Multicore OCaml

What's coming in 2021

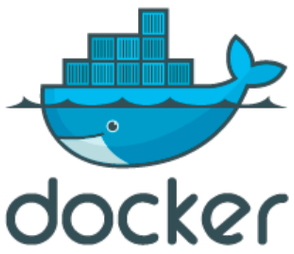
"KC" Sivaramakrishnan and Anil Madhavapeddy

OCaml Labs

IIT MADRAS

UNIVERSITY OF CAMBRIDGE

Industry



Tarides

Projects

The Astrée Static Analyzer

COMPCERT

Industry

Projects

ahrefs

Jane Street

FACEBOOK

docker

Tarides

Tezos

Bloomberg

The Astrée Static Analyzer

flow

hack

COMPCERT

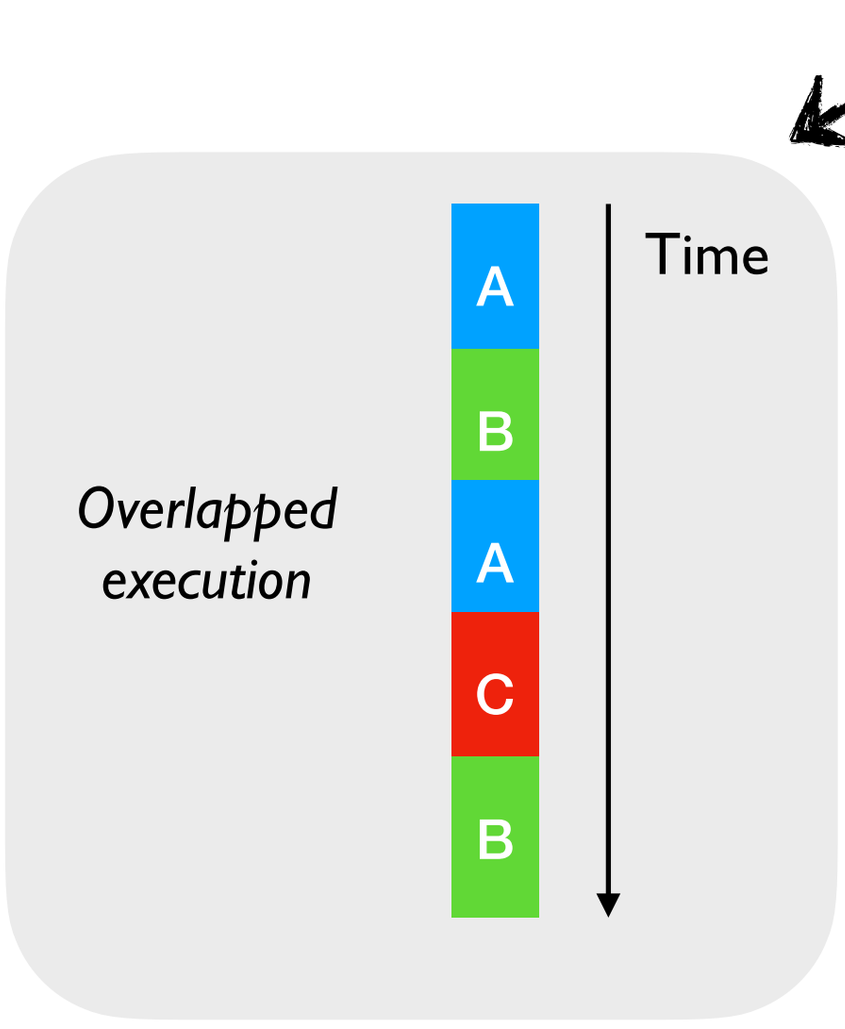
No multicore support!

Multicore OCaml

- Adds native support for *concurrency* and *parallelism* to OCaml

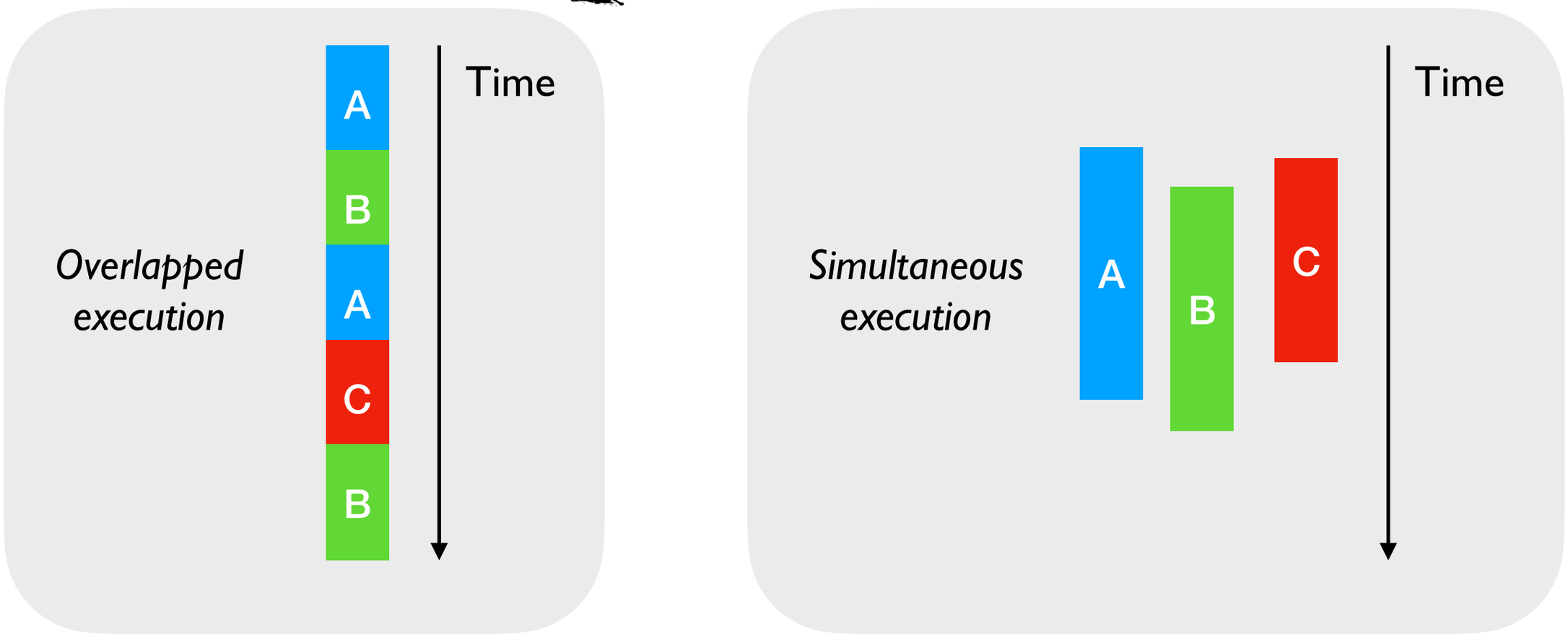
Multicore OCaml

- Adds native support for *concurrency* and *parallelism* to OCaml



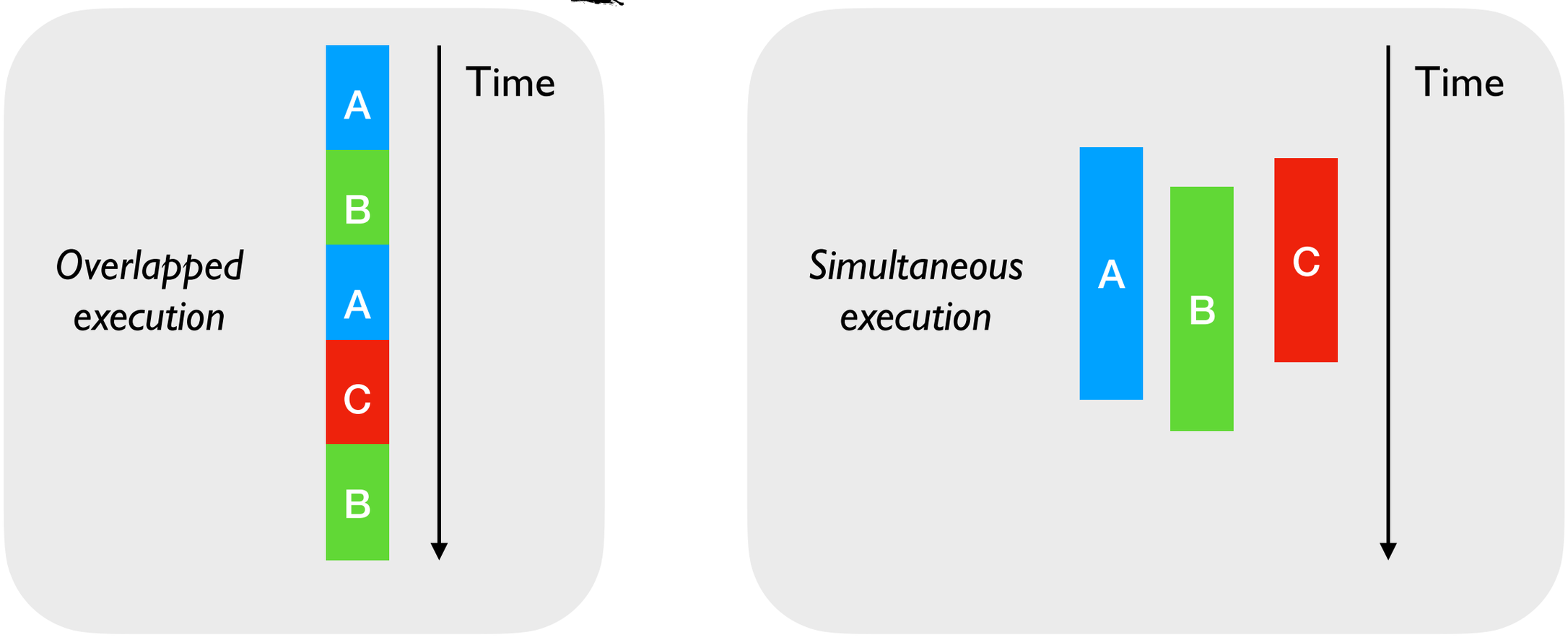
Multicore OCaml

- Adds native support for *concurrency* and *parallelism* to OCaml



Multicore OCaml

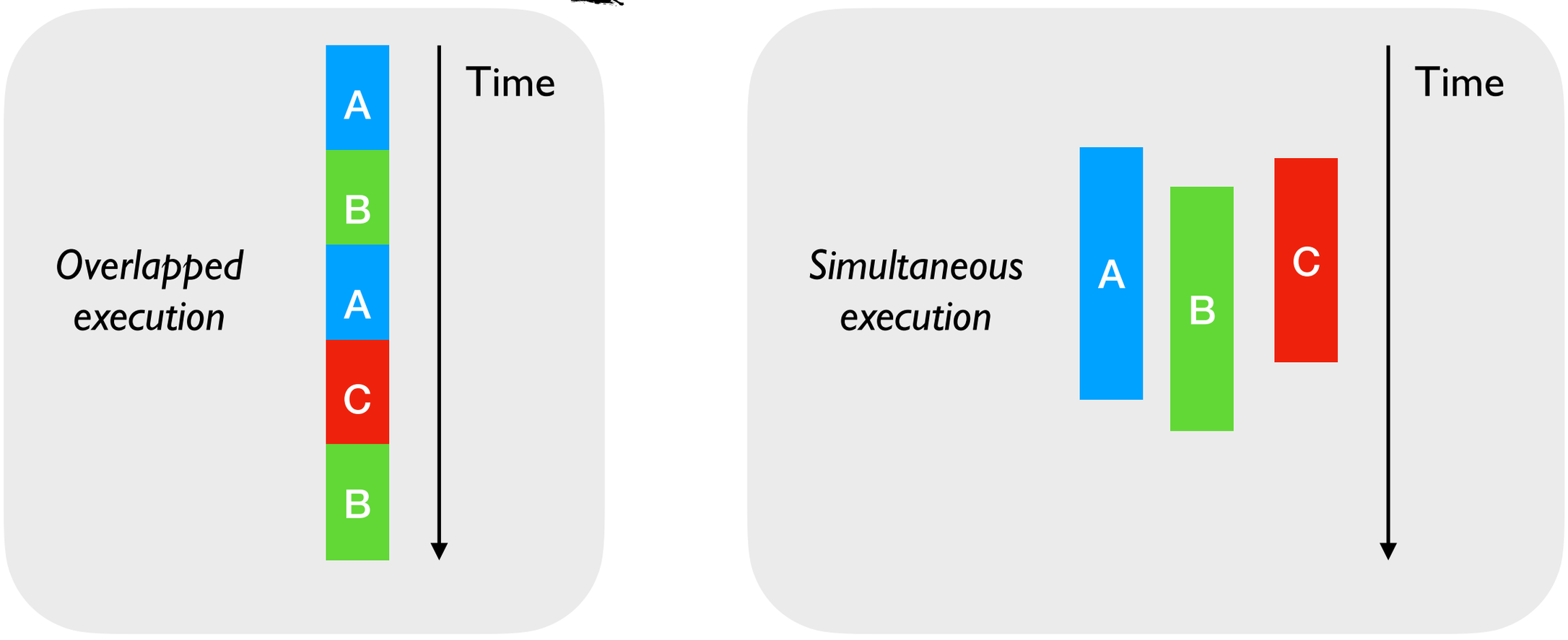
- Adds native support for *concurrency* and *parallelism* to OCaml



Effect Handlers

Multicore OCaml

- Adds native support for *concurrency* and *parallelism* to OCaml



Effect Handlers

Domains

Challenges

- Millions of lines of legacy code
  - ✦ Written without *concurrency* and *parallelism* in mind
  - ✦ Cost of refactoring sequential code itself is *prohibitive*

Challenges

- Millions of lines of legacy code
  - ✦ Written without *concurrency* and *parallelism* in mind
  - ✦ Cost of refactoring sequential code itself is *prohibitive*
- Low-latency and predictable performance
  - ✦ Great for applications that require ~10ms latency

Challenges

- Millions of lines of legacy code
  - ✦ Written without *concurrency* and *parallelism* in mind
  - ✦ Cost of refactoring sequential code itself is *prohibitive*
- Low-latency and predictable performance
  - ✦ Great for applications that require ~10ms latency
- Excellent compatibility with debugging and profiling tools
  - ✦ gdb, lldb, perf, libunwind, etc.

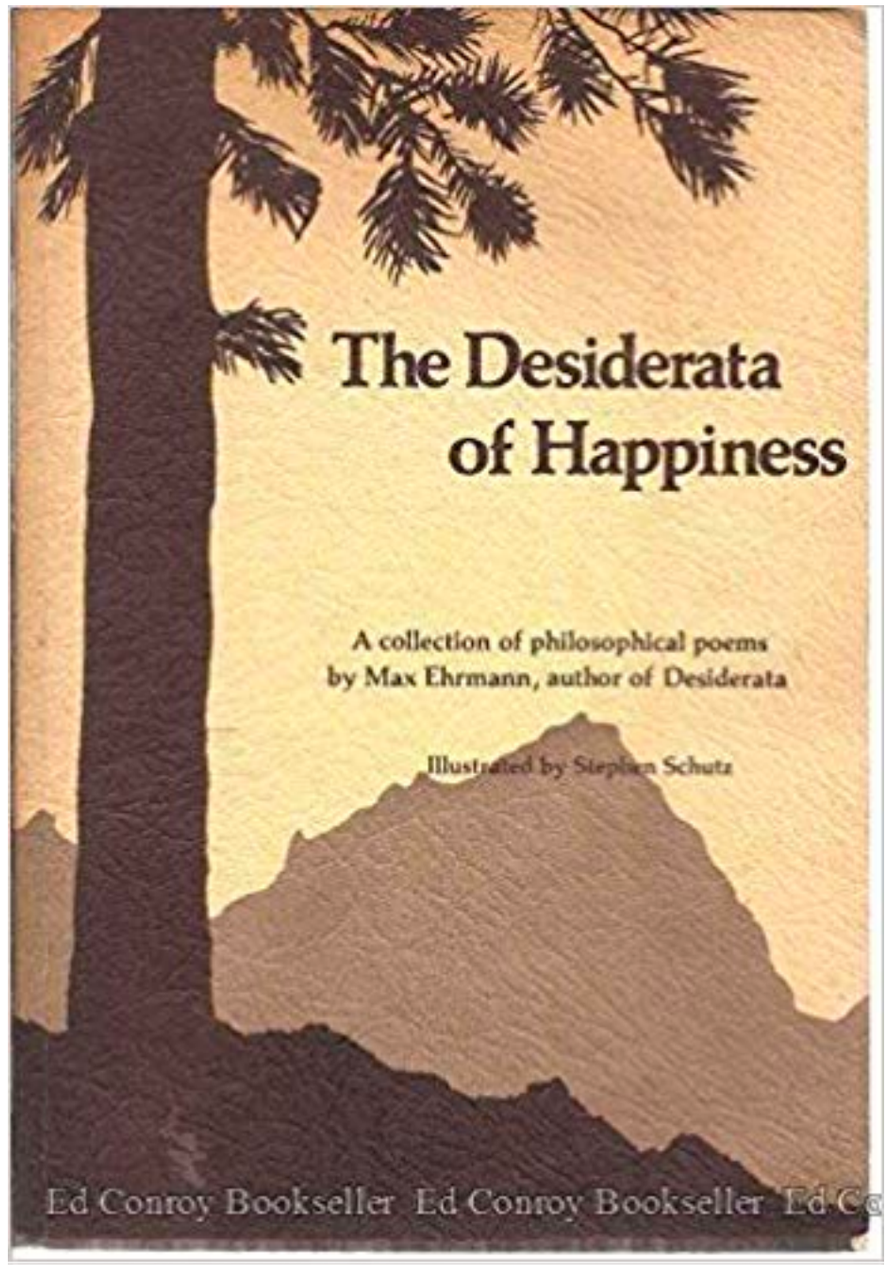
Challenges

- Millions of lines of legacy code
  - ✦ Written without *concurrency* and *parallelism* in mind
  - ✦ Cost of refactoring sequential code itself is *prohibitive*
- Low-latency and predictable performance
  - ✦ Great for applications that require ~10ms latency
- Excellent compatibility with debugging and profiling tools
  - ✦ gdb, lldb, perf, libunwind, etc.

