OCaml’s Parallel Runtime System

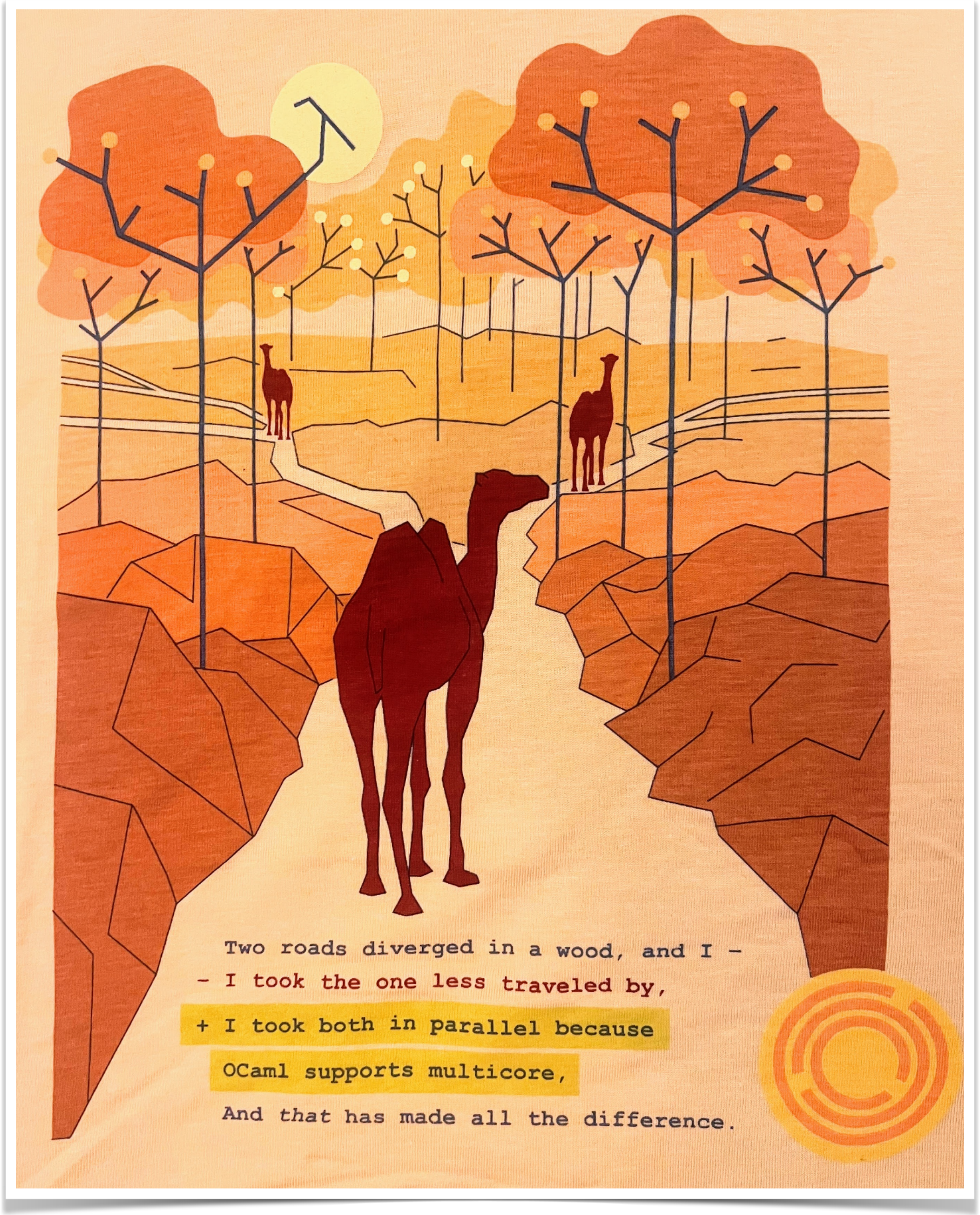
KC Sivaramakrishnan

Parallel Functional Programming @ Chalmers

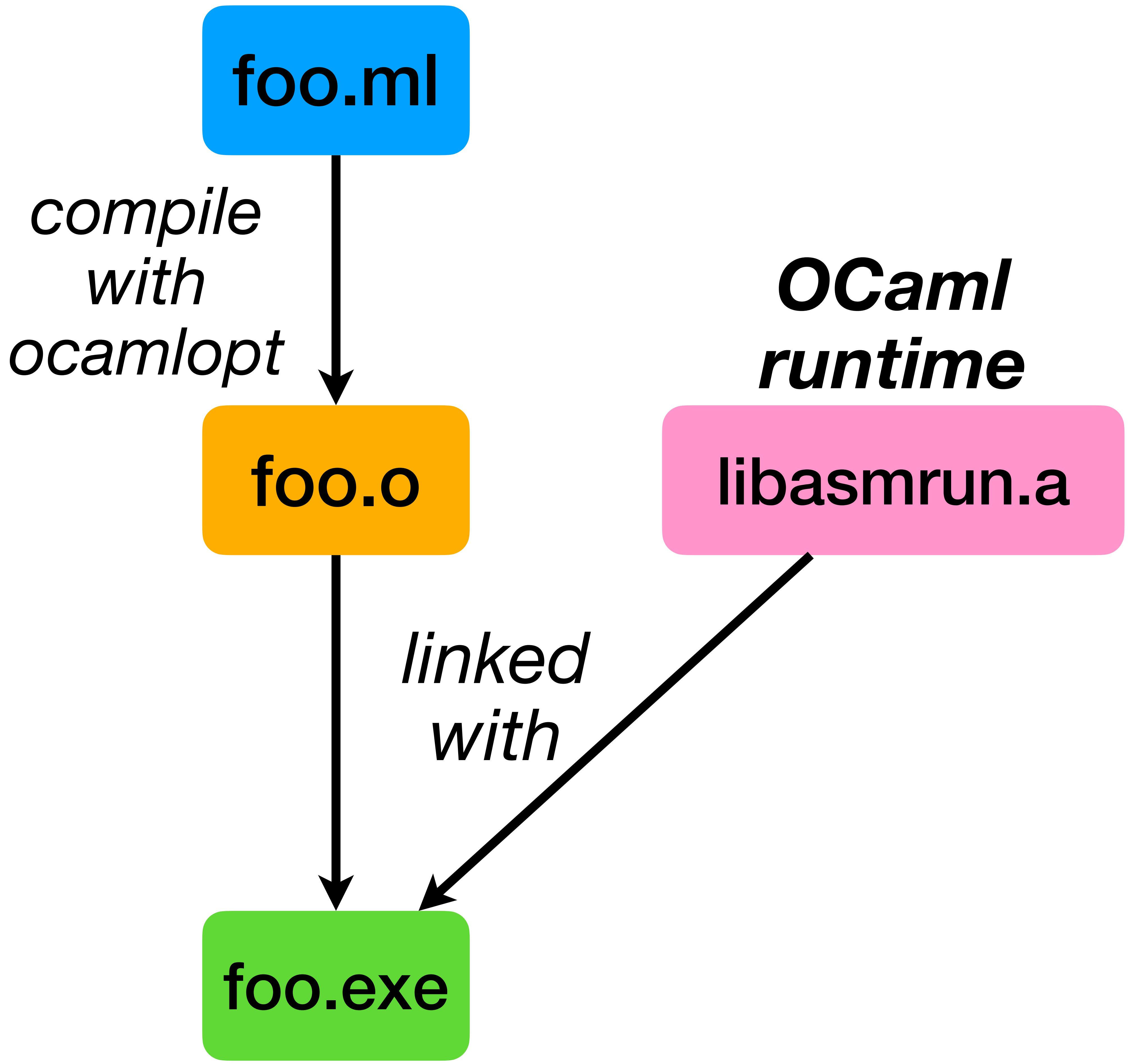
May 2025

OCaml 5

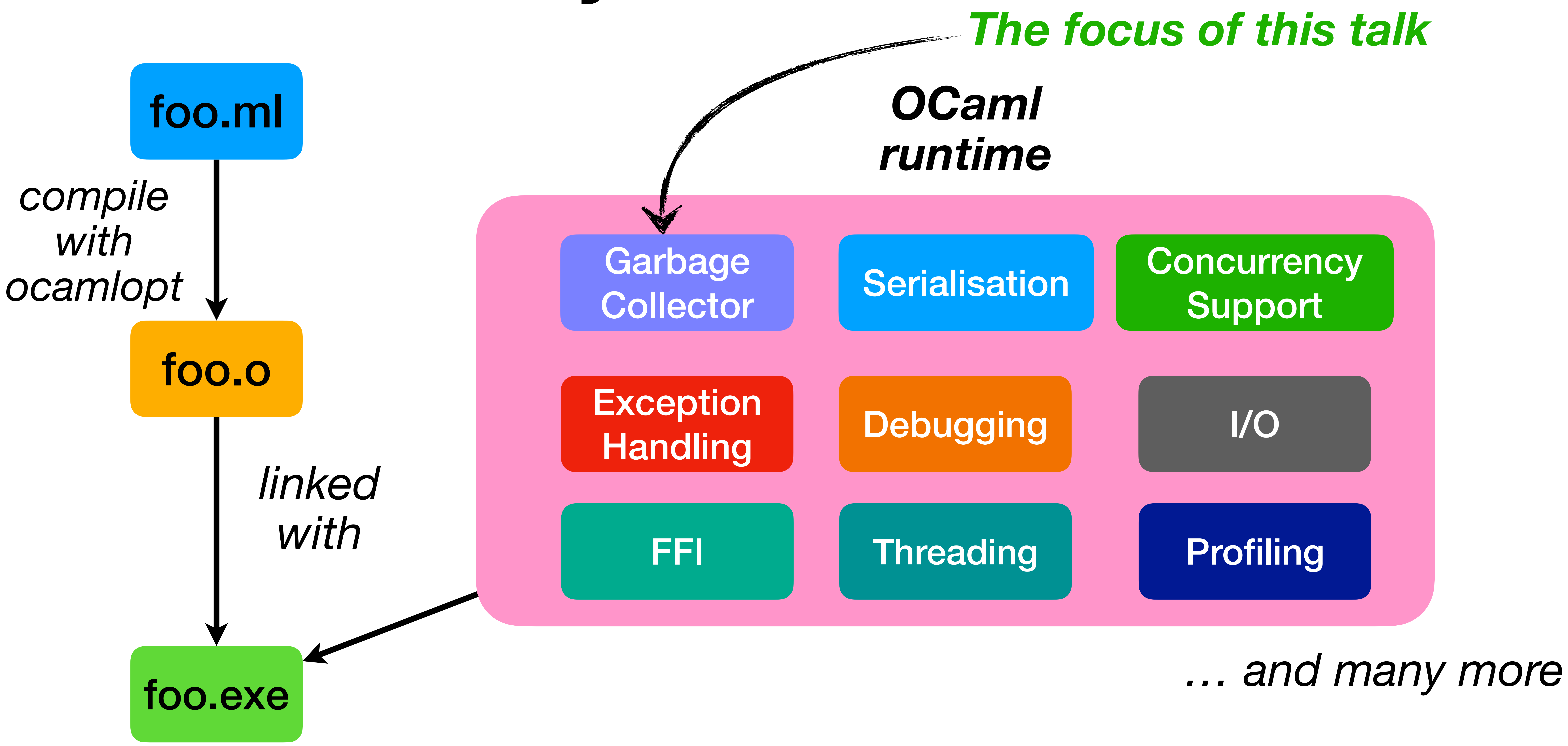
- Native-support for ***concurrency*** and ***parallelism*** to OCaml
- Started in 2014 as "Multicore OCaml" project
  - OCaml 5.0 released in Dec 2022
  - **5.1** — Sep 2023; **5.2** — May 2024; **5.3** — Jan 2025
- This talk
  - *What does a GC need to do to support parallelism?*



OCaml Runtime System



OCaml Runtime System



Today's lecture

1. OCaml 4 sequential GC design
2. OCaml 5 multicore GC design
3. Experience porting a multi-process application to multicore

OCaml Garbage Collector (GC)

Whence the GC

```
let rec unzip lst =
  match lst with
  | [] -> ([], [])
  | (x, y) :: rest ->
      let (xs, ys) = unzip rest in
      (x :: xs, y :: ys)
```

