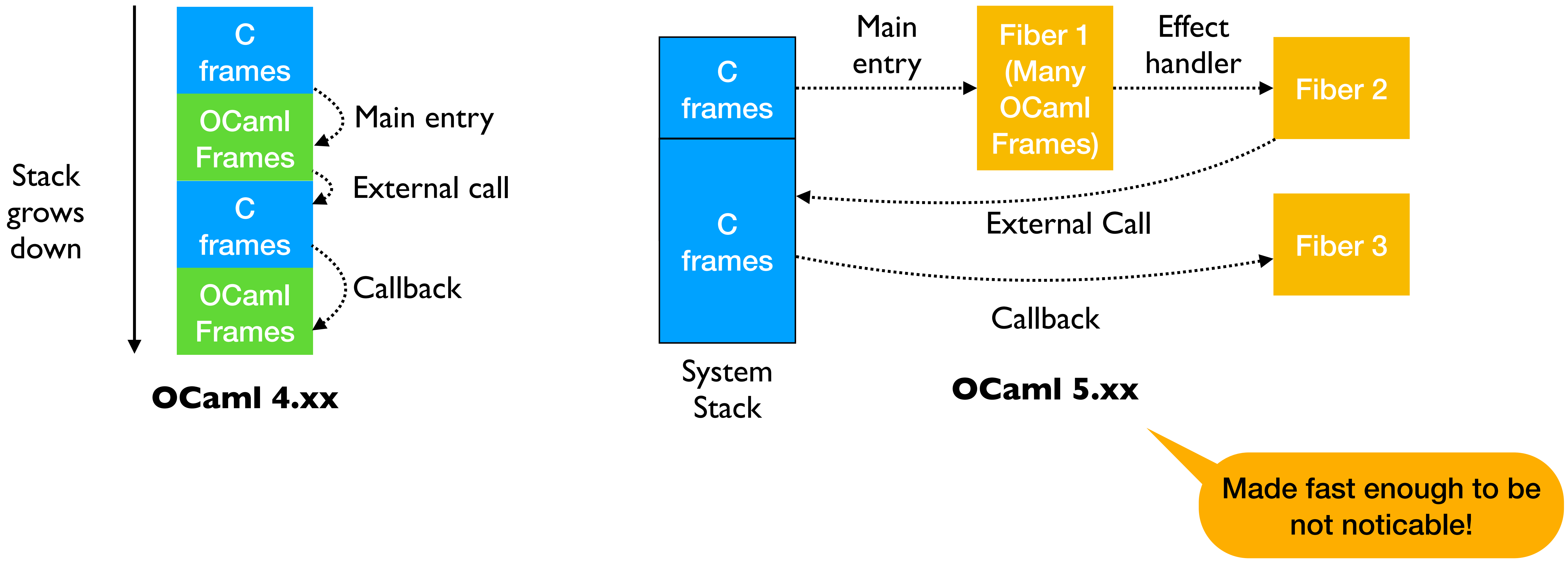
https://github.com/ocaml-multicore/eio

Representing Stack & Continuations

- Program stack is a stack of runtime-managed *dynamically growing* fibers
  - No pointers into the OCaml stack ➔ reallocate fibers on stack overflow
- Stack switching is ***fast!!***
  - One shot continuations ➔ No copying of frames
  - No callee-saved registers in OCaml ➔ No registers to save and restore at switches
  - *Few 10s of intructions; 5 to 10ns for stack switch*
- Need *stack overflow checks* in OCaml function prologue
  - Branch predictor correctly predicts almost always

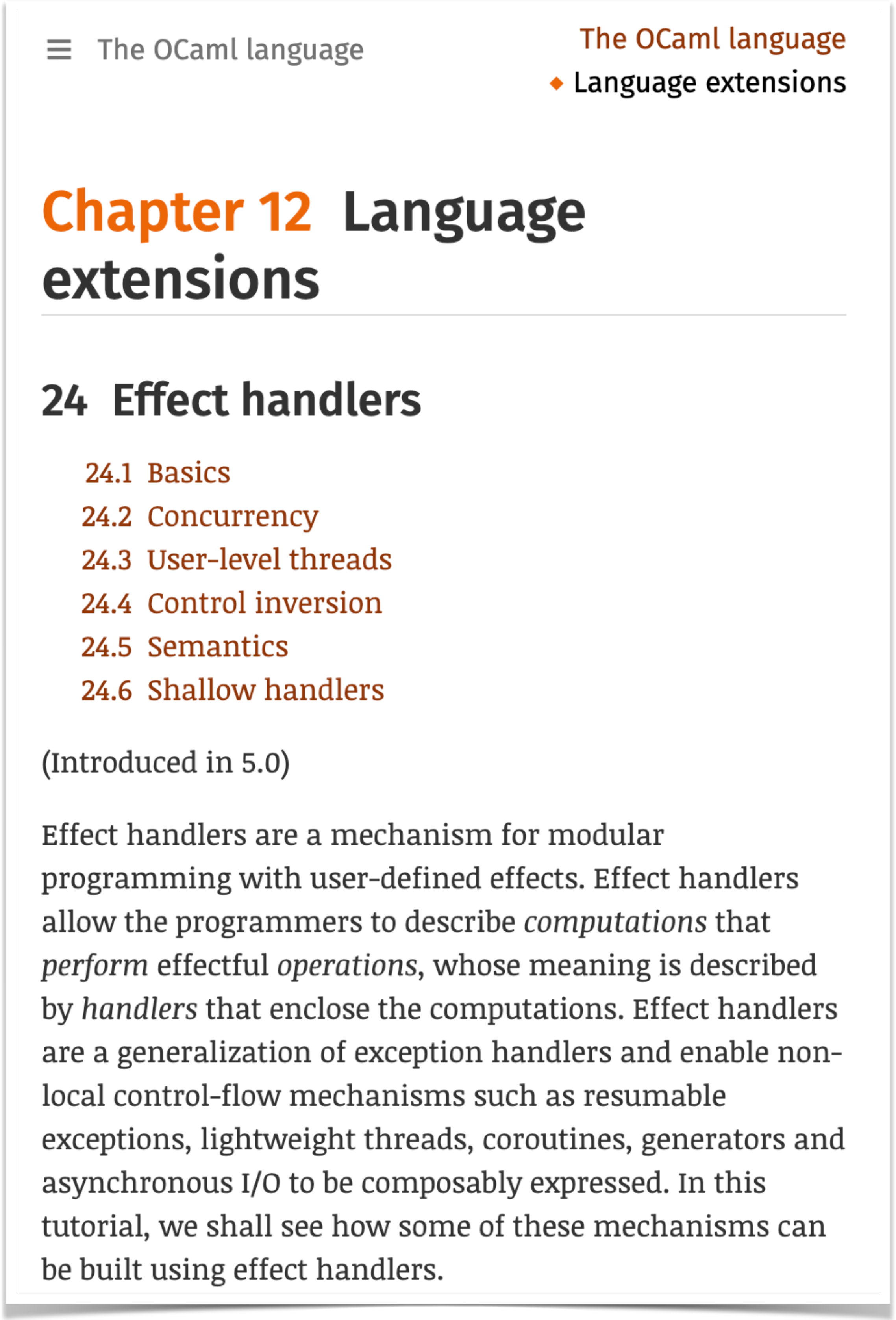
Representing Stack & Continuations

- No stack overflow checks in C code
  - *Need to perform C calls on system stack!*