Backwards compatibility before scalability

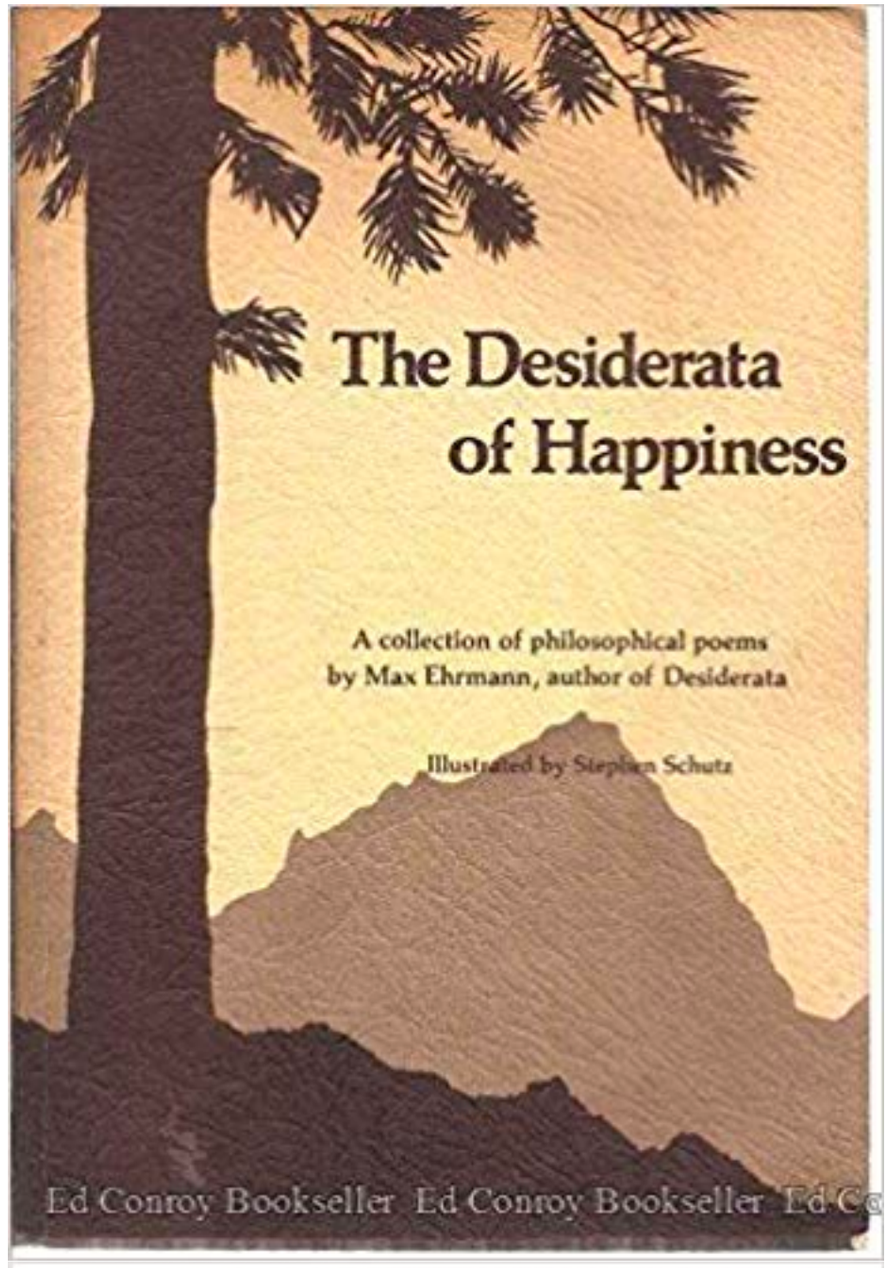
Desiderata

- Feature backwards compatibility
  - ✦ Do not break existing code



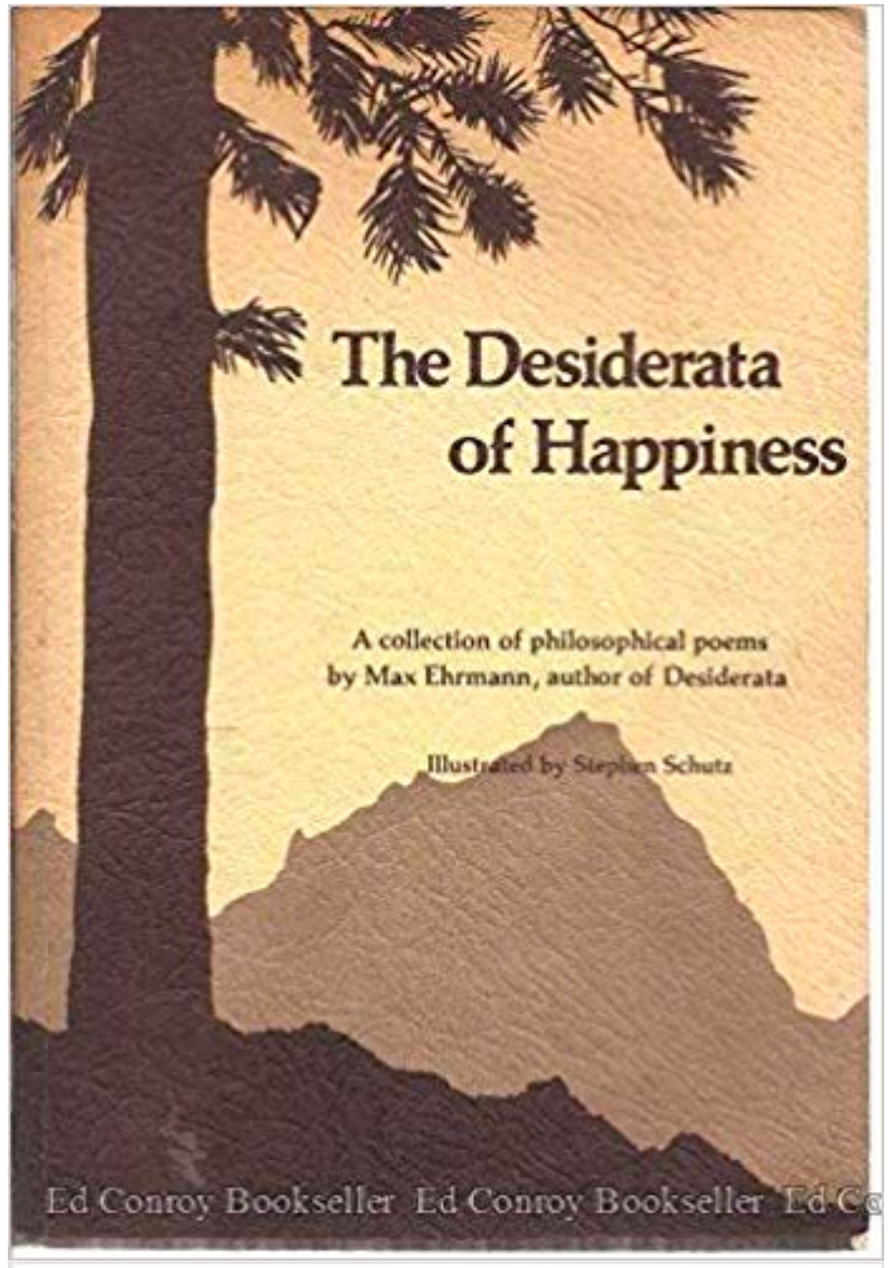
Desiderata

- Feature backwards compatibility
  - ✦ Do not break existing code
- Performance backwards compatibility
  - ✦ Existing programs run just as fast using just the same memory



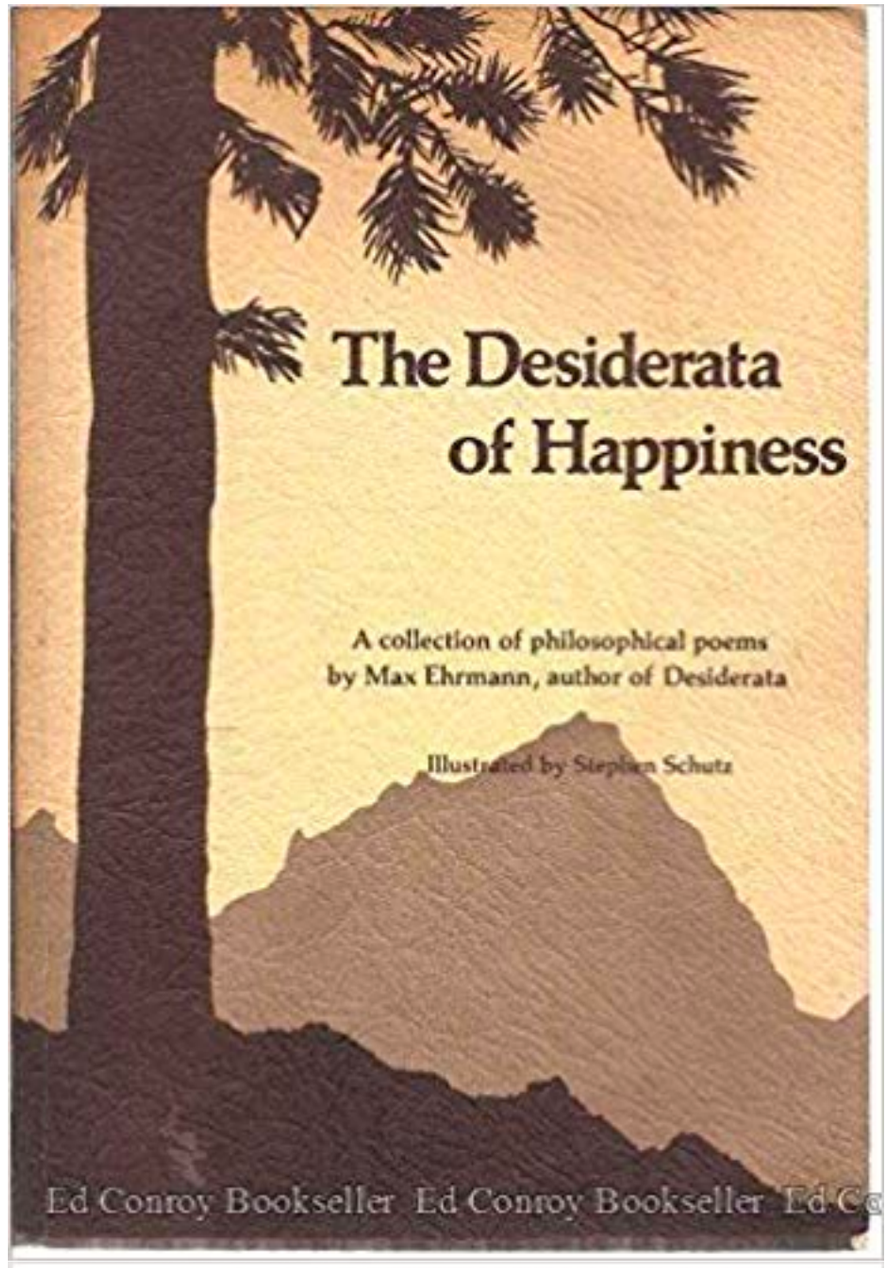
Desiderata

- Feature backwards compatibility
  - ✦ Do not break existing code
- Performance backwards compatibility
  - ✦ Existing programs run just as fast using just the same memory
- GC Latency before multicore scalability



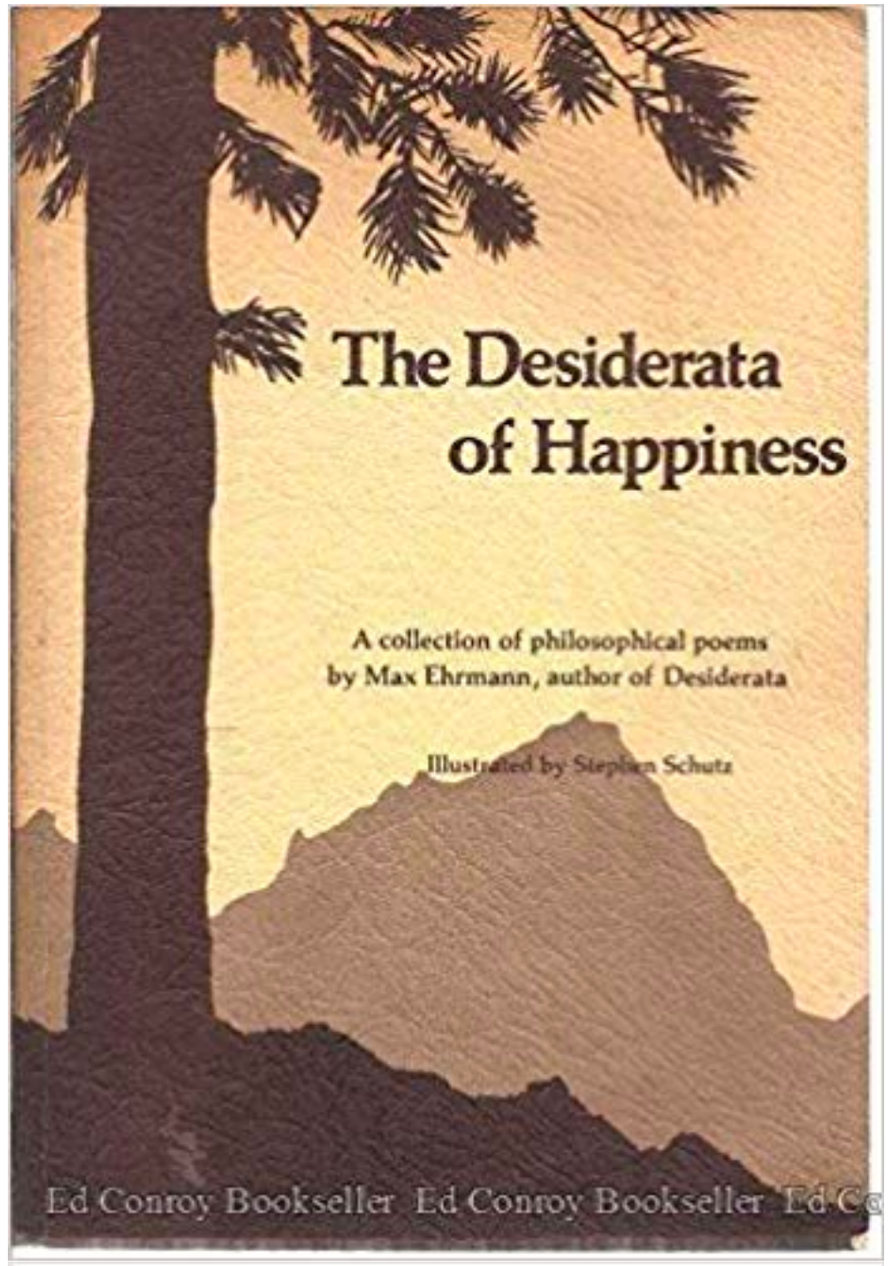
Desiderata

- Feature backwards compatibility
  - ✦ Do not break existing code
- Performance backwards compatibility
  - ✦ Existing programs run just as fast using just the same memory
- GC Latency before multicore scalability
- Compatibility with program inspection tools



Desiderata

- Feature backwards compatibility
  - ✦ Do not break existing code
- Performance backwards compatibility
  - ✦ Existing programs run just as fast using just the same memory
- GC Latency before multicore scalability
- Compatibility with program inspection tools
- Performant concurrent and parallel programming abstractions



Rest of the talk

- *Domains* for shared memory parallelism
- *Effect handlers* for concurrent programming

Domains for Parallelism

- A unit of parallelism

Domains for Parallelism

- A unit of parallelism
- **Heavyweight** — maps onto a OS thread
  - ✦ Recommended to have 1 domain per core

Domains for Parallelism

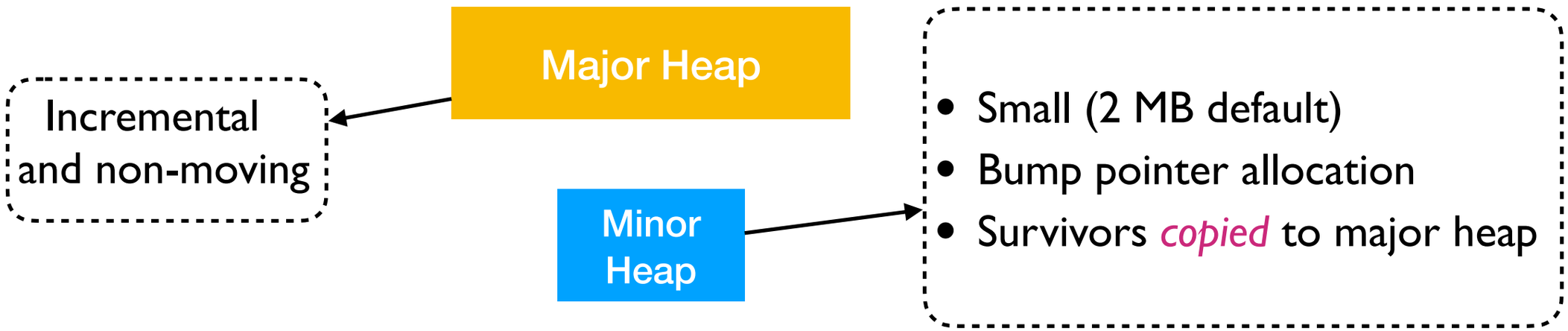
- A unit of parallelism
- **Heavyweight** — maps onto a OS thread
  - ✦ Recommended to have 1 domain per core
- Low-level domain API
  - ✦ Spawn & join, wait & notify
  - ✦ Domain-local storage
  - ✦ Atomic memory operations
    - ✤ Dolan et al, "Bounding Data Races in Space and Time", PLDI'18

Domains for Parallelism

- A unit of parallelism
- **Heavyweight** — maps onto a OS thread
  - ✦ Recommended to have 1 domain per core
- Low-level domain API
  - ✦ Spawn & join, wait & notify
  - ✦ Domain-local storage
  - ✦ Atomic memory operations
    - ✤ Dolan et al, "Bounding Data Races in Space and Time", PLDI'18
- No restrictions on sharing objects between domains
  - ✦ But how does it work?