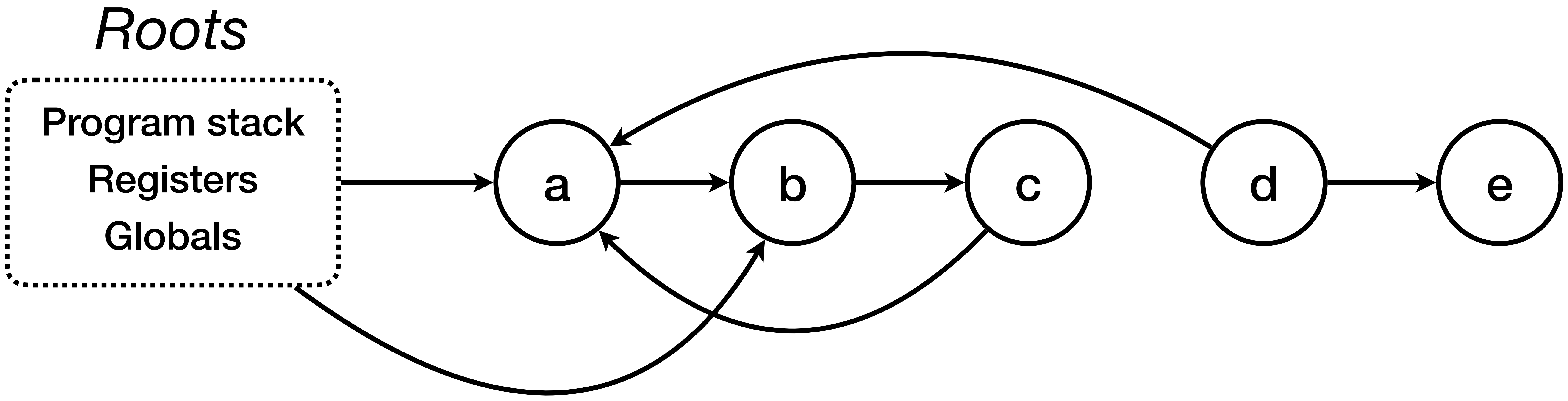
Allocations

Whence the GC

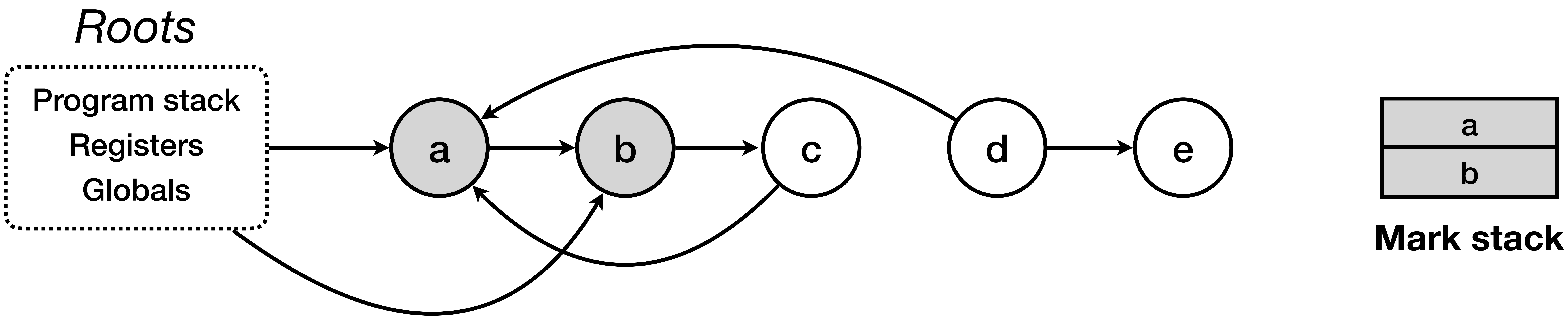
Free*

```
let rec unzip lst =
  match lst with
  | [] -> ([], [])
  | (x, y) :: rest ->
    let (xs, ys) = unzip rest in
    (x :: xs, y :: ys)
```

Mark and Sweep GC

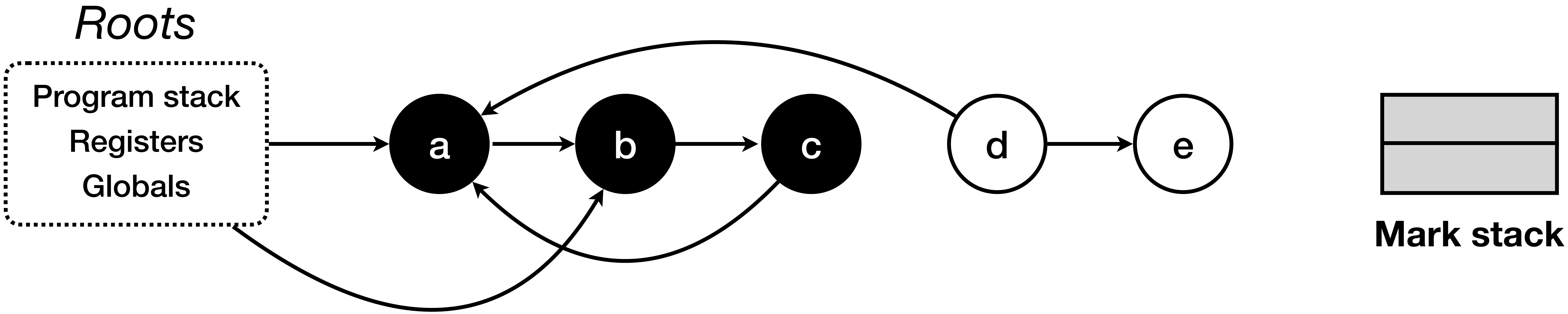


Mark and Sweep GC