Summary — Effect Handlers

- Effect handlers brings *simple*, *fast*, *backwards compatible* native concurrency to OCaml
- Support for
  - Integration with GDB (DWARF backtraces)
  - frame-pointers (perf, eBPF)
- No static type system
  - Unhandled effects are runtime errors (just like exceptions)!



≡ The OCaml language

The OCaml language
◆ Language extensions

Chapter 12 Language extensions

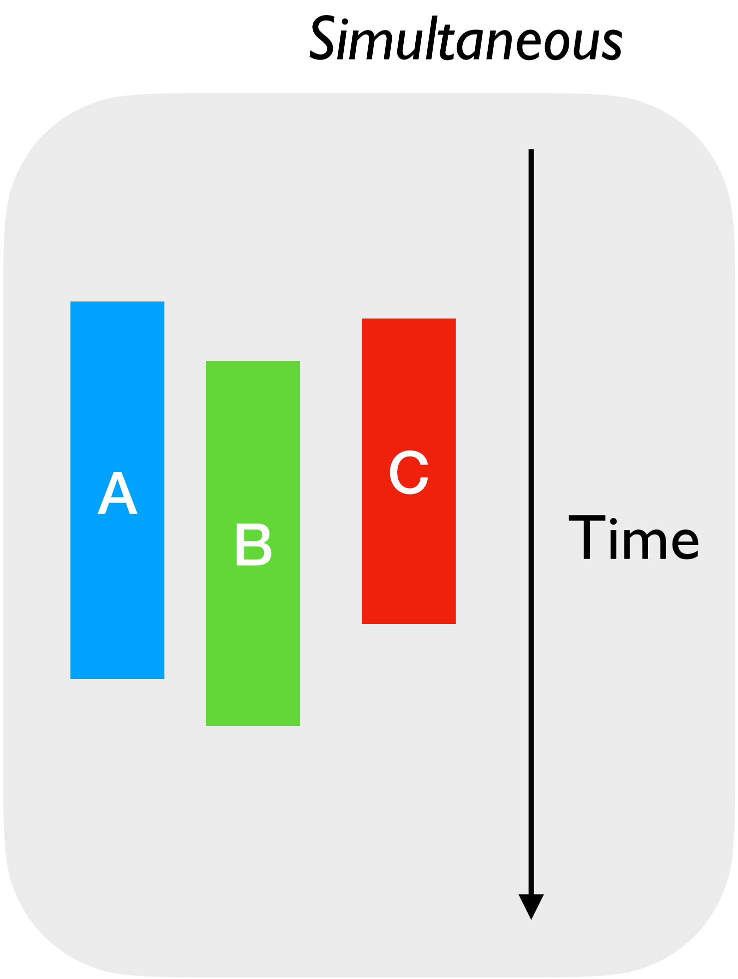
24 Effect handlers

- 24.1 Basics
- 24.2 Concurrency
- 24.3 User-level threads
- 24.4 Control inversion
- 24.5 Semantics
- 24.6 Shallow handlers

(Introduced in 5.0)

Effect handlers are a mechanism for modular programming with user-defined effects. Effect handlers allow the programmers to describe *computations* that *perform* effectful *operations*, whose meaning is described by *handlers* that enclose the computations. Effect handlers are a generalization of exception handlers and enable non-local control-flow mechanisms such as resumable exceptions, lightweight threads, coroutines, generators and asynchronous I/O to be composably expressed. In this tutorial, we shall see how some of these mechanisms can be built using effect handlers.

Parallelism



Domains

- A unit of parallelism
- ***Heavyweight*** — maps onto an OS thread
  - Aim to have 1 domain per physical core
- Stdlib exposes
  - Spawn & join, Mutex, Condition, domain-local storage
  - Atomic references
- Relaxed memory model
  - Data-race-free programs have sequential consistency
  - *Programs with data races are type/memory safe!*
    - Unlike C++, unsafe Rust
    - Important when porting sequential code to be made parallel