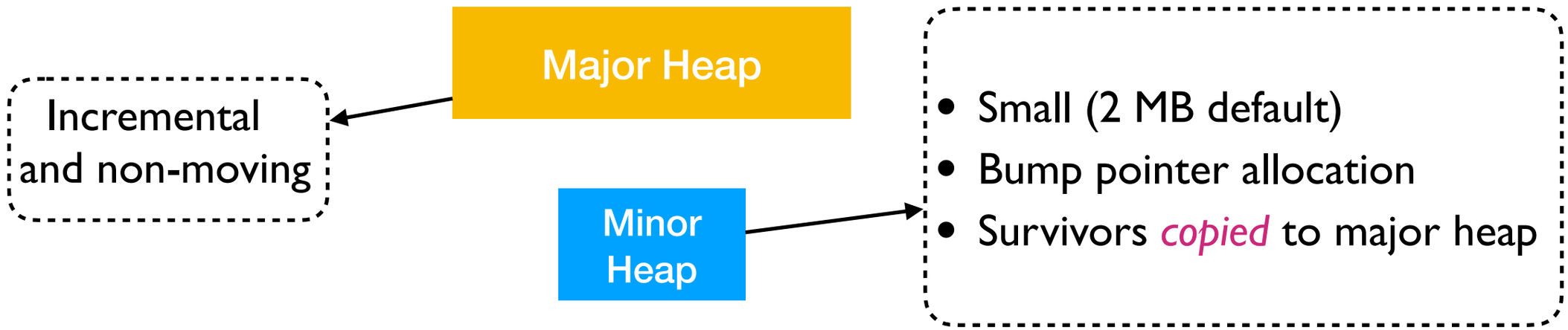
Stock OCaml GC

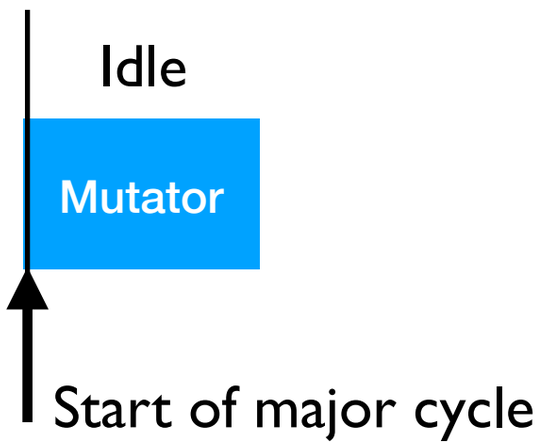
- A generational, non-moving, incremental, mark-and-sweep GC



Stock OCaml GC

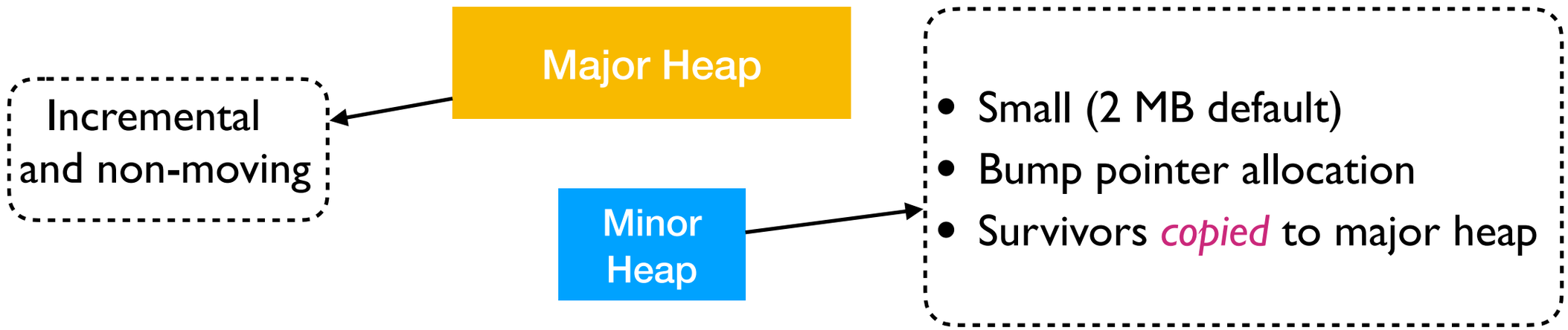
- A generational, non-moving, incremental, mark-and-sweep GC

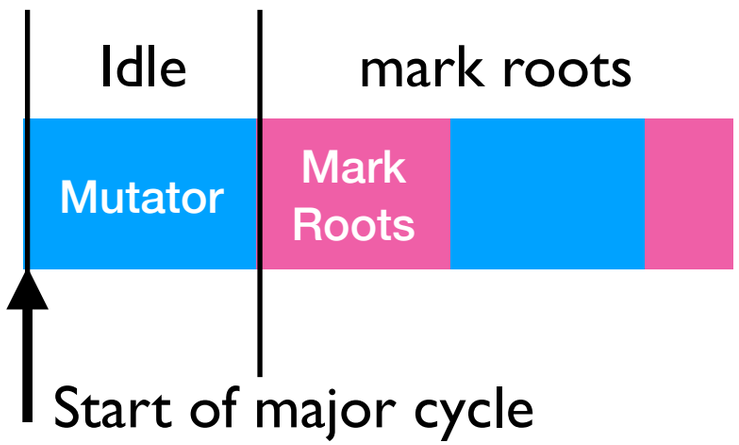




Stock OCaml GC

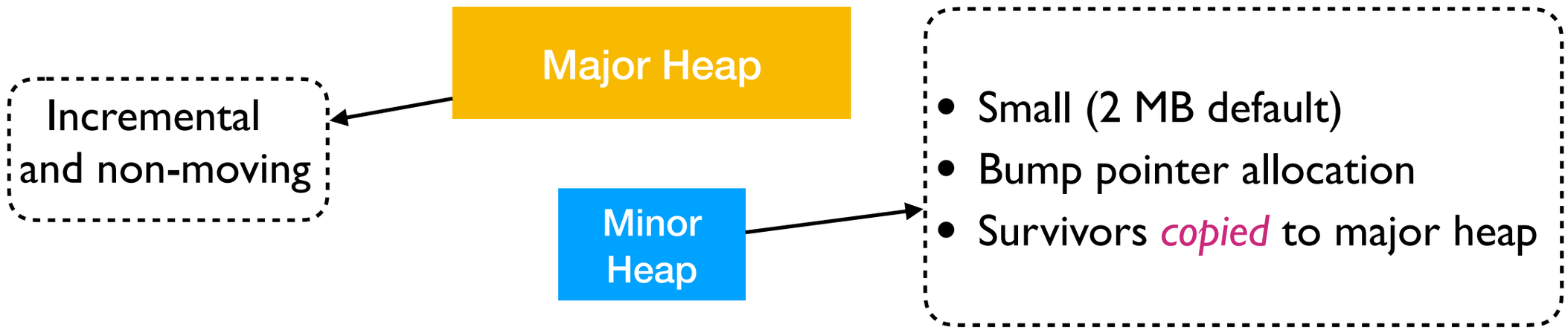
- A generational, non-moving, incremental, mark-and-sweep GC

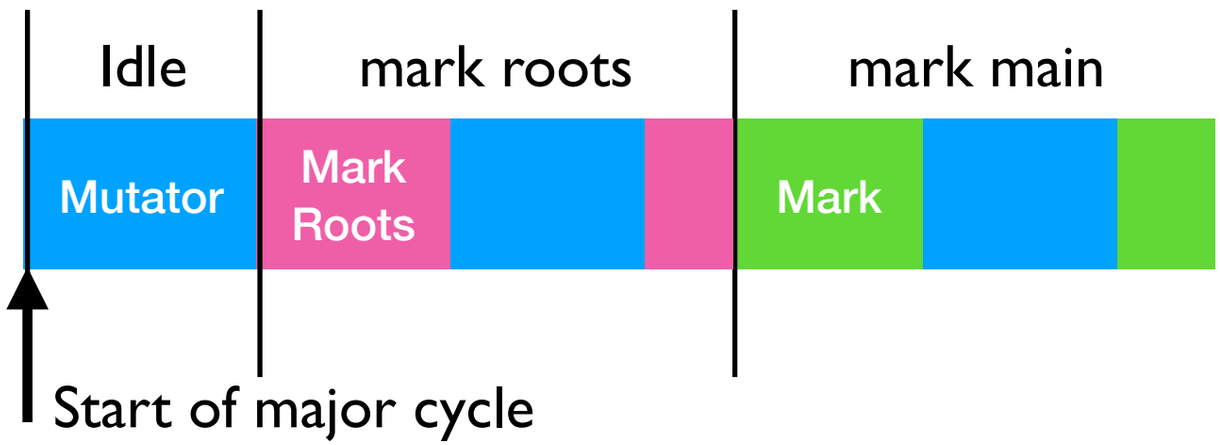




Stock OCaml GC

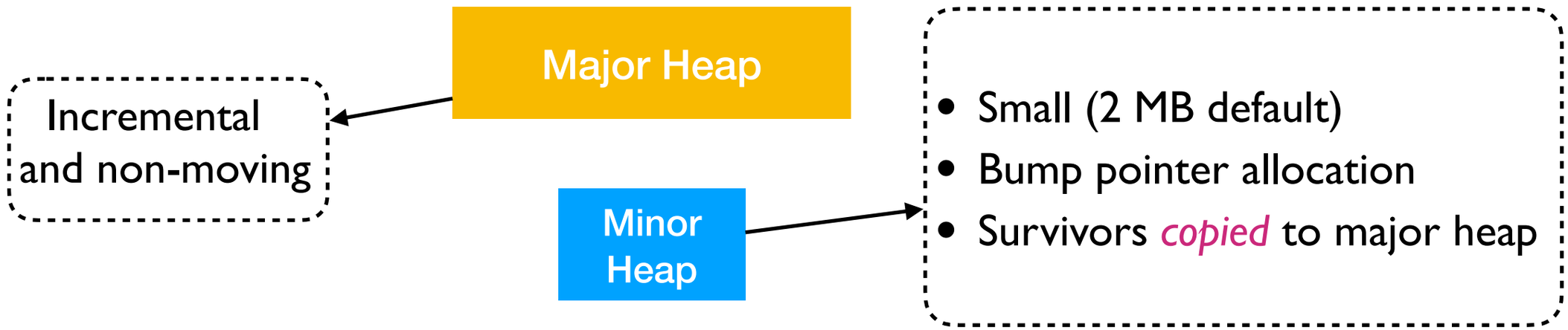
- A generational, non-moving, incremental, mark-and-sweep GC

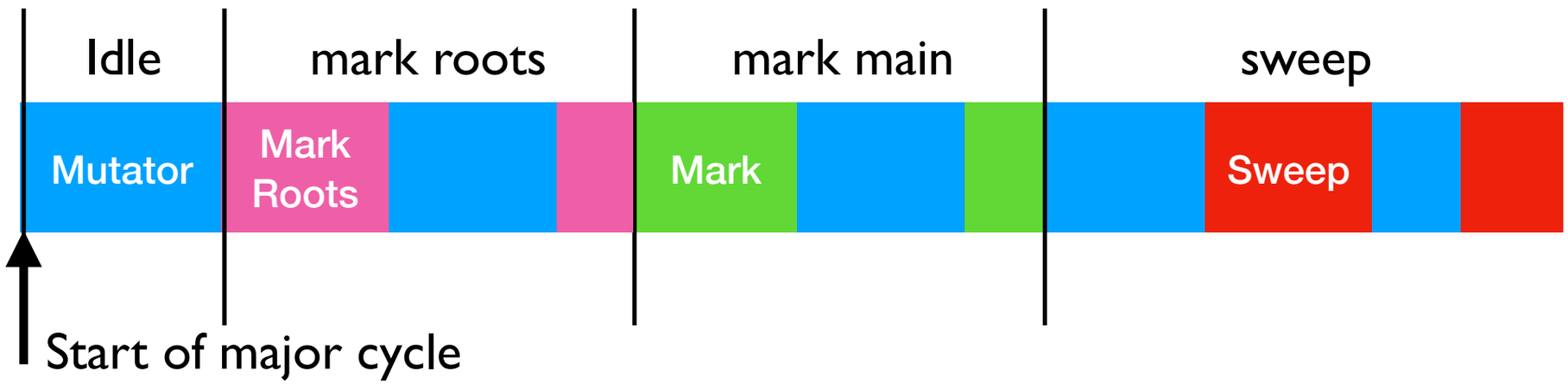




Stock OCaml GC

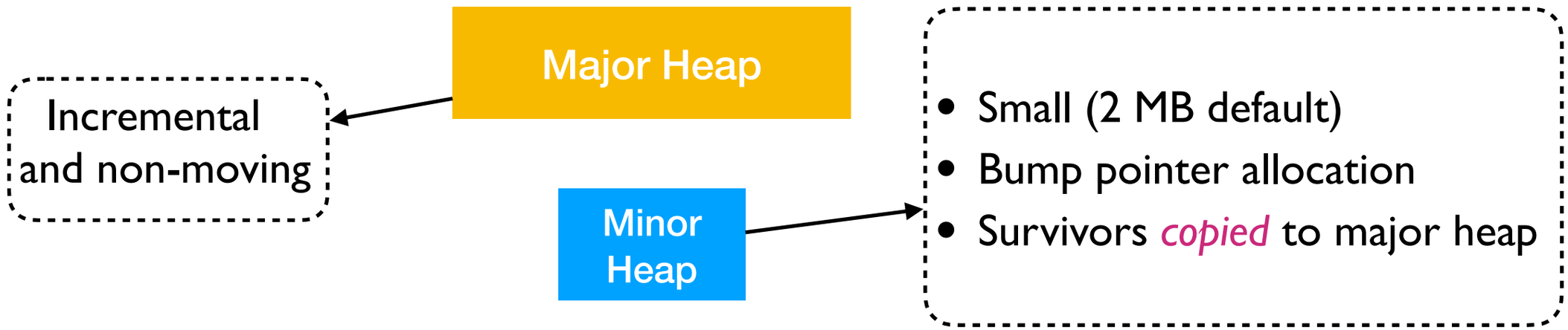
- A generational, non-moving, incremental, mark-and-sweep GC

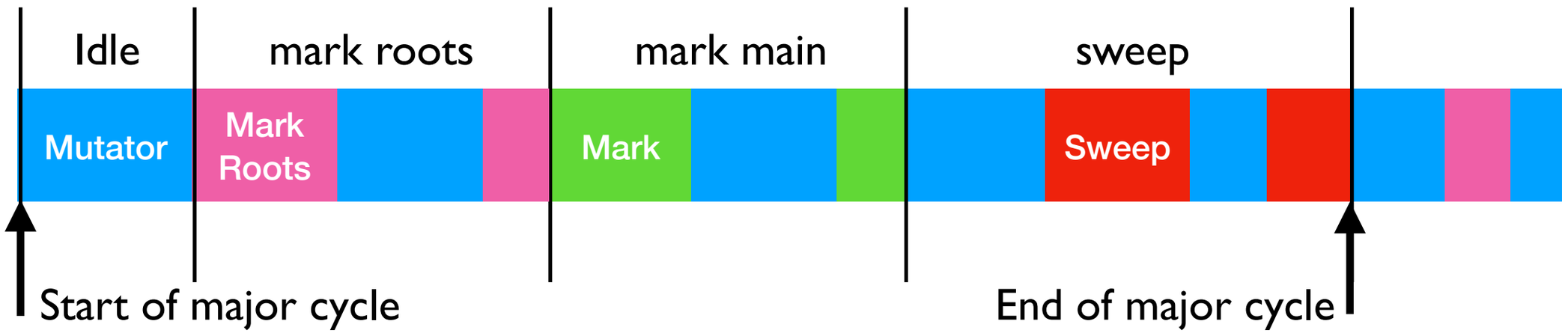




Stock OCaml GC

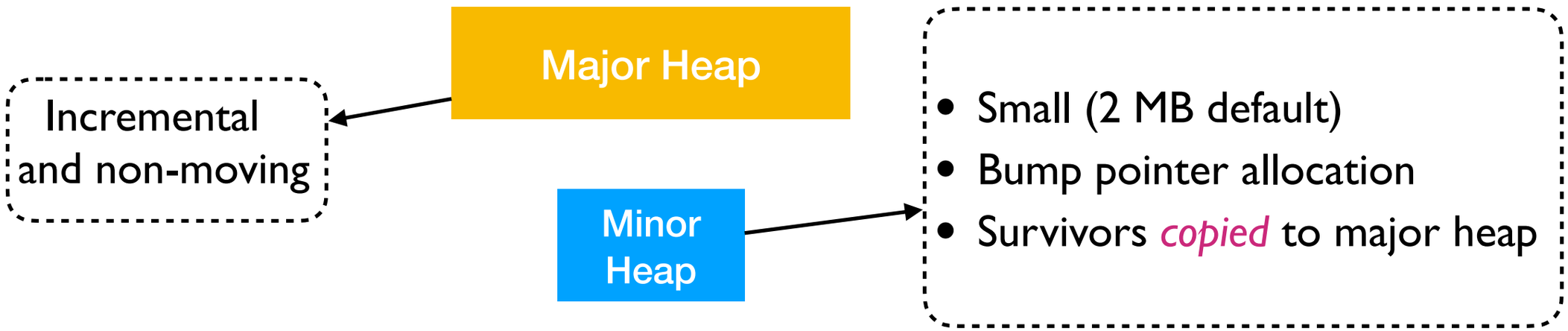
- A generational, non-moving, incremental, mark-and-sweep GC

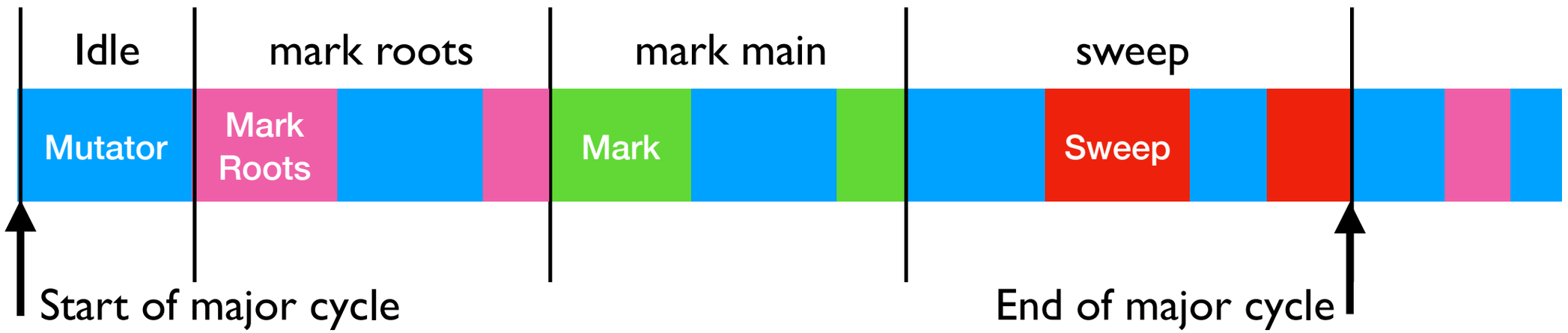




Stock OCaml GC

- A generational, non-moving, incremental, mark-and-sweep GC

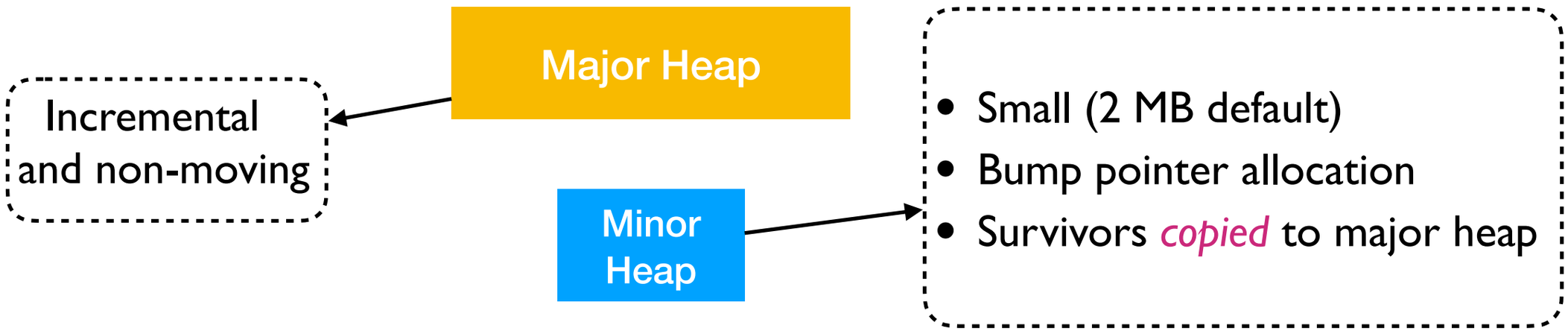


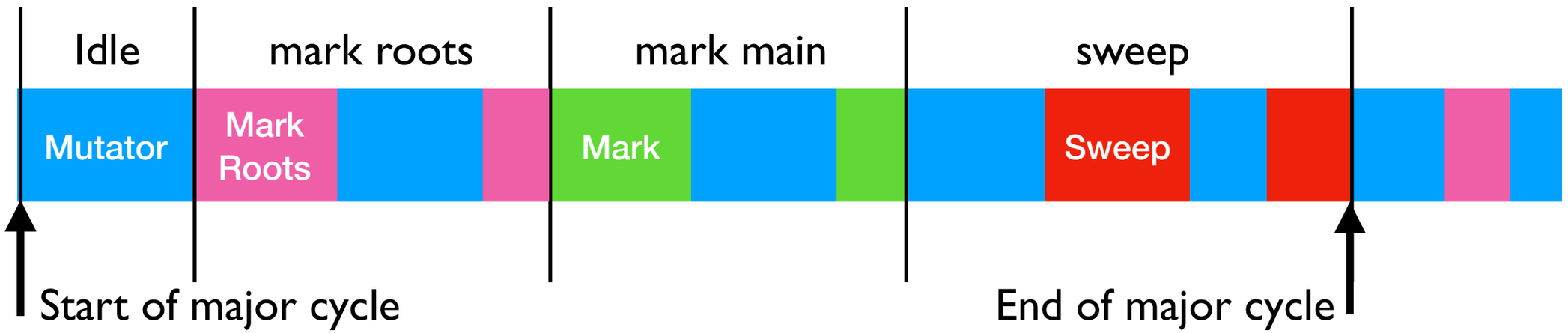


- Fast allocations

Stock OCaml GC

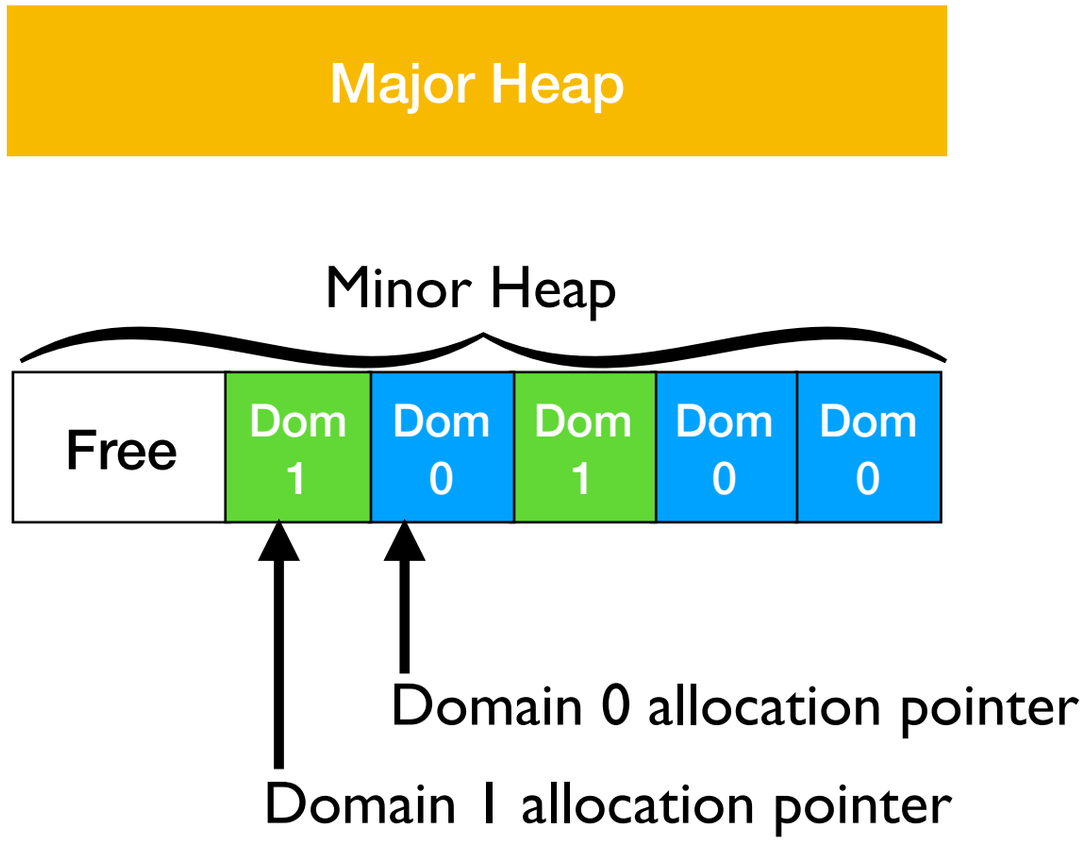
- A generational, non-moving, incremental, mark-and-sweep GC



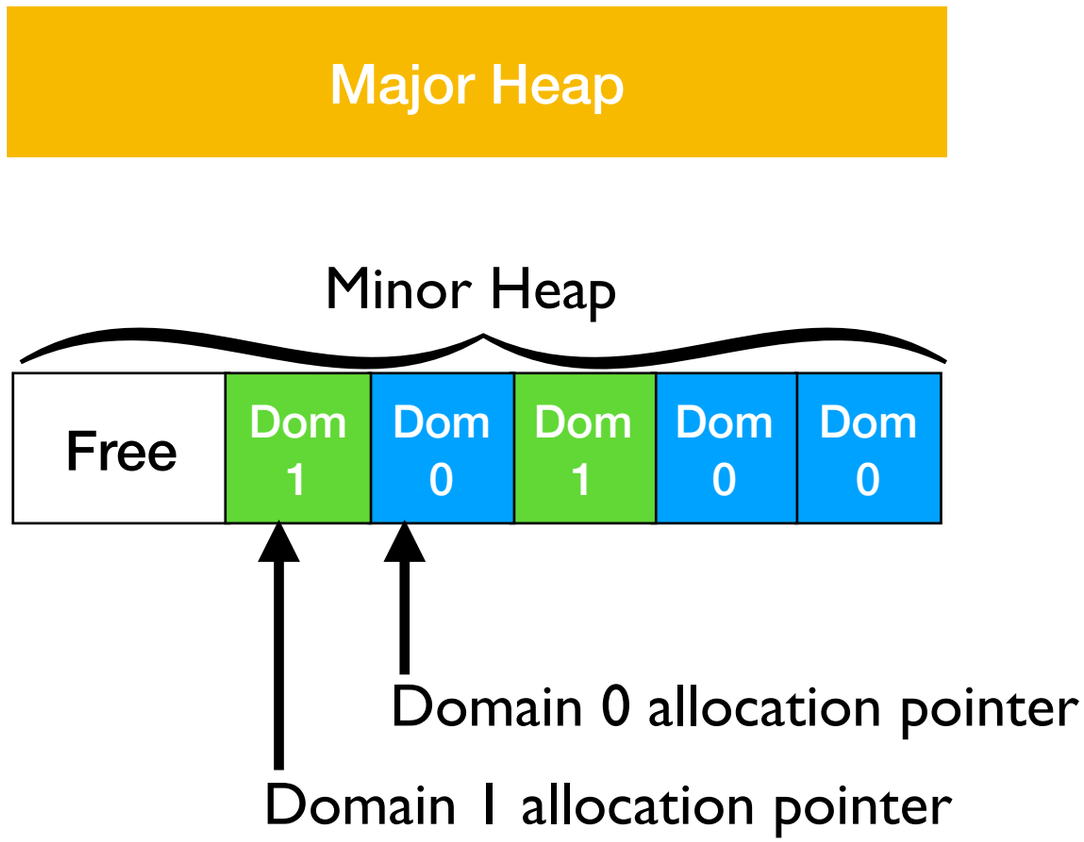


- Fast allocations
- Max GC latency **< 10 ms**, 99th percentile latency **< 1 ms**

Multicore OCaml GC

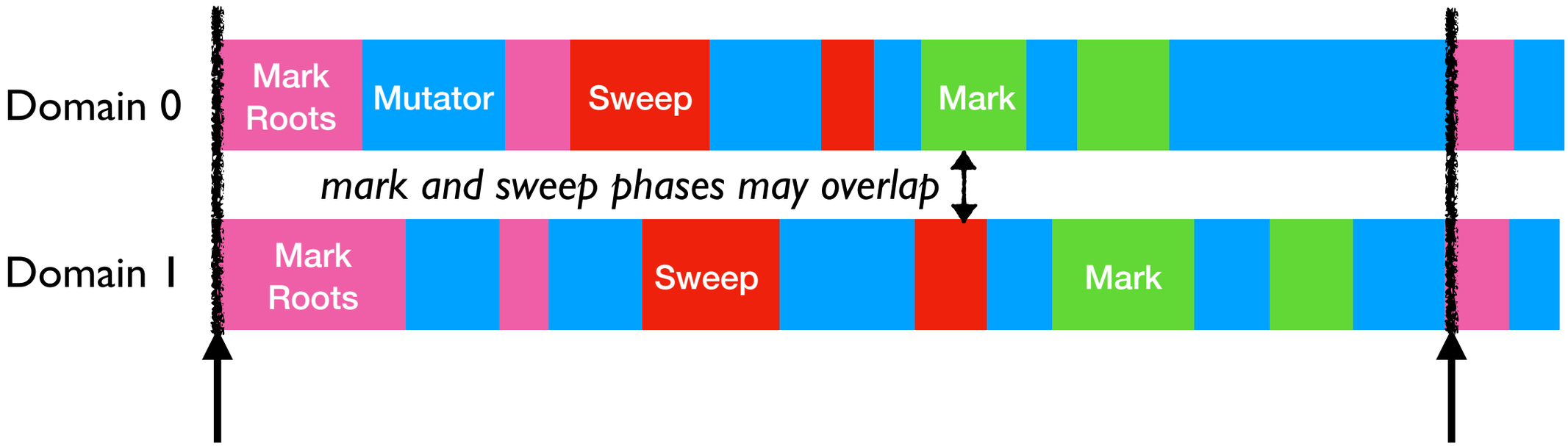


Multicore OCaml GC



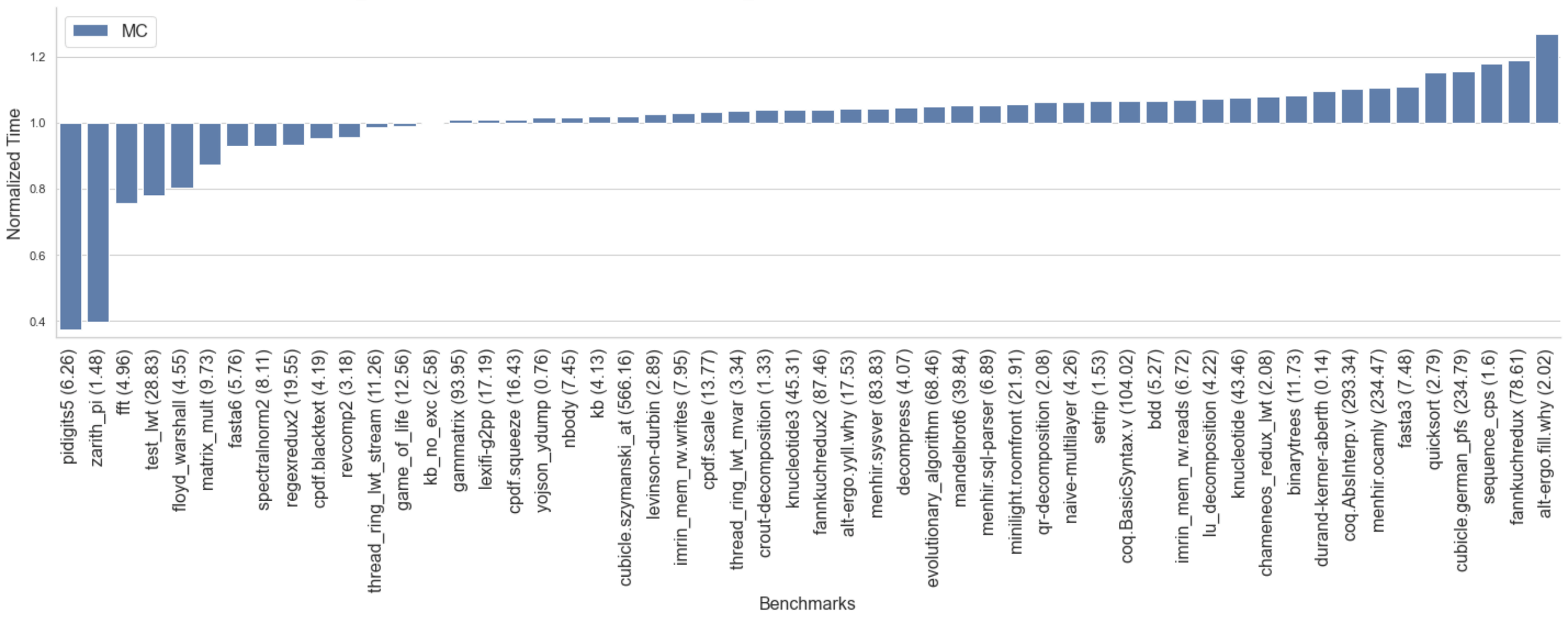
- *Stop-the-world parallel minor collection* for minor heap
  - ✦ 2 global barriers / minor gc
  - ✦ On 24 cores, ~10 ms pauses