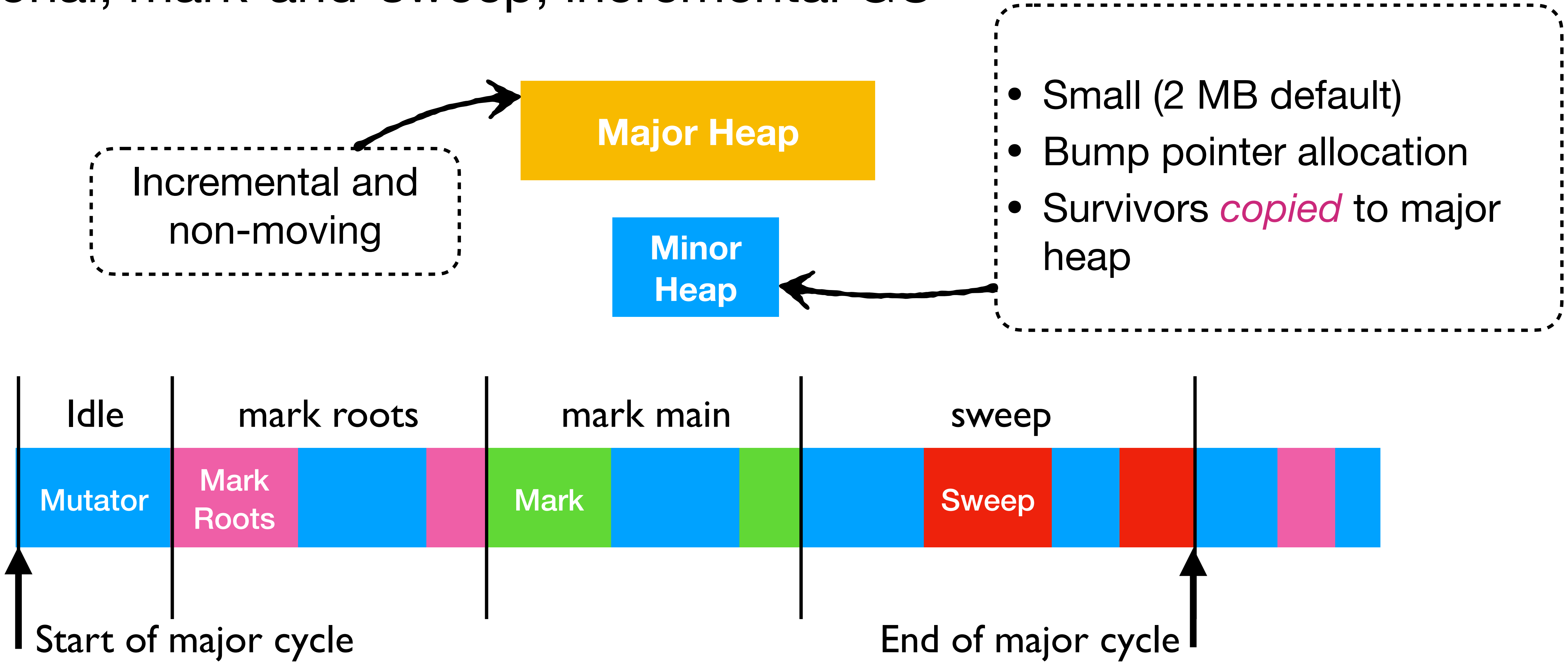
Chapter 10 Memory model: The hard bits

This chapter describes the details of OCaml relaxed memory model. The relaxed memory model describes what values an OCaml program is allowed to witness when reading a memory location. If you are interested in high-level parallel programming in OCaml, please have a look at the parallel programming chapter 9.

This chapter is aimed at experts who would like to understand the details of the OCaml memory model from a practitioner's perspective. For a formal definition of the OCaml memory model, its guarantees and the compilation to hardware memory models, please have a look at the PLDI 2018 paper on Bounding Data Races in Space and Time. The memory model presented in this chapter is an extension of the one presented in the PLDI 2018 paper. This chapter also covers some pragmatic aspects of the memory model that are not covered in the paper.