Multicore OCaml GC

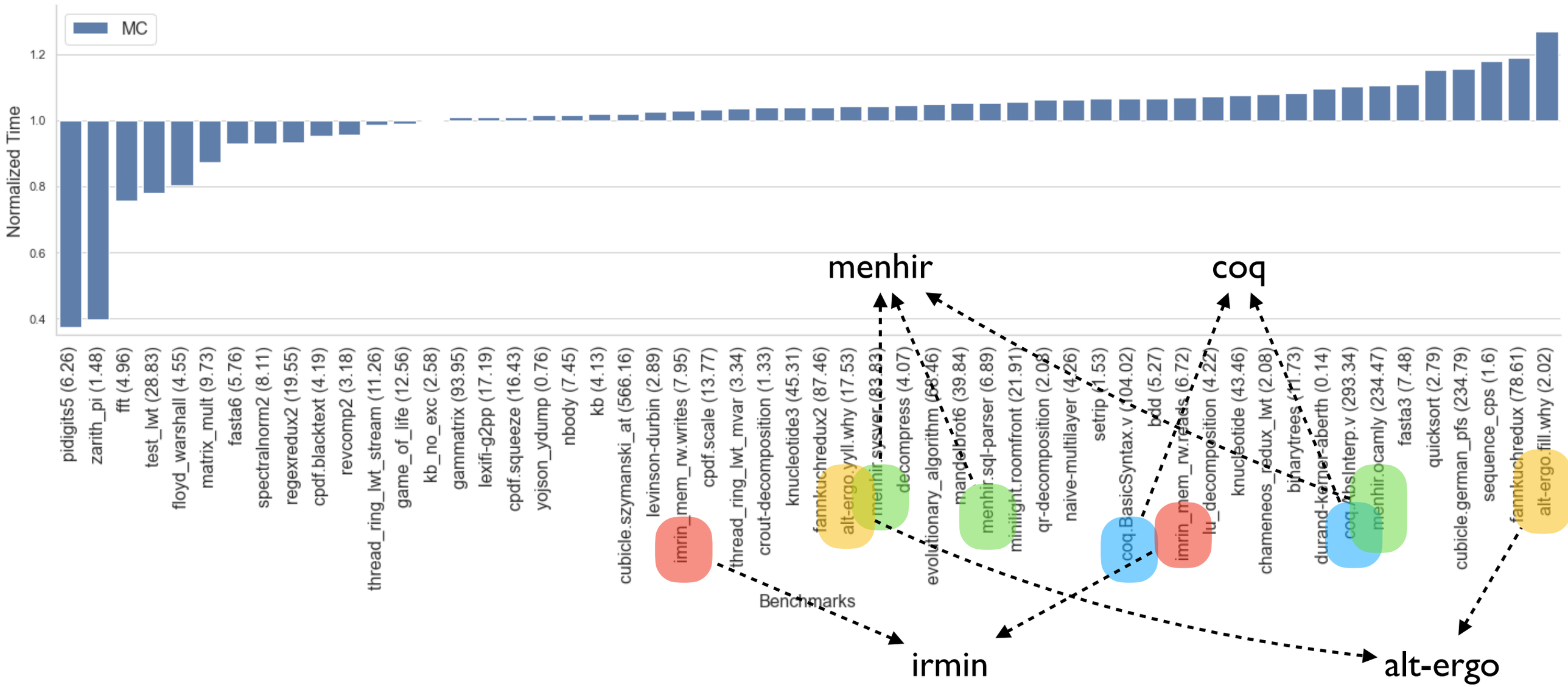


- *Mostly-concurrent mark-and-sweep* for major collection
  - ✦ All the marking and sweeping work done *without synchronization*
  - ✦ 3 barriers per cycle (worst case) to agree end of GC phases
    - ✤ 2 barriers for the two kinds of finalisers in OCaml
  - ✦ ~5 ms pauses on 24 cores

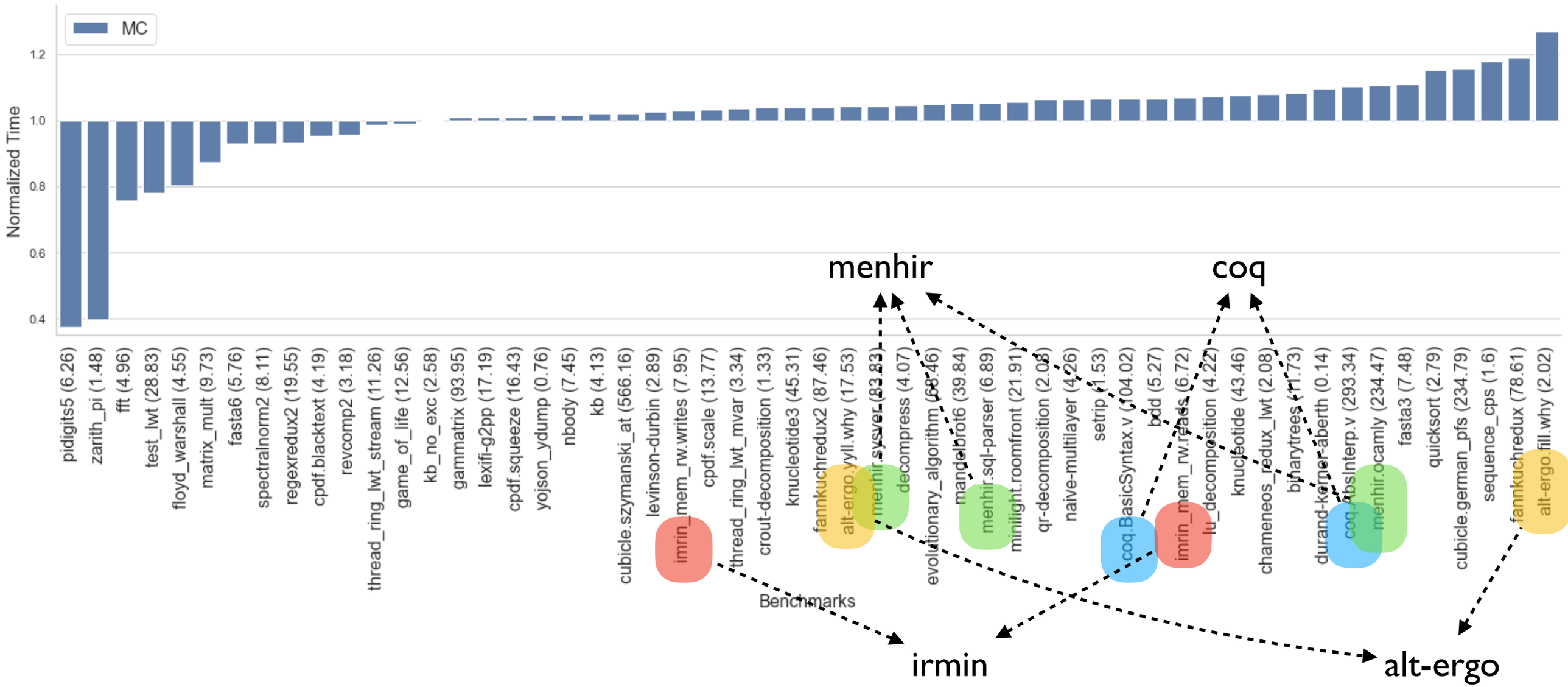
Sequential performance



Sequential performance



Sequential performance



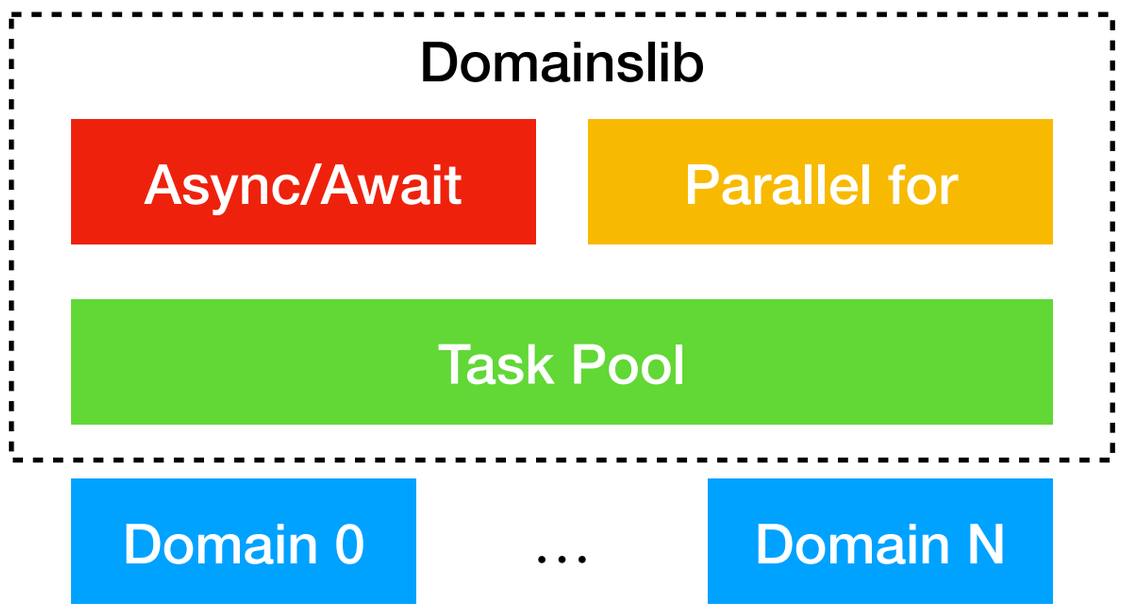
- ~1% faster than stock (geomean of normalised running times)
  - ✦ Difference under measurement noise mostly
  - ✦ Outliers due to difference in allocators

Domainslib for parallel programming

- Domain API exposed by the compiler is too low-level

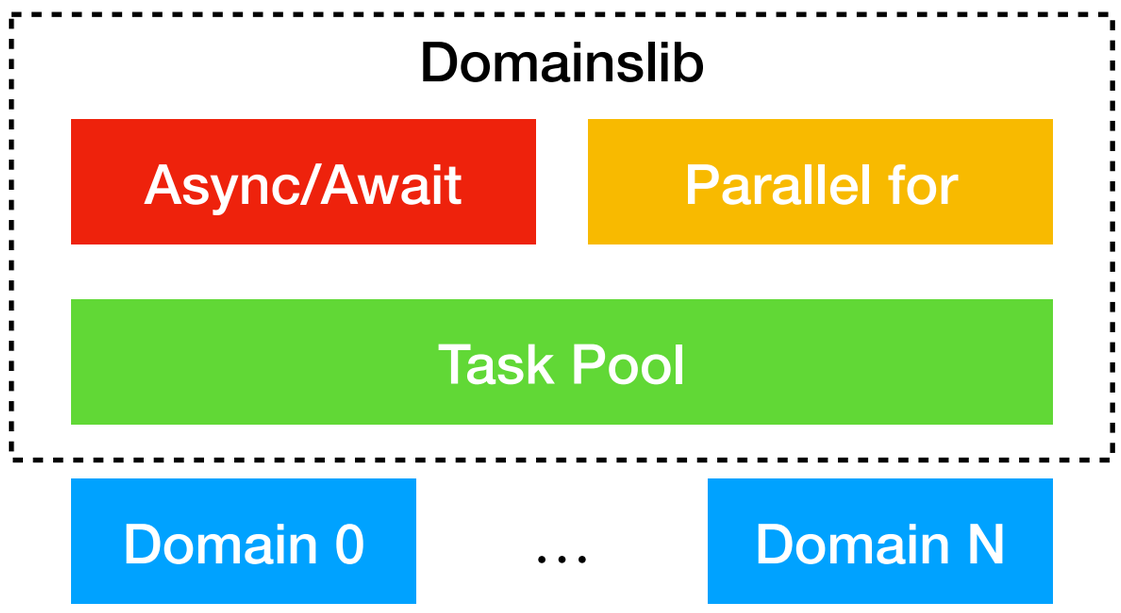
Domainslib for parallel programming

- Domain API exposed by the compiler is too low-level
- Domainslib - *https://github.com/ocaml-multicore/domainslib*



Domainslib for parallel programming

- Domain API exposed by the compiler is too low-level
- Domainslib - *https://github.com/ocaml-multicore/domainslib*



Let's look at examples!

Recursive Fibonacci - Sequential

```
let rec fib n =
  if n < 2 then 1
  else fib (n-1) + fib (n-2)
```

Recursive Fibonacci - Parallel

```
module T = Domainslib.Task
```

```
let fib n =
  let pool = T.setup_pool ~num_domains:(num_domains - 1) in
  let res = fib_par pool n in
  T.teardown_pool pool;
  res
```

Recursive Fibonacci - Parallel

```
module T = Domainslib.Task



let rec fib_par pool n =
  if n <= 40 then fib_seq n
  else
    let a = T.async pool (fun _ -> fib_par pool (n-1)) in
    let b = T.async pool (fun _ -> fib_par pool (n-2)) in
    T.await pool a + T.await pool b

let fib n =
  let pool = T.setup_pool ~num_domains:(num_domains - 1) in
  let res = fib_par pool n in
  T.teardown_pool pool;
  res
```

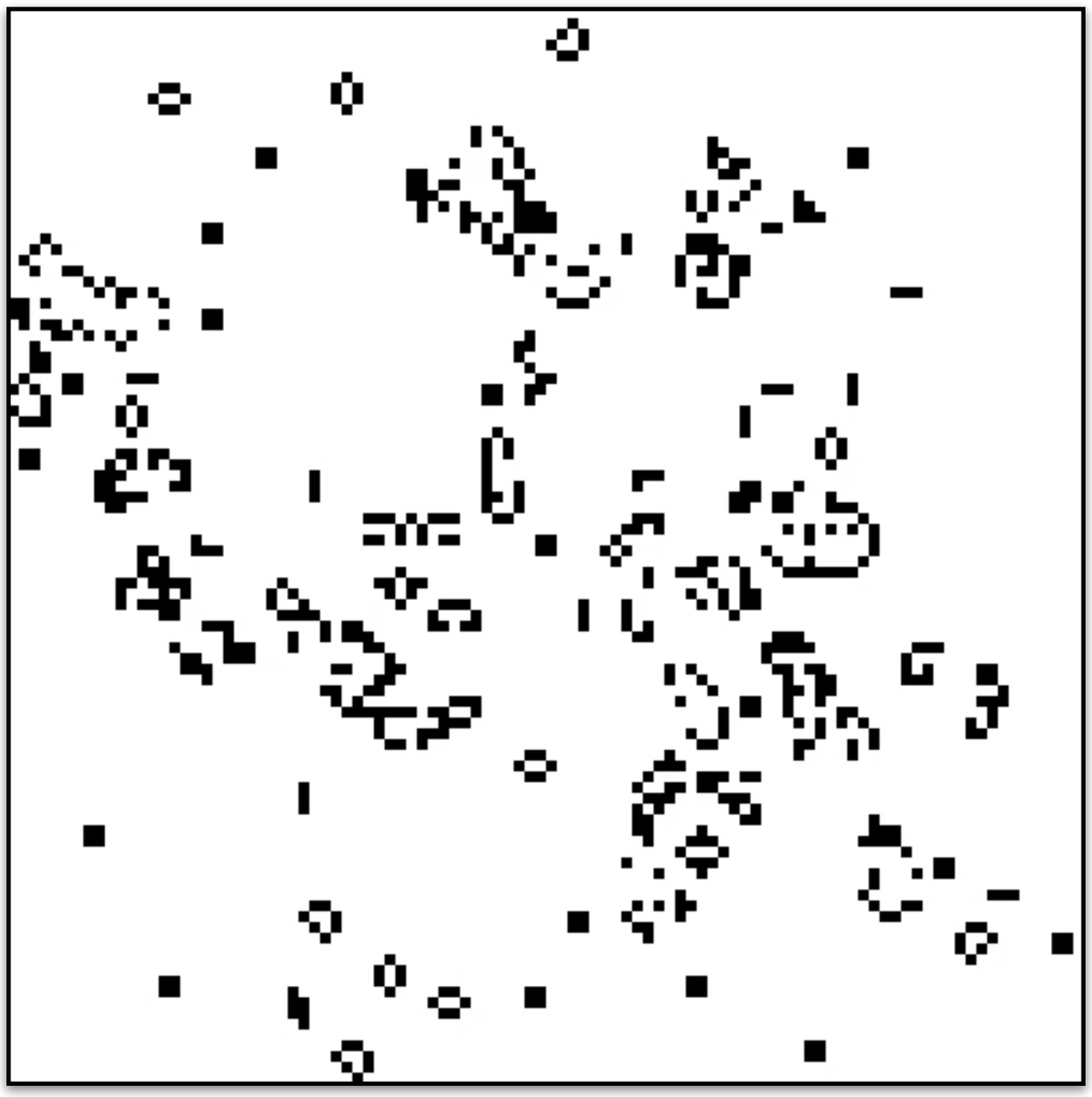
Recursive Fibonacci - Parallel

```
module T = Domainslib.Task

let rec fib_seq n =
  if n < 2 then 1
  else fib_seq (n-1) + fib_seq (n-2)

let rec fib_par pool n =
  if n <= 40 then fib_seq n
  else
    let a = T.async pool (fun _ -> fib_par pool (n-1)) in
    let b = T.async pool (fun _ -> fib_par pool (n-2)) in
    T.await pool a + T.await pool b

let fib n =
  let pool = T.setup_pool ~num_domains:(num_domains - 1) in
  let res = fib_par pool n in
  T.teardown_pool pool;
  res
```