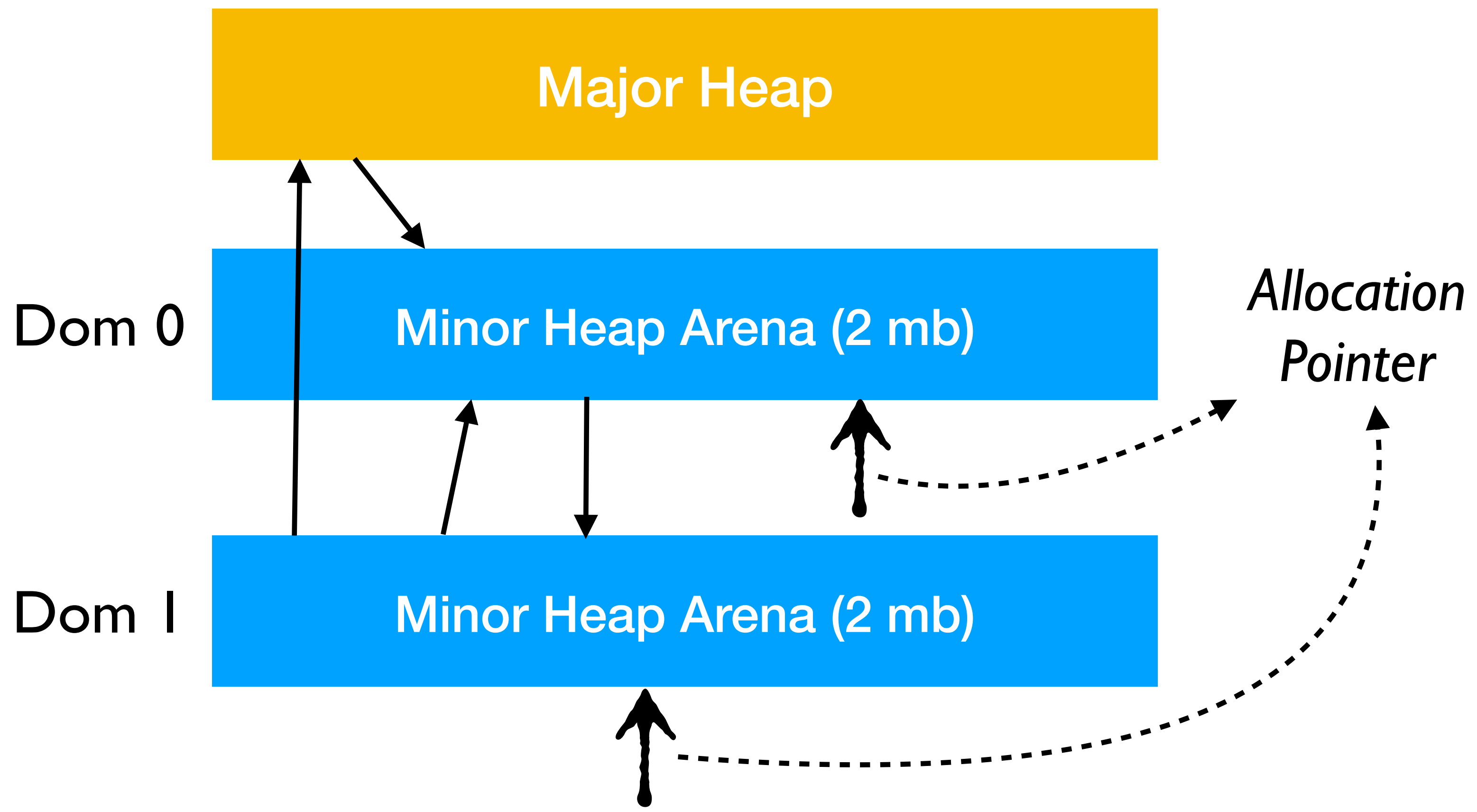
OCaml 4 GC

- Generational, mark-and-sweep, incremental GC



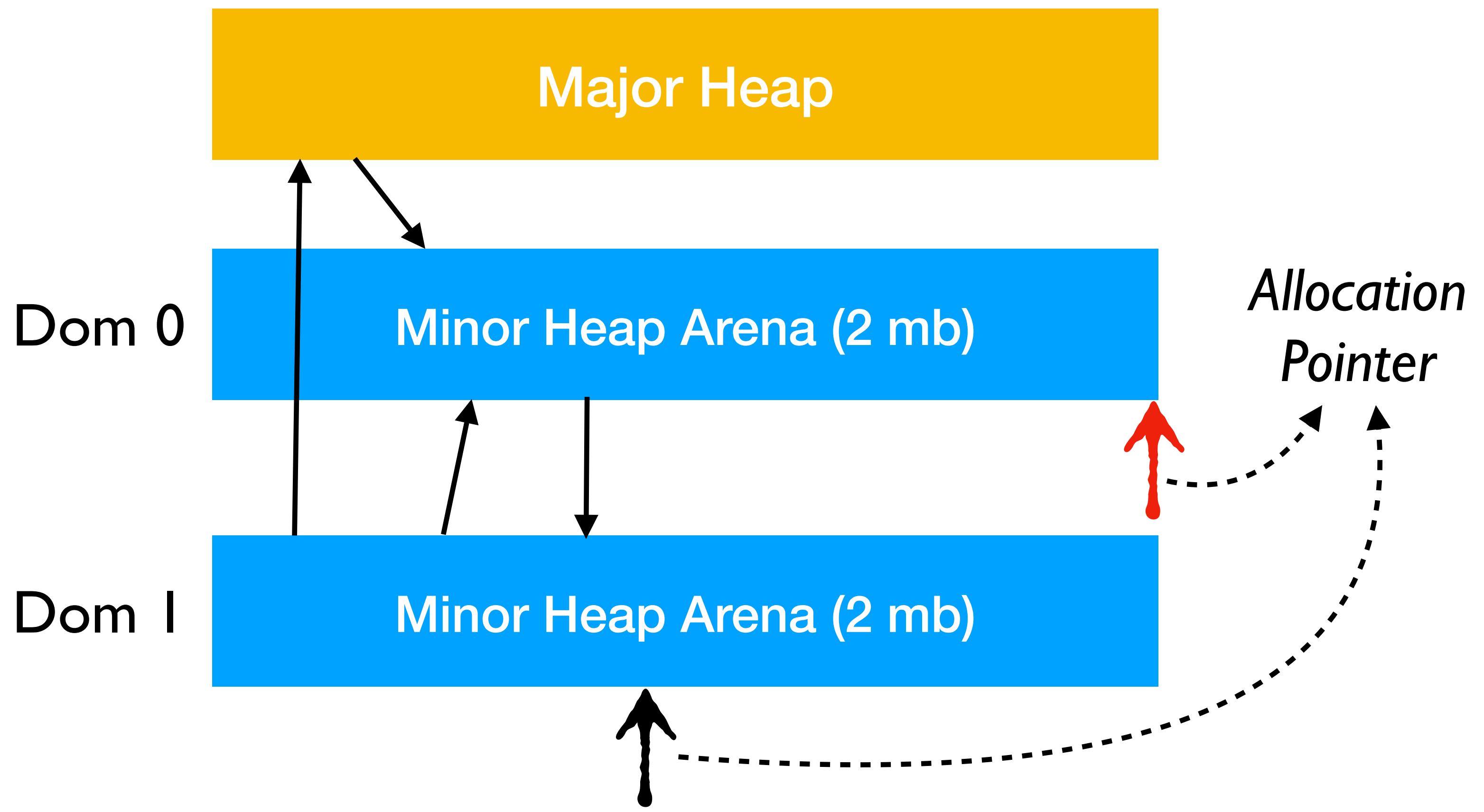
- Fast local allocations
- Max GC latency **< 10 ms**, 99th percentile latency **< 1 ms**

OCaml 5 minor GC



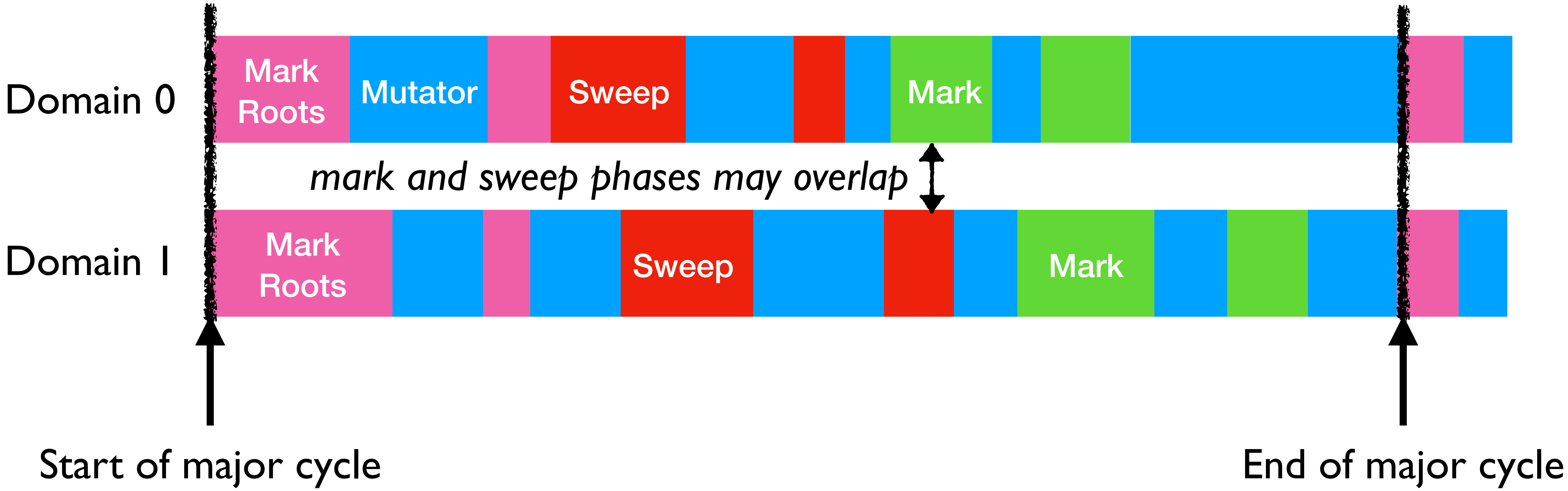
- Private minor heap arenas per domain
  - *Fast allocations without synchronisation*
- No restrictions on pointers between minor heap arenas and major heap