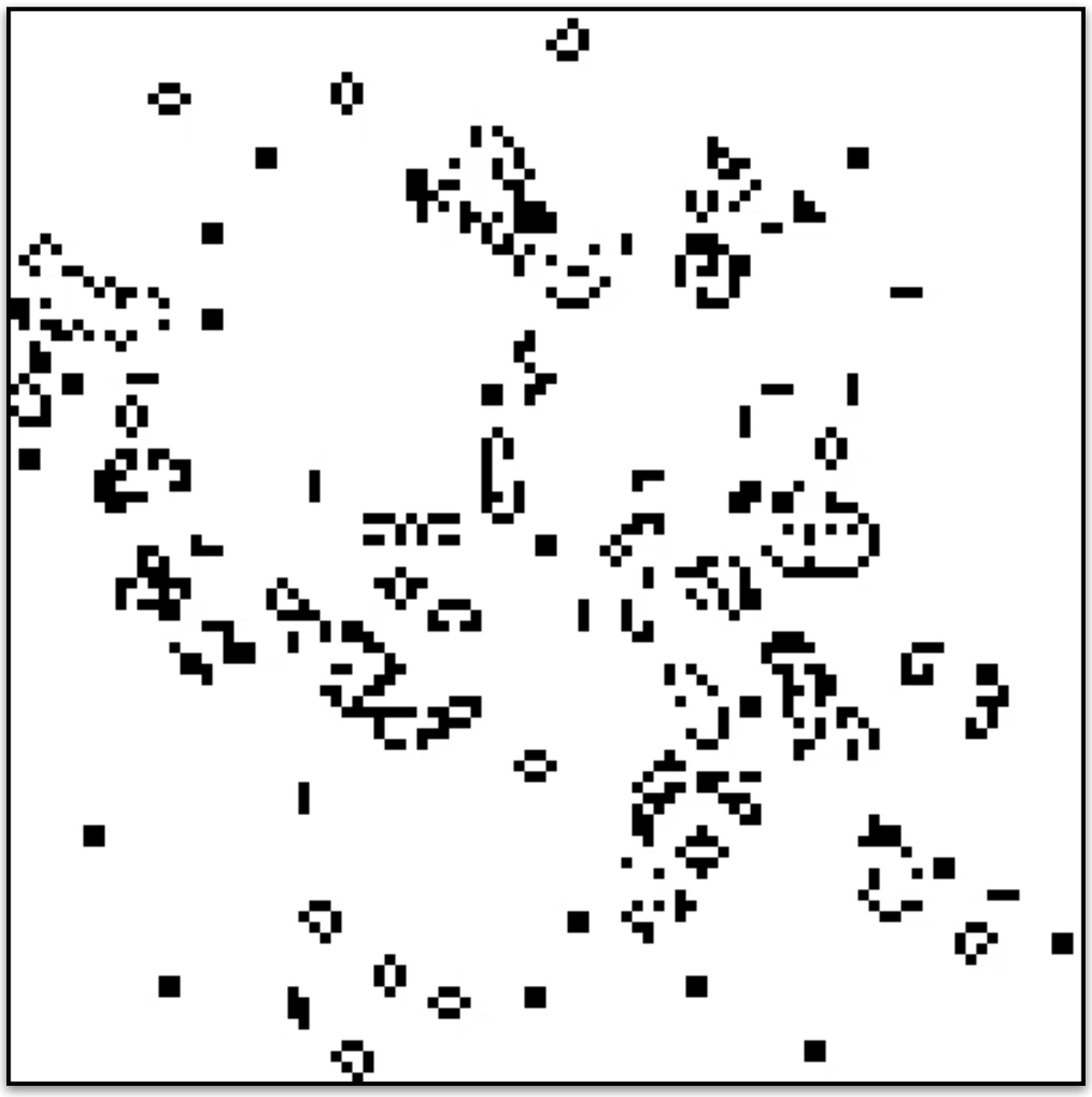
Performance: fib(48)

Cores	Time (Seconds)	Vs Serial	Vs Self
1	37.787	0.98	1
2	19.034	1.94	1.99
4	9.723	3.8	3.89
8	5.023	7.36	7.52
16	2.914	12.68	12.97
24	2.201	**16.79**	**17.17**

Conway's Game of Life



Conway's Game of Life



Conway's Game of Life

```
let next () =
  ...
  for x = 0 to board_size - 1 do
    for y = 0 to board_size - 1 do
      next_board.(x).(y) <- next_cell cur_board x y
    done
  done;
  ...
```

Conway's Game of Life

```
let next () =
  ...
  for x = 0 to board_size - 1 do
    for y = 0 to board_size - 1 do
      next_board.(x).(y) <- next_cell cur_board x y
    done
  done;
  ...
```

```
let next () =
  ...
  T.parallel_for pool ~start:0 ~finish:(board_size - 1)
    ~body:(fun x ->
       for y = 0 to board_size - 1 do
         next_board.(x).(y) <- next_cell cur_board x y
       done);
  ...
```

Performance: Game of Life

Board size = 1024, Iterations = 512

Cores	Time (Seconds)	Vs Serial	Vs Self
1	24.326	1	1
2	12.290	1.980	1.98
4	6.260	3.890	3.89
8	3.238	7.51	7.51
16	1.726	14.09	14.09
24	1.212	**20.07**	**20.07**

Parallelism is not Concurrency

Parallelism is a performance hack

whereas

concurrency is a program structuring mechanism

Parallelism is not Concurrency

Parallelism is a performance hack

whereas

concurrency is a program structuring mechanism

- Lwt and Async - concurrent programming libraries in OCaml
  - ✦ Callback-oriented programming with nicer syntax

Parallelism is not Concurrency

Parallelism is a performance hack

whereas

concurrency is a program structuring mechanism

- Lwt and Async - concurrent programming libraries in OCaml
  - ✦ Callback-oriented programming with nicer syntax
- Suffers many pitfalls of callback-oriented programming
  - ✦ No backtraces, exceptions can't be used, monadic syntax

Parallelism is not Concurrency

Parallelism is a performance hack

whereas

concurrency is a program structuring mechanism

- Lwt and Async - concurrent programming libraries in OCaml
  - ✦ Callback-oriented programming with nicer syntax
- Suffers many pitfalls of callback-oriented programming
  - ✦ No backtraces, exceptions can't be used, monadic syntax
- Go (goroutines) and GHC Haskell (threads) have better abstractions — lightweight threads

Parallelism is not Concurrency

Parallelism is a performance hack

whereas

concurrency is a program structuring mechanism

- Lwt and Async - concurrent programming libraries in OCaml
  - ✦ Callback-oriented programming with nicer syntax
- Suffers many pitfalls of callback-oriented programming
  - ✦ No backtraces, exceptions can't be used, monadic syntax
- Go (goroutines) and GHC Haskell (threads) have better abstractions — lightweight threads

Should we add lightweight threads to OCaml?

Effect Handlers

- A mechanism for programming with *user-defined effects*

Effect Handlers

- A mechanism for programming with *user-defined effects*
- Modular basis of non-local control-flow mechanisms
  - ✦ Exceptions, generators, lightweight threads, promises, asynchronous IO, coroutines

Effect Handlers

- A mechanism for programming with *user-defined effects*
- Modular basis of non-local control-flow mechanisms
  - ✦ Exceptions, generators, lightweight threads, promises, asynchronous IO, coroutines
- Effect *declaration* separate from *interpretation* (c.f. exceptions)

Effect Handlers

- A mechanism for programming with *user-defined effects*
- Modular basis of non-local control-flow mechanisms
  - ✦ Exceptions, generators, lightweight threads, promises, asynchronous IO, coroutines
- Effect *declaration* separate from *interpretation* (c.f. exceptions)

```
effect E : string

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 Handlers

- A mechanism for programming with *user-defined effects*
- Modular basis of non-local control-flow mechanisms
  - ✦ Exceptions, generators, lightweight threads, promises, asynchronous IO, coroutines
- Effect *declaration* separate from *interpretation* (c.f. exceptions)

effect declaration

```
effect E : string

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 Handlers

- A mechanism for programming with *user-defined effects*
- Modular basis of non-local control-flow mechanisms
  - ✦ Exceptions, generators, lightweight threads, promises, asynchronous IO, coroutines
- Effect *declaration* separate from *interpretation* (c.f. exceptions)

effect declaration

```
effect E : string

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 "
```

computation

Effect Handlers

- A mechanism for programming with *user-defined effects*
- Modular basis of non-local control-flow mechanisms
  - ✦ Exceptions, generators, lightweight threads, promises, asynchronous IO, coroutines
- Effect *declaration* separate from *interpretation* (c.f. exceptions)

```
effect E : string

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

computation

handler

Effect Handlers

- A mechanism for programming with *user-defined effects*
- Modular basis of non-local control-flow mechanisms
  - ✦ Exceptions, generators, lightweight threads, promises, asynchronous IO, coroutines
- Effect *declaration* separate from *interpretation* (c.f. exceptions)

```
effect E : string

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

handler

Effect Handlers

- A mechanism for programming with *user-defined effects*
- Modular basis of non-local control-flow mechanisms
  - ✦ Exceptions, generators, lightweight threads, promises, asynchronous IO, coroutines
- Effect *declaration* separate from *interpretation* (c.f. exceptions)

```
effect E : string

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

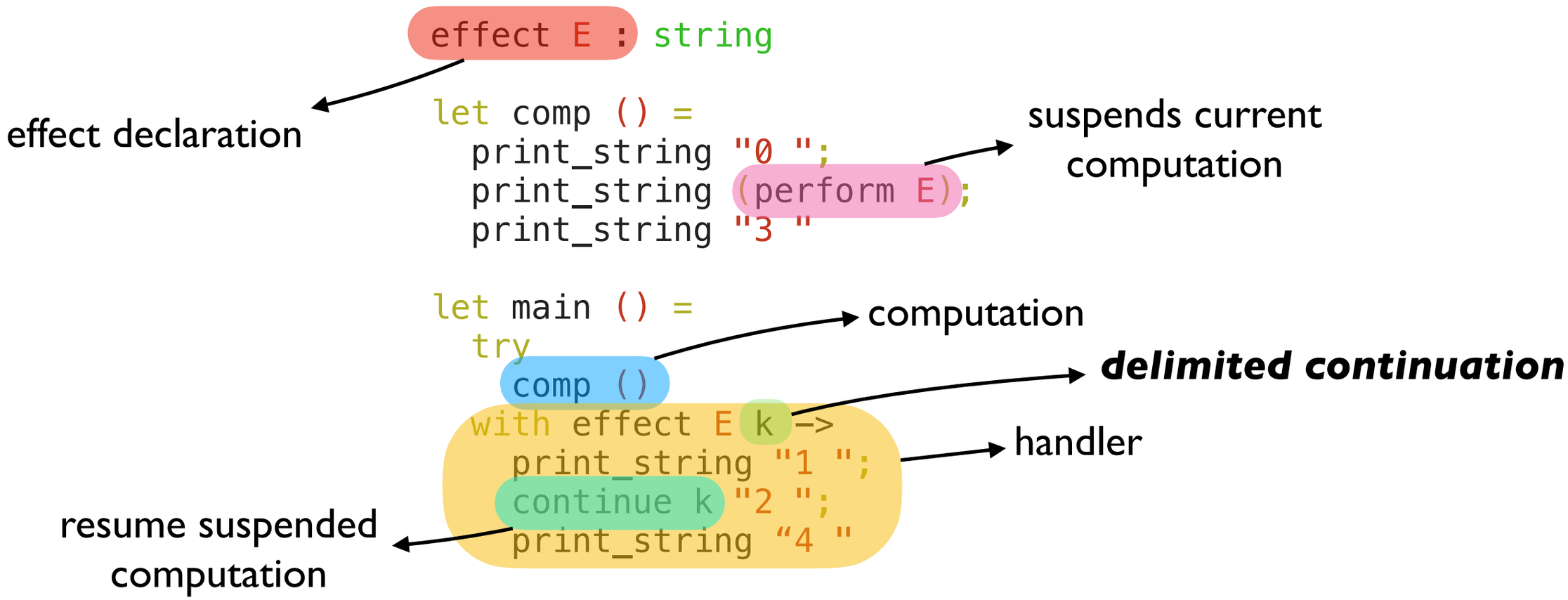
computation

delimited continuation

handler

Effect Handlers

- A mechanism for programming with *user-defined effects*
- Modular basis of non-local control-flow mechanisms
  - ✦ Exceptions, generators, lightweight threads, promises, asynchronous IO, coroutines
- Effect *declaration* separate from *interpretation* (c.f. exceptions)



Stepping through the example

```
effect E : string

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 →

Stepping through the example

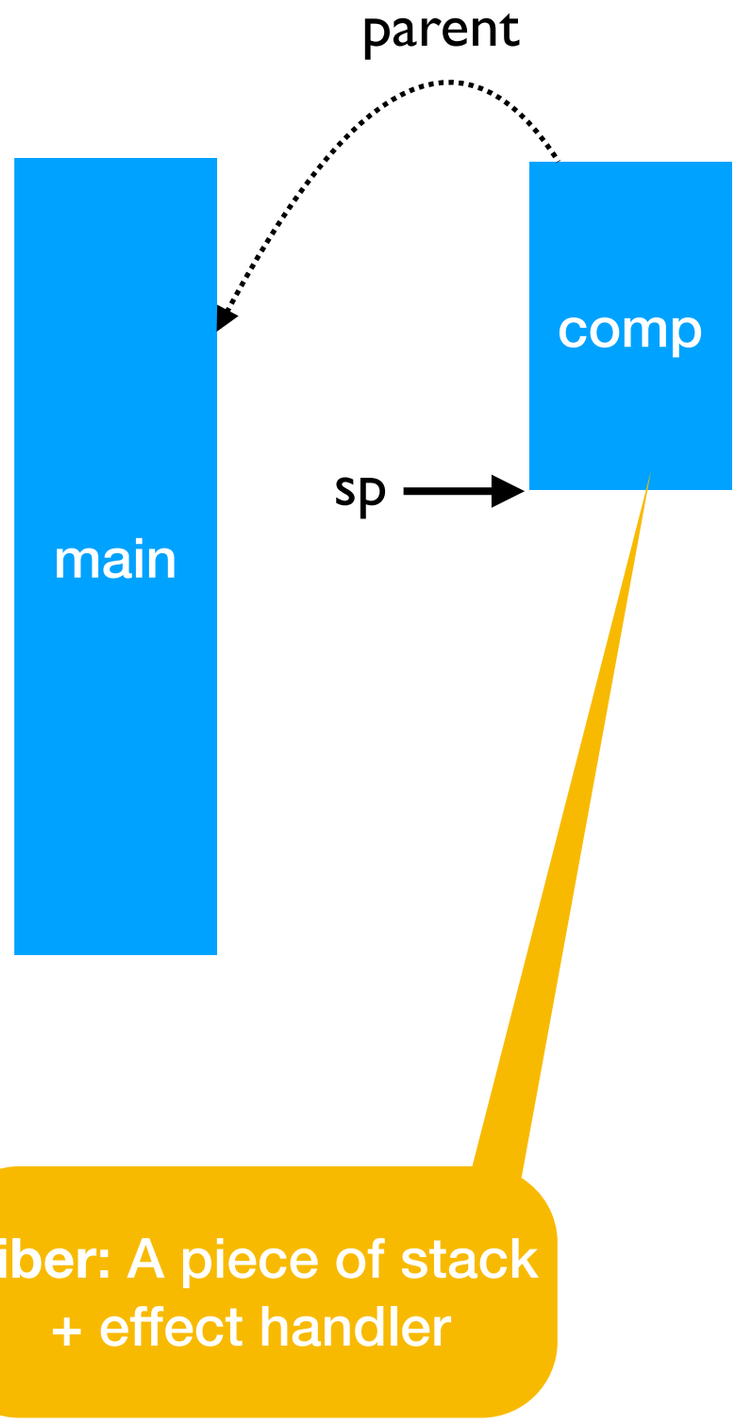
```
effect E : string

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 →

Stepping through the example

```
effect E : string

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 "
```

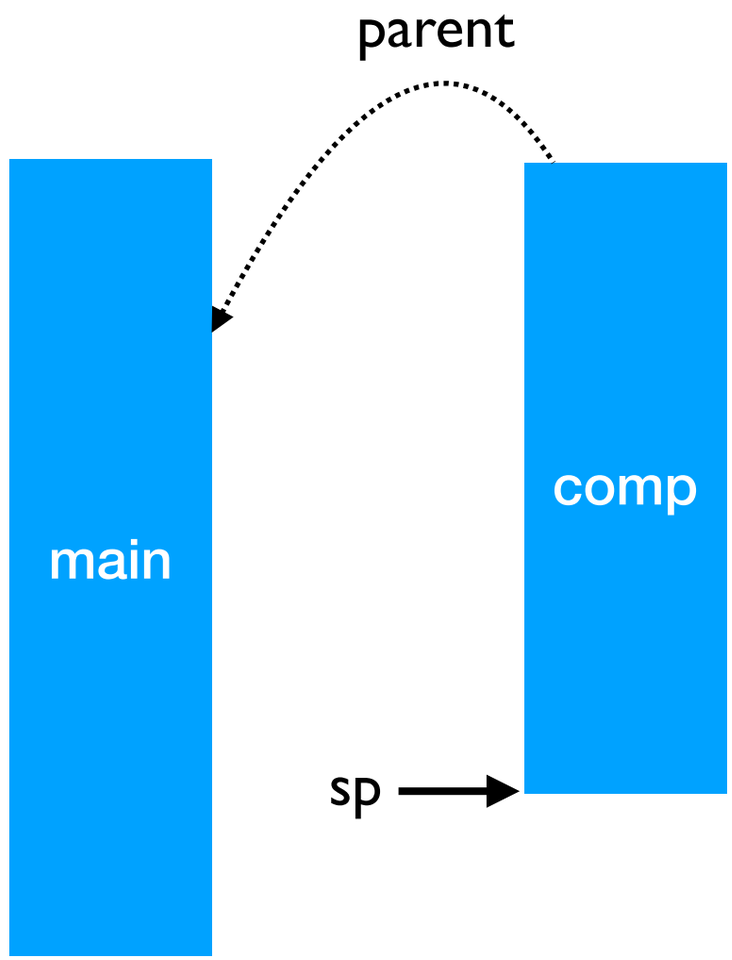
pc →



Stepping through the example