OCaml 5 minor GC



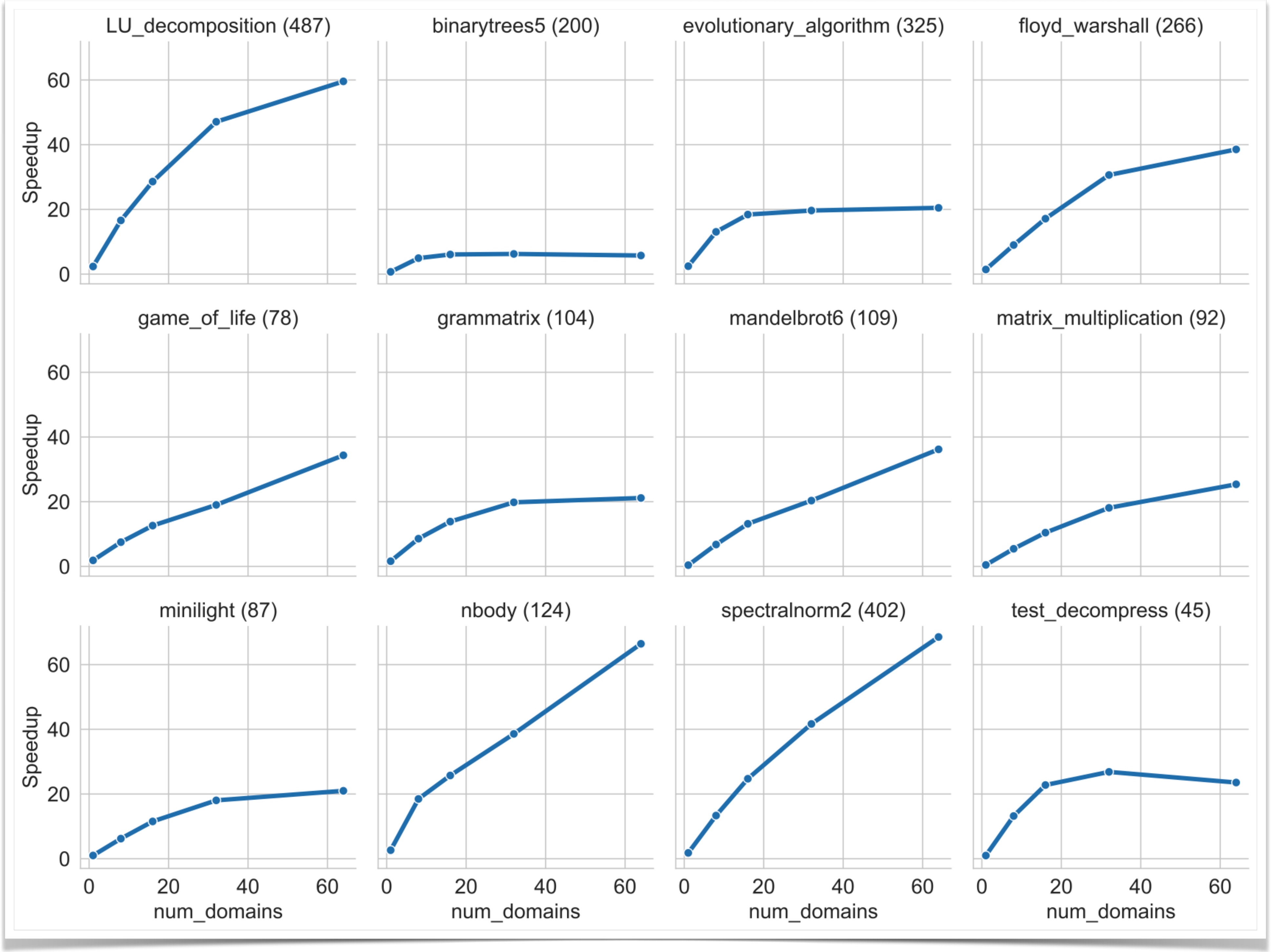
- *Stop-the-world* *parallel* collection for minor heaps
    - 2 barriers / minor gc; (some) work sharing between gc threads
- On 24 cores, w/ default heap size (2MB / arena), ***< 10 ms*** pause for completeing minor GC

OCaml 5 major GC



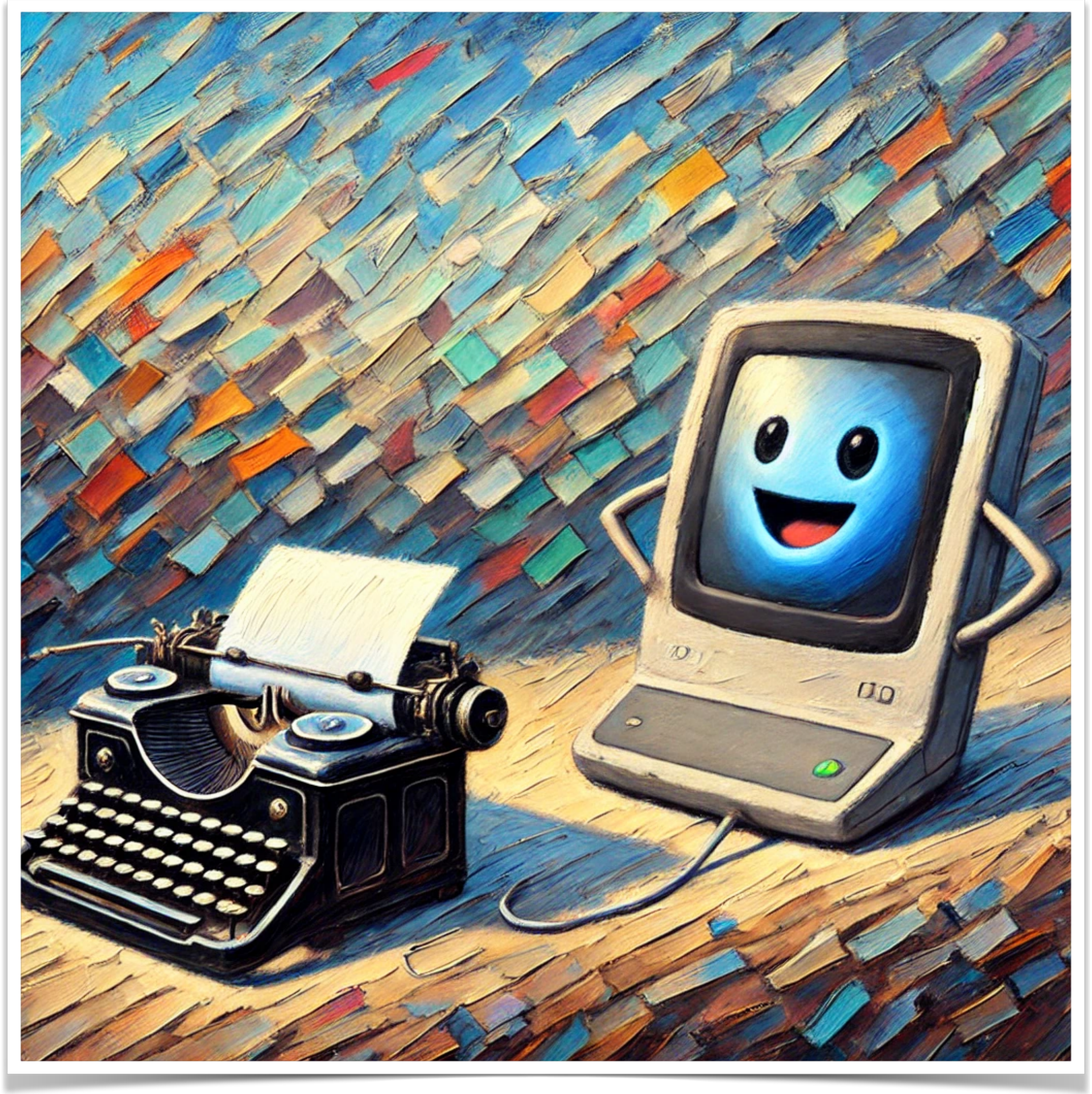
- Mostly concurrent mark-and-sweep GC
- 3 barriers / cycle (when not using ephemerons)
    - 1 each at the end of mark, finalise_first, finalise_last phases
- On 24 cores, **< 5 ms** pauses at barriers
    - Only to agree that the phase has ended