```
effect E : string

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 →

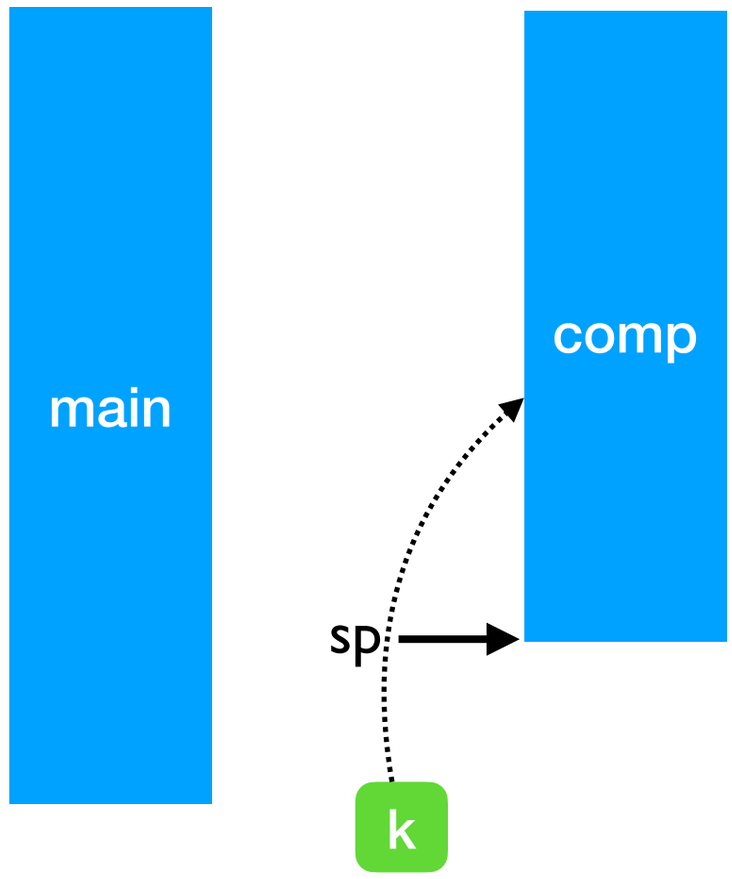


0

Stepping through the example

```
effect E : string

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 →

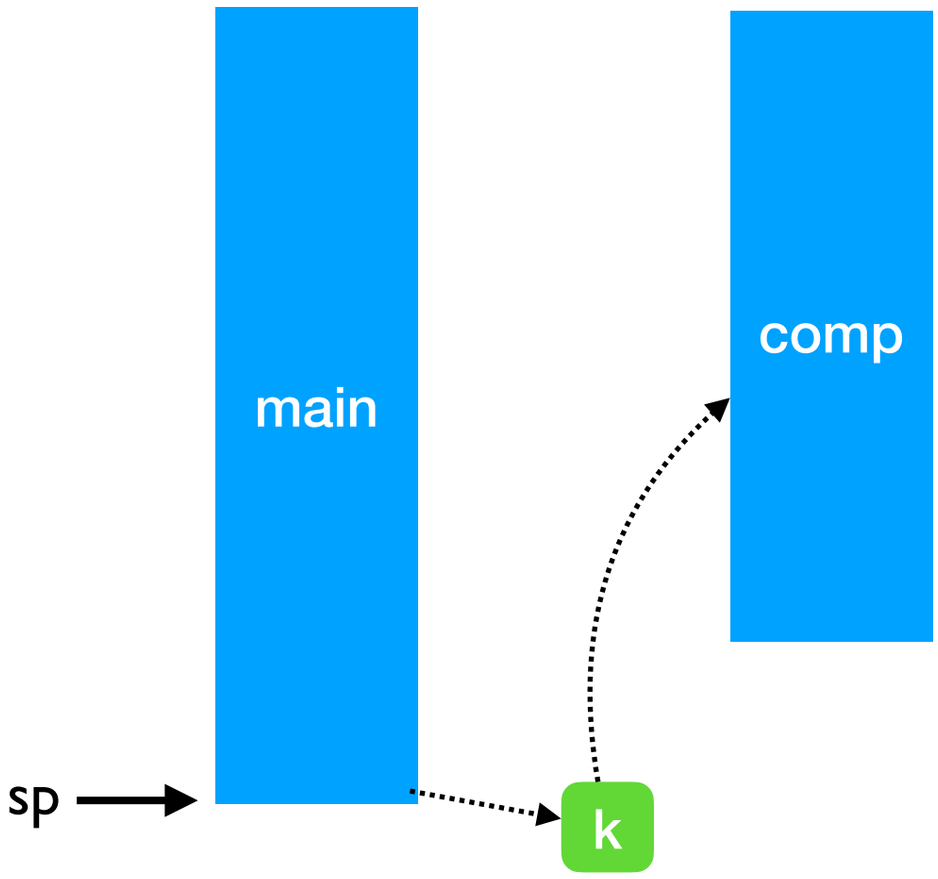


0

Stepping through the example

```
effect E : string

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 →

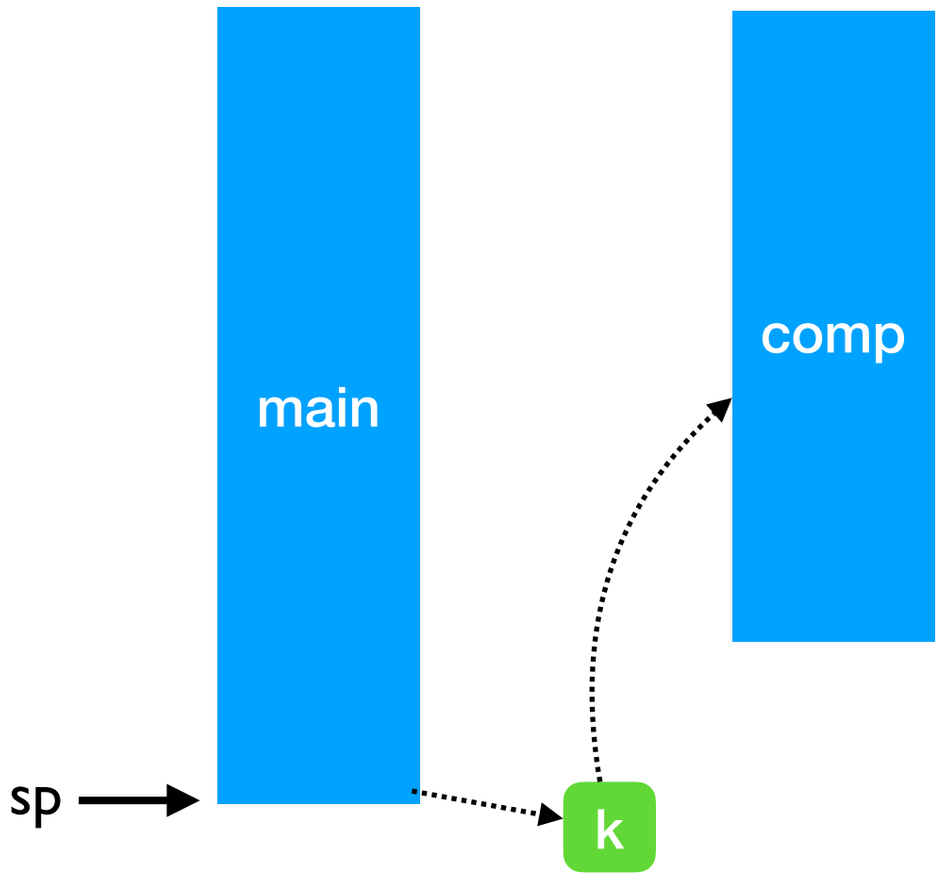


0

Stepping through the example

```
effect E : string

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 →

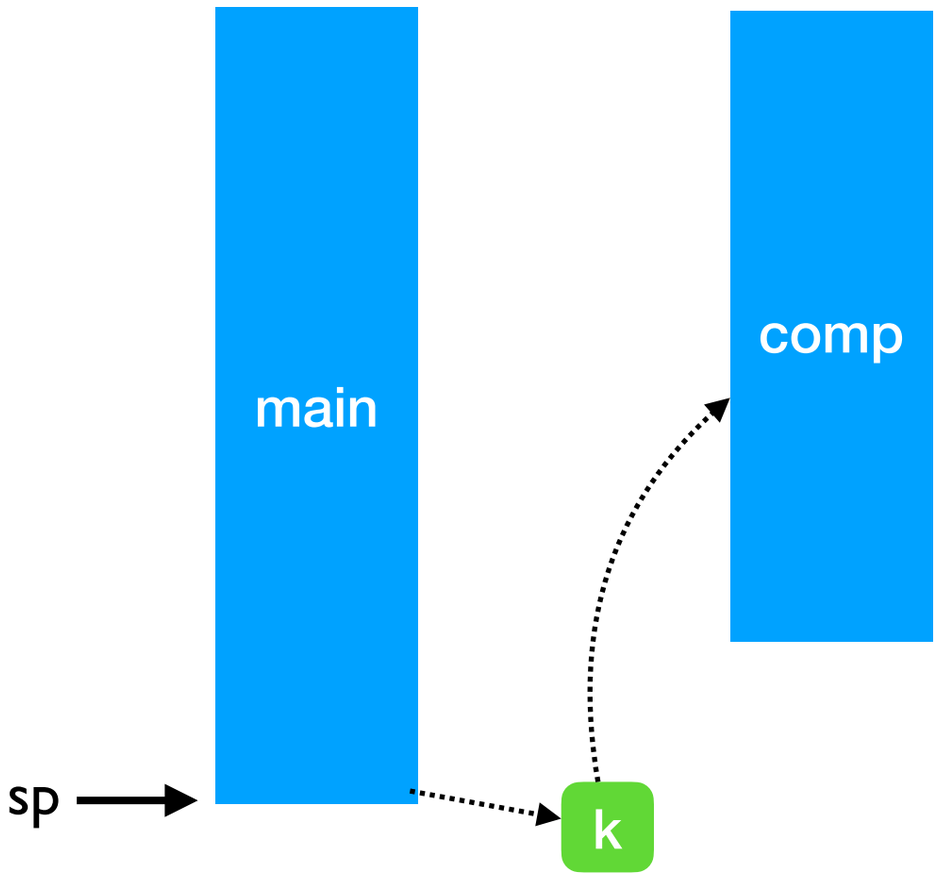


0

Stepping through the example

```
effect E : string

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 →

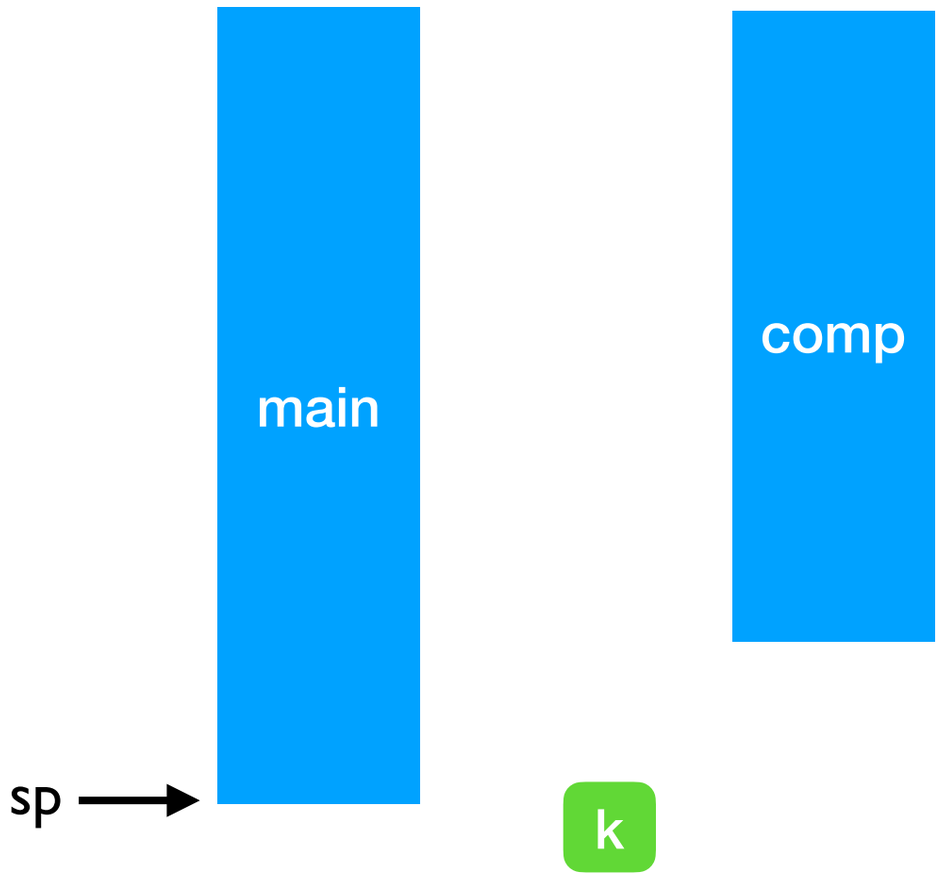


0 1

Stepping through the example

```
effect E : string

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 →

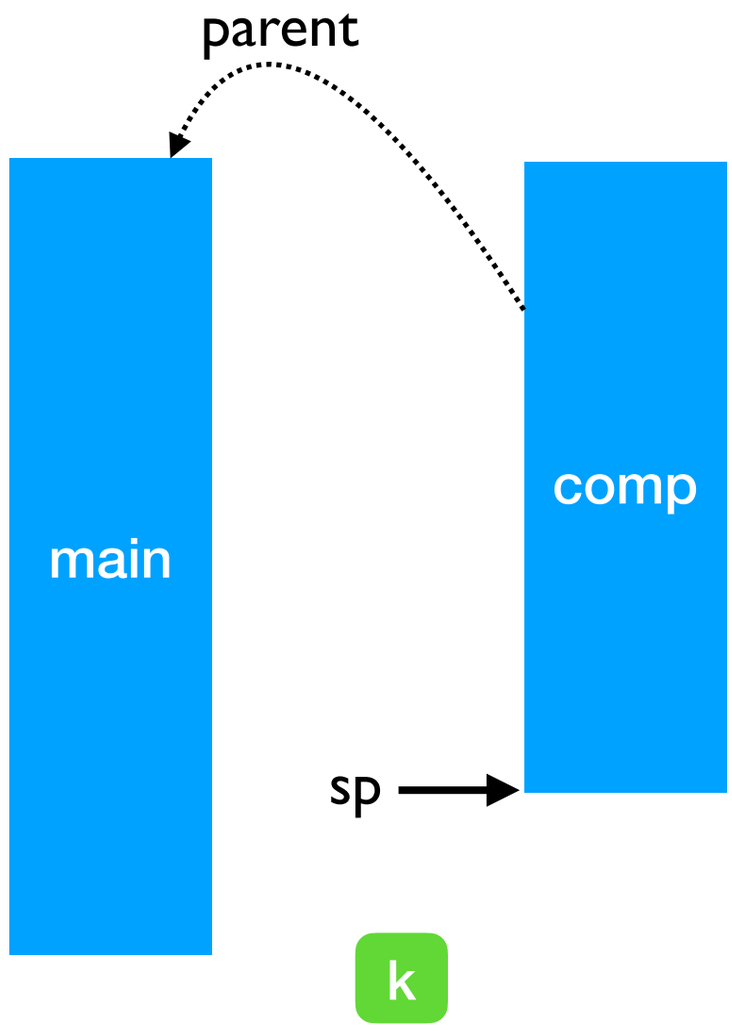


0 1

Stepping through the example