Scalability



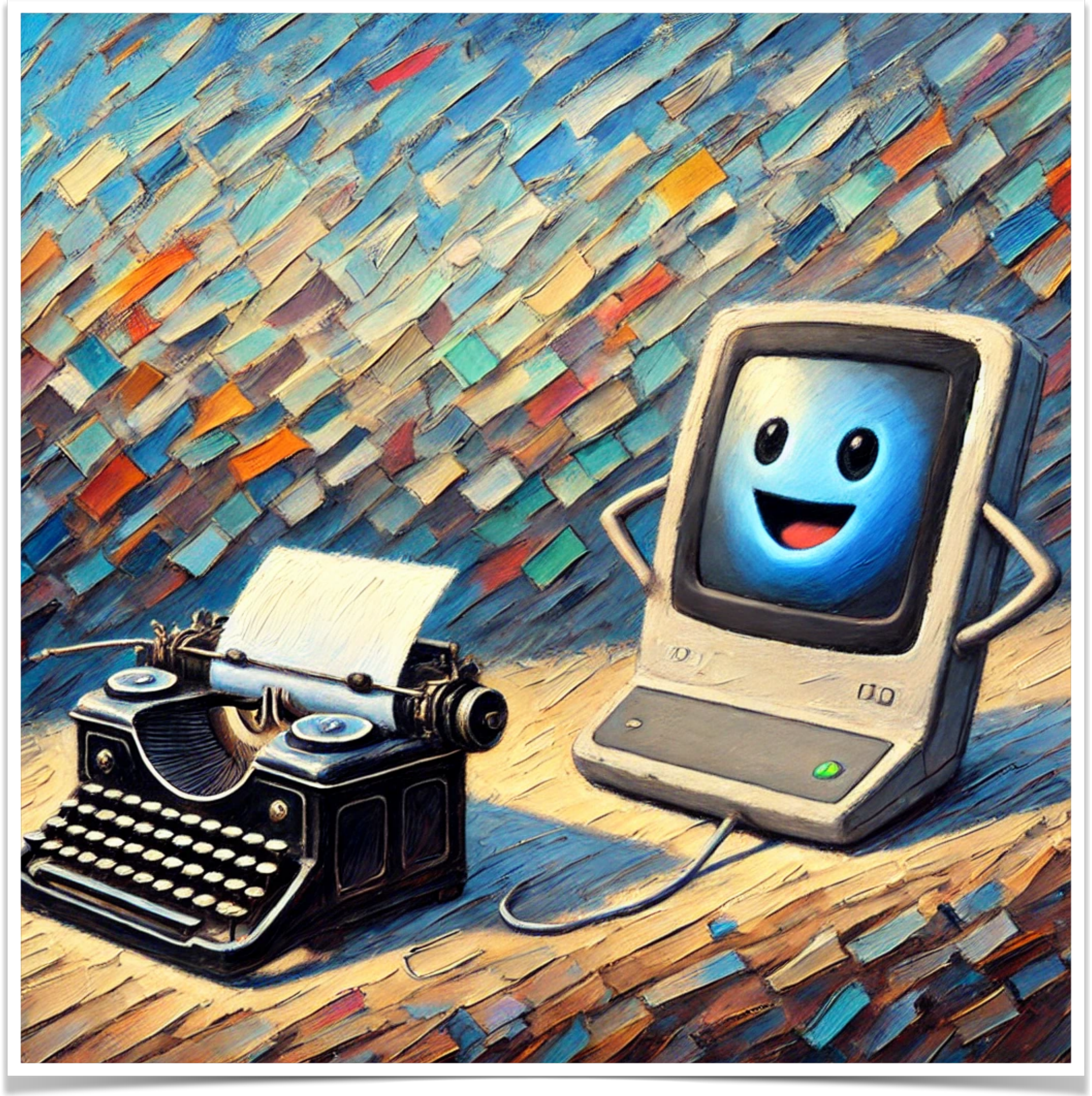
Backwards compatibility

- Both effect handlers and GC designed for *backwards compatibility*
  - Performance, tooling support, features (almost all of them)
- **Performance**
  - OCaml 5 is designed to run sequential programs as well as OCaml 4
  - *Any significant performance regressions (5%+) is a bug*; please report it!