```
effect E : string

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 →

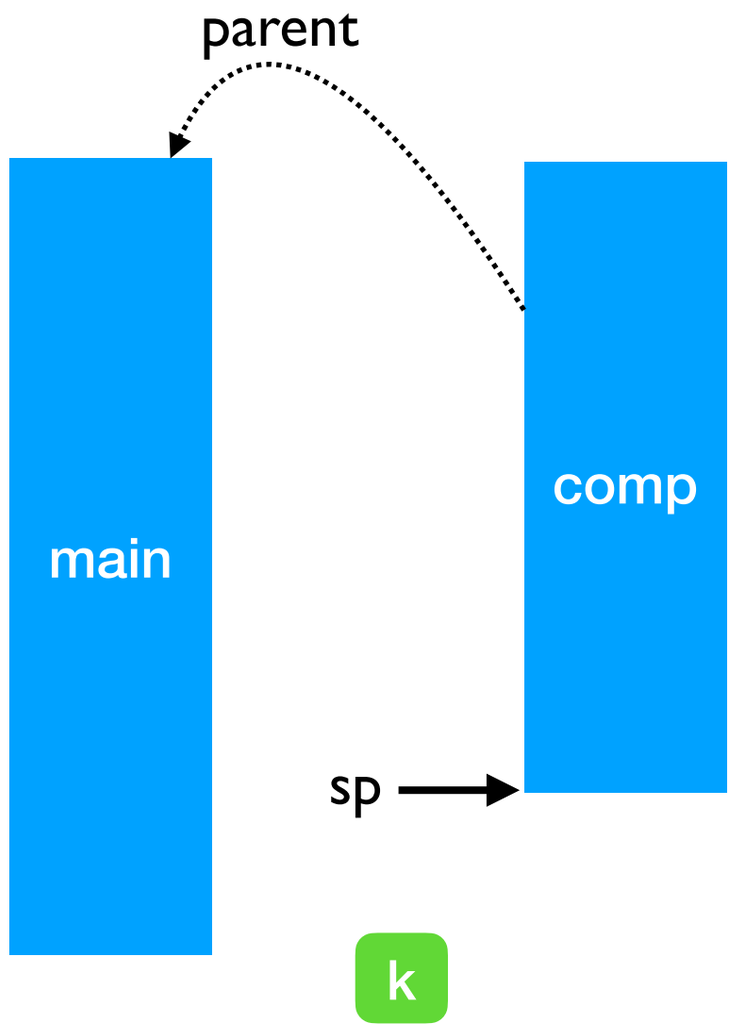


0 1

Stepping through the example

```
effect E : string

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 →

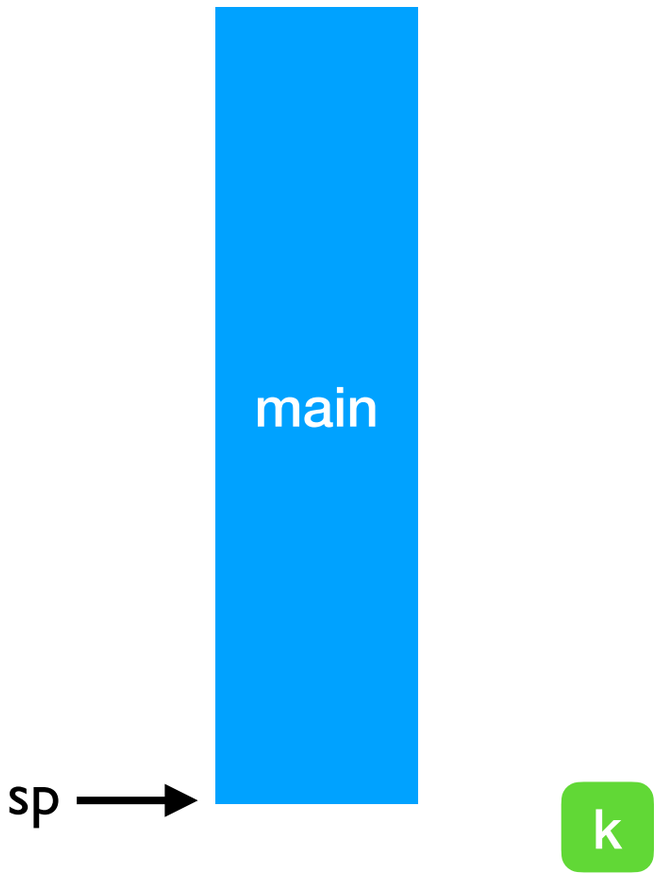


0 1 2

Stepping through the example

```
effect E : string

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 →

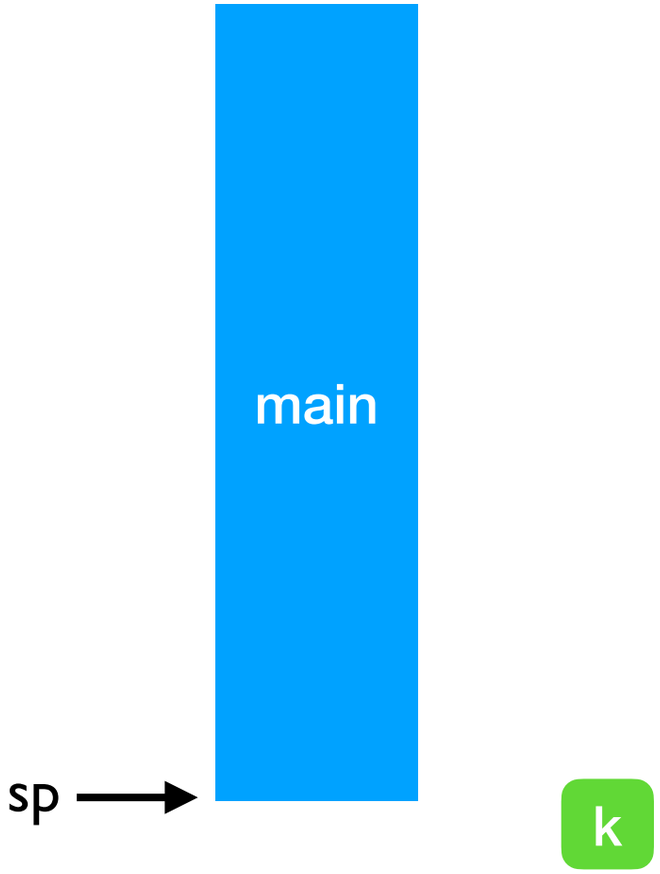


0 1 2 3

Stepping through the example

```
effect E : string

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 →



0 1 2 3 4

Lightweight Threading

```
effect Fork  : (unit -> unit) -> unit
effect Yield : unit
```

Lightweight Threading

```
effect Fork  : (unit -> unit) -> unit
effect Yield : unit

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 ()
    | effect Yield k -> enqueue k; run_next ()
    | effect (Fork f) k -> enqueue k; spawn f
 in
 spawn main
```

Lightweight Threading

```
effect Fork  : (unit -> unit) -> unit
effect Yield : unit

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 ()
    | 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
```

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
```

- Direct-style (no monads)
- User-code need not be aware of effects

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

Generators

- Generators — non-continuous traversal of data structure by yielding values
  - ✦ Primitives in JavaScript and Python
  - ✦ Can be *derived automatically* from iterator using effect handlers

Generators

- Generators — non-continuous traversal of data structure by yielding values
  - ✦ Primitives in JavaScript and Python
  - ✦ Can be *derived automatically* from iterator using effect handlers
- Task — traverse a complete binary-tree of depth 25
  - ✦ 2^{26} stack switches

Generators

- Generators — non-continuous traversal of data structure by yielding values
  - ✦ Primitives in JavaScript and Python
  - ✦ Can be *derived automatically* from iterator using effect handlers
- Task — traverse a complete binary-tree of depth 25
  - ✦ 2^{26} stack switches
- *Iterator* — idiomatic recursive traversal

Generators

- Generators — non-continuous traversal of data structure by yielding values
  - ✦ Primitives in JavaScript and Python
  - ✦ Can be *derived automatically* from iterator using effect handlers
- Task — traverse a complete binary-tree of depth 25
  - ✦ 2^{26} stack switches
- *Iterator* — idiomatic recursive traversal
- Generator
  - ✦ Hand-written generator (*hw-generator*)
    - ✤ CPS translation + defunctionalization to remove intermediate closure allocation
  - ✦ Generator using effect handlers (*eh-generator*)

Performance: Generators

Multicore OCaml

Variant	Time (milliseconds)
Iterator (baseline)	202
hw-generator	837 (**3.76x**)
eh-generator	1879 (**9.30x**)

Performance: Generators

Multicore OCaml

Variant	Time (milliseconds)
Iterator (baseline)	202
hw-generator	837 (**3.76x**)
eh-generator	1879 (**9.30x**)

nodejs 14.07

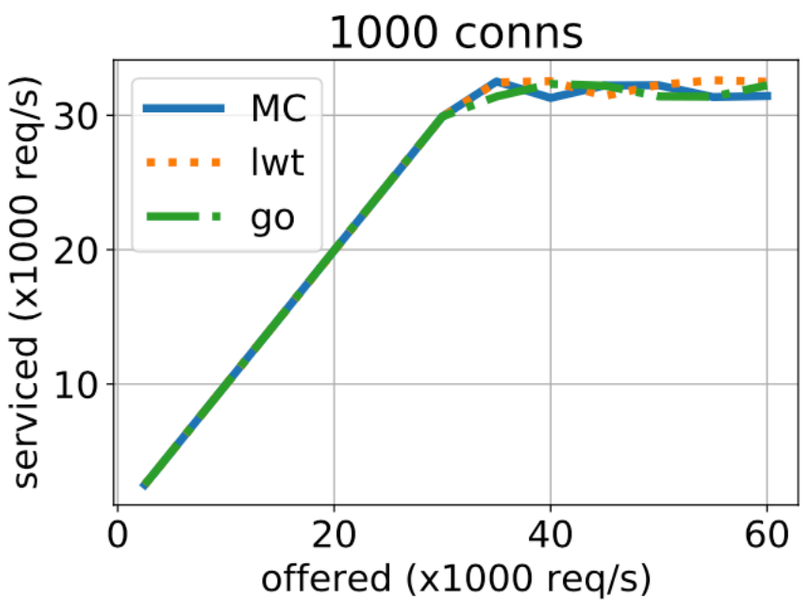
Variant	Time (milliseconds)
Iterator (baseline)	492
generator	43842 (**89.1x**)

Performance: WebServer

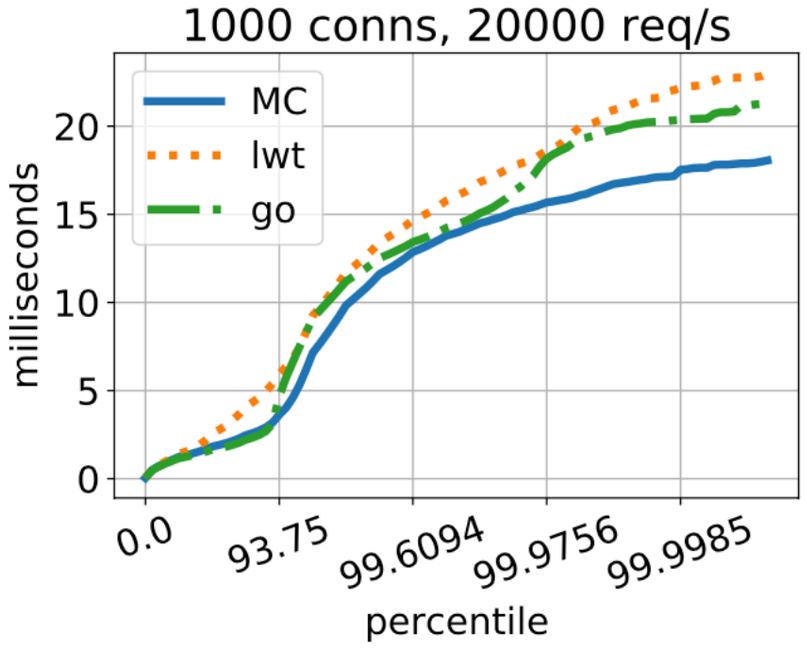
- Effect handlers for asynchronous I/O in direct-style
  - ✦ https://github.com/kayceesrk/ocaml-aeio/
- Variants
  - ✦ **Go** + net/http (GOMAXPROCS=1)
  - ✦ OCaml + http/af + **Lwt** (explicit callbacks)
  - ✦ OCaml + http/af + Effect handlers (**MC**)
- Performance measured using wrk2

Performance: WebServer

- Effect handlers for asynchronous I/O in direct-style
  - ✦ https://github.com/kayceesrk/ocaml-aeio/
- Variants
  - ✦ **Go** + net/http (GOMAXPROCS=1)
  - ✦ OCaml + http/af + **Lwt** (explicit callbacks)
  - ✦ OCaml + http/af + Effect handlers (**MC**)
- Performance measured using wrk2



(a) Throughput



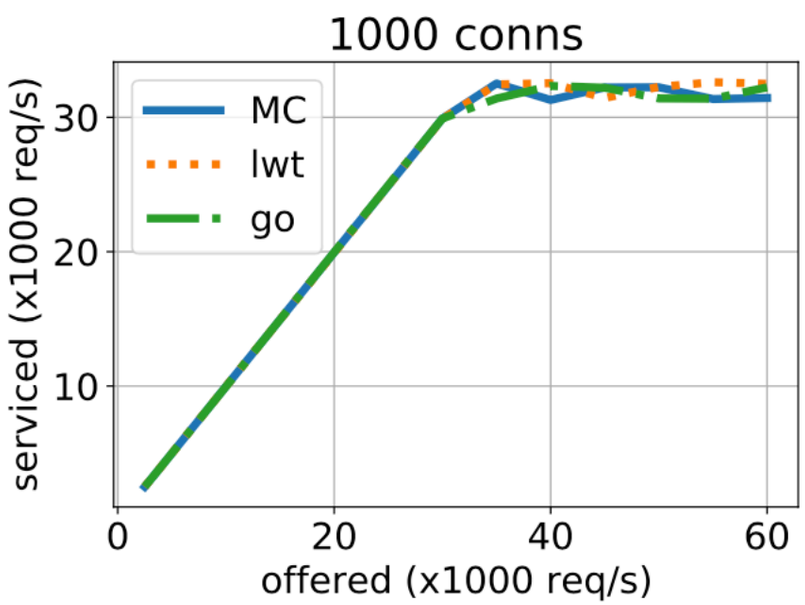
(b) Tail latency

Performance: WebServer

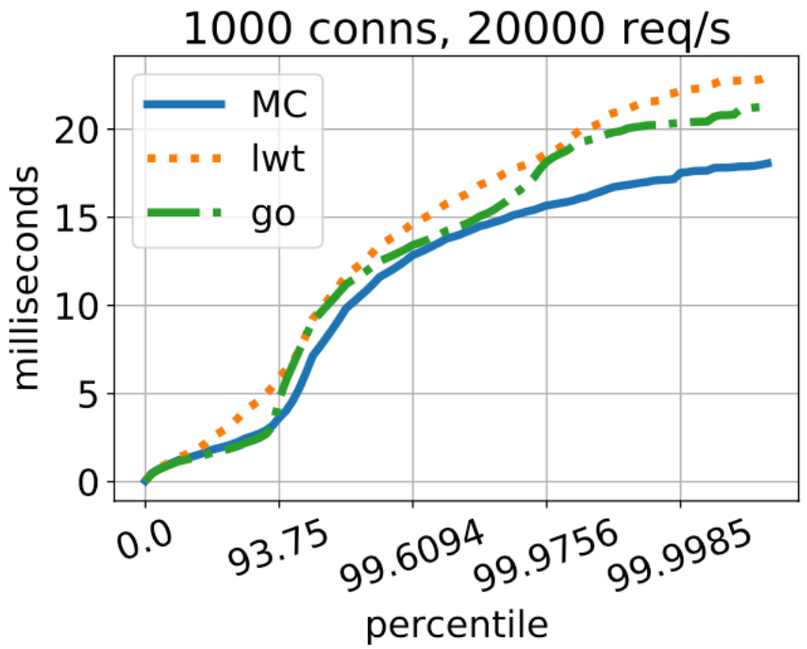
- Effect handlers for asynchronous I/O in direct-style
  - ✦ https://github.com/kayceesrk/ocaml-aeio/
- Variants
  - ✦ **Go** + net/http (GOMAXPROCS=1)
  - ✦ OCaml + http/af + **Lwt** (explicit callbacks)
  - ✦ OCaml + http/af + Effect handlers (**MC**)

- Direct style (no monadic syntax)
- Can use OCaml exceptions!
- Backtrace per thread (request)
- gdb & perf work!

- Performance measured using wrk2



(a) Throughput



(b) Tail latency

Upstreaming Plan

Upstreaming Plan

1. Domains-only multicore to be upstreamed first

Upstreaming Plan

1. Domains-only multicore to be upstreamed first
2. Runtime support for effect handlers
    - No effect syntax but all the compiler and runtime bits in

Upstreaming Plan

1. Domains-only multicore to be upstreamed first
2. Runtime support for effect handlers
   - No effect syntax but all the compiler and runtime bits in
3. Effect system
   a. Track user-defined effects in the type
   b. Track ambinet effects (ref, IO) in the type
   c. *OCaml becomes a pure language* (in the Haskell sense).

Upstreaming Plan

1. Domains-only multicore to be upstreamed first
2. Runtime support for effect handlers
   - No effect syntax but all the compiler and runtime bits in
3. Effect system
   a. Track user-defined effects in the type
   b. Track ambinet effects (ref, IO) in the type
   c. *OCaml becomes a pure language* (in the Haskell sense).

```
let foo () = print_string "hello, world"

val foo : unit -[ io ]-> unit
```

Syntax is still in the works

Multicore OCaml + Tezos

- Thanks to Tezos Foundation for funding Multicore OCaml development!

Multicore OCaml + Tezos

- Thanks to Tezos Foundation for funding Multicore OCaml development!
- Multicore + Tezos
  - ✦ Parallel Lwt preemptive tasks
  - ✦ Direct-style asynchronous IO library
    - ✤ Bridge the gap between Async and Lwt
  - ✦ Parallelising Irmin (storage layer of Tezos)

Multicore OCaml + Tezos

- Thanks to Tezos Foundation for funding Multicore OCaml development!
- Multicore + Tezos
  - ✦ Parallel Lwt preemptive tasks
  - ✦ Direct-style asynchronous IO library
    - ✤ Bridge the gap between Async and Lwt
  - ✦ Parallelising Irmin (storage layer of Tezos)
- An end-to-end Multicore Tezos demonstrator (mid-2021)

Thanks!

Install Multicore OCaml

```
$ opam switch create 4.10.0+multicore \
    --packages=ocaml-variants.4.10.0+multicore \
    --repositories=multicore=git+https://github.com/ocaml-multicore/multicore-opam.git,default
```

- Multicore OCaml — https://github.com/ocaml-multicore/ocaml-multicore
- Effects Examples — https://github.com/ocaml-multicore/effects-examples
- Sivaramakrishnan et al, "*Retrofitting Parallelism onto OCaml*", ICFP 2020
- Dolan et al, "*Concurrent System Programming with Effect Handlers*", TFP 2017