Backwards compatibility

- **Feature set**
  - All of the language including finalisers, weak references, ephemerons, systhreads supported
    - Compaction (manual) is manual, no naked pointers
  - Programs with data races are *type and memory safe*!
  - Racy use of Stdlib may yield surprising results, but *will not crash*!
    - think Queue, Hashtbl, Lazy, Unix, etc.
- **Existing tools continue to work**
  - GDB, perf, eBFP, statmemprof



Porting Applications to OCaml 5

Based on work done by Thomas Leonard @ Tarides
https://roscidus.com/blog/blog/2024/07/22/performance-2/

Solver service

- ocaml-ci — CI for OCaml projects
  - Free to use for the OCaml community
  - Build and run tests on a matrix of platforms on *every commit*
    - **OCaml compilers** (4.02 — 5.2), **architectures** (32- and 64-bit x86, ARM, PPC64, s390x), **OSes** (Alpine, Debian, Fedora, FreeBSD, macOS, OpenSUSE and Ubuntu, in multiple versions)
- Select compatible versions of its dependencies
  - ~1s per solve; 132 solver runs per commit!
- Solves are done by solver-service
  - 160-core ARM machine
  - Lwt-based; sub-process based parallelism for solves
- *Port it to OCaml 5 to take advantage of better concurrency and shared-memory parallelism*

Solver service in OCaml 5

- Used Eio to port from *multi-process* parallel to *shared-memory* parallel
  - Support for asynchronous IO (incl ***io_uring***!) and parallelism
  - *Structured concurrency* and *switches* for resource management
- Outcome
  - Simple code, more stable (switches), removal of lots of communication logic
  - No function colouring!
    - Reclaim the use of `try…with`, `for` and `while` loops!
- Used TSan to ensure that data races are removed

ThreadSanitizer (since 5.2)

- Detect data races dynamically
- Part of the LLVM project — C++, Go, Swift

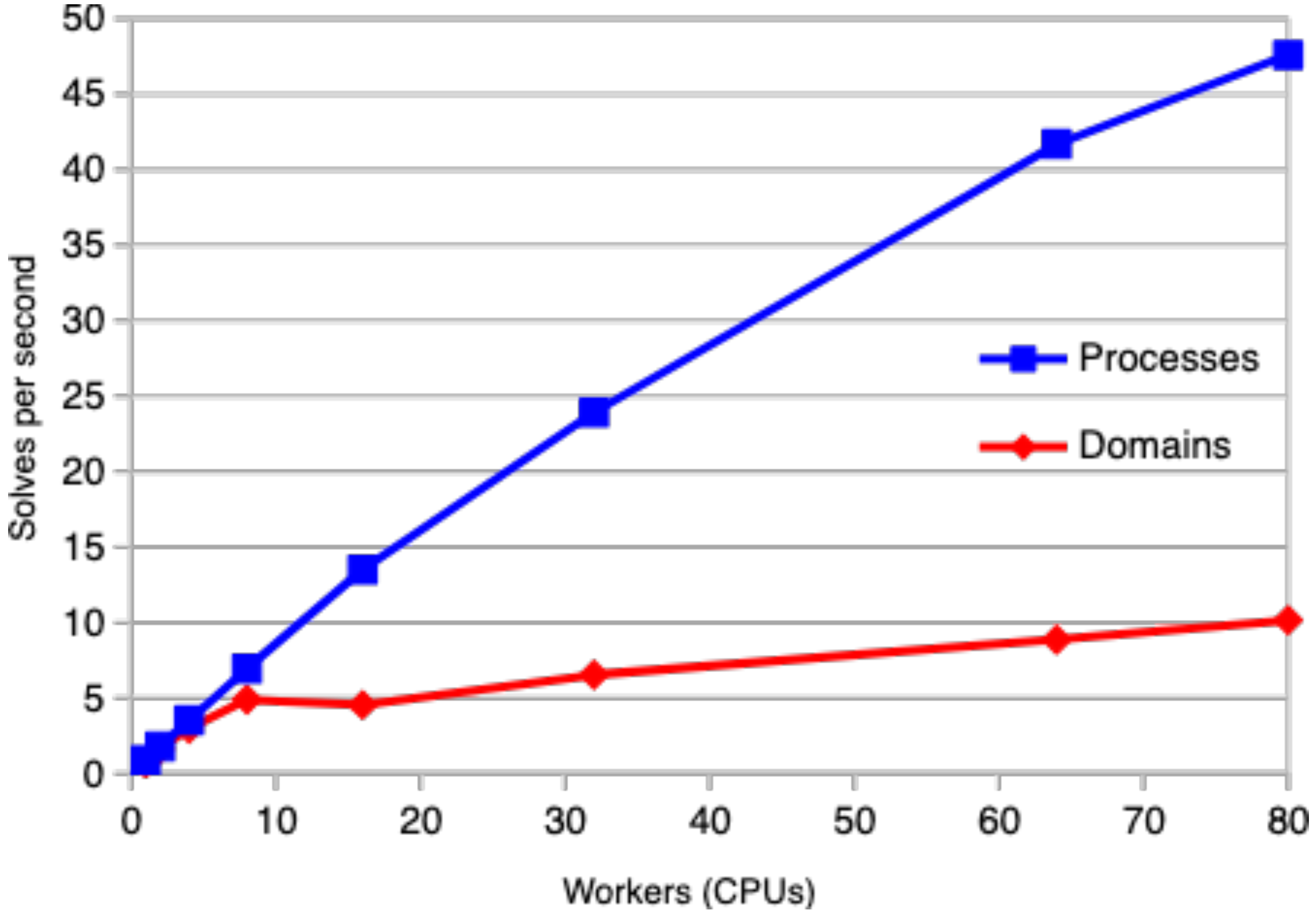
```ocaml
1  let a = ref 0 and b = ref 0
2
3  let d1 () =
4    a := 1;
5    !b
6
7  let d2 () =
8    b := 1;
9    !a
10
11 let () =
12   let h = Domain.spawn d2 in
13   let r1 = d1 () in
14   let r2 = Domain.join h in
15   assert (not (r1 = 0 && r2 = 0))
```

```
==================
WARNING: ThreadSanitizer: data race (pid=3808831)
  Write of size 8 at 0x8febe0 by thread T1 (mutexes: write M9(
    #0 camlSimple_race.d2_274 simple_race.ml:8 (simple_race.ex
    #1 camlDomain.body_706 stdlib/domain.ml:211 (simple_race.e
    #2 caml_start_program <null> (simple_race.exe+0x47cf37)
    #3 caml_callback_exn runtime/callback.c:197 (simple_race.e
    #4 domain_thread_func runtime/domain.c:1167 (simple_race.e

  Previous read of size 8 at 0x8febe0 by main thread (mutexes
    #0 camlSimple_race.d1_271 simple_race.ml:5 (simple_race.ex
    #1 camlSimple_race.entry simple_race.ml:13 (simple_race.ex
    #2 caml_program <null> (simple_race.exe+0x41ffb9)
    #3 caml_start_program <null> (simple_race.exe+0x47cf37)
[...]
```

Eio solver service performance

- … was underwhelming ….initially

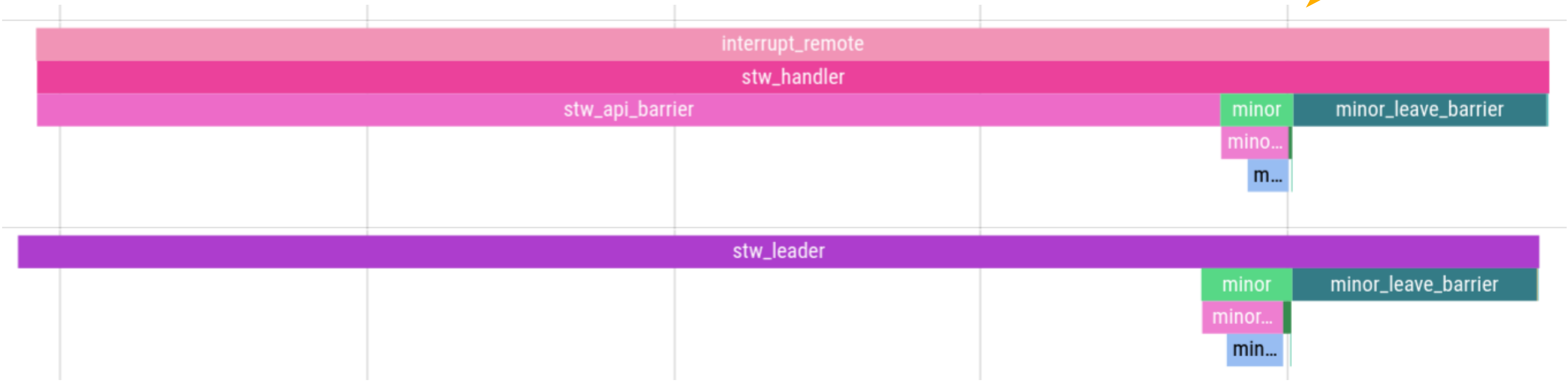


Performance analysis

- perf (incl. call graph), eBFP works
  - Frame-pointers across effect handlers!
- Runtime Events
  - *Every OCaml 5 program has tracing support built-in*
  - Events are written to a shared ring buffer that can be read by an external process

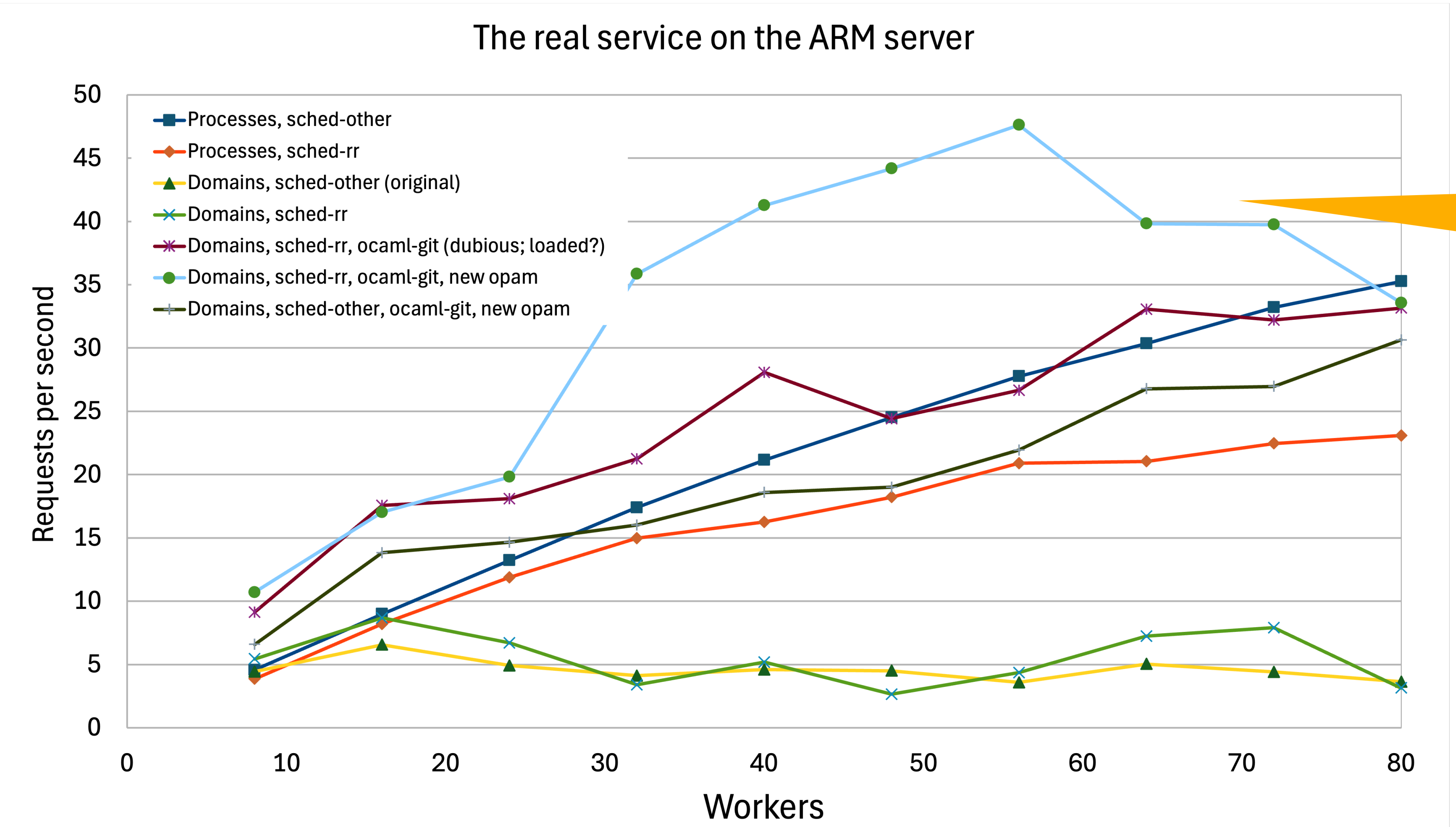
```
$ olly trace foo.trace foo.exe
```

https://perfetto.dev/



Problem indentified

- Switch from `sched_other` to `sched_rr`
- `git log` for each solve to find earliest commit
  - 50ms penalty for STW subprocess spawn
  - Avoid by implementing it in OCaml



Takeaways for introducing shared-memory parallellism

- Use Eio for concurrency and parallelism in OCaml 5
  - Makes your asynchronous IO program more reliable
- Other libraries
  - **Saturn**: Verified multicore safe data structures
  - **Kcas**: Software transactional memory for OCaml
- Use TSan to remove data races
  - Data races will not lead to crashes
- Expect that the initial performance may be underwhelming
  - Existing external tools such as perf, eBPF based profiling, statmemprof continue to work
  - New tools are available on OCaml 5 enabled through *runtime events* — Olly, eio-trace, etc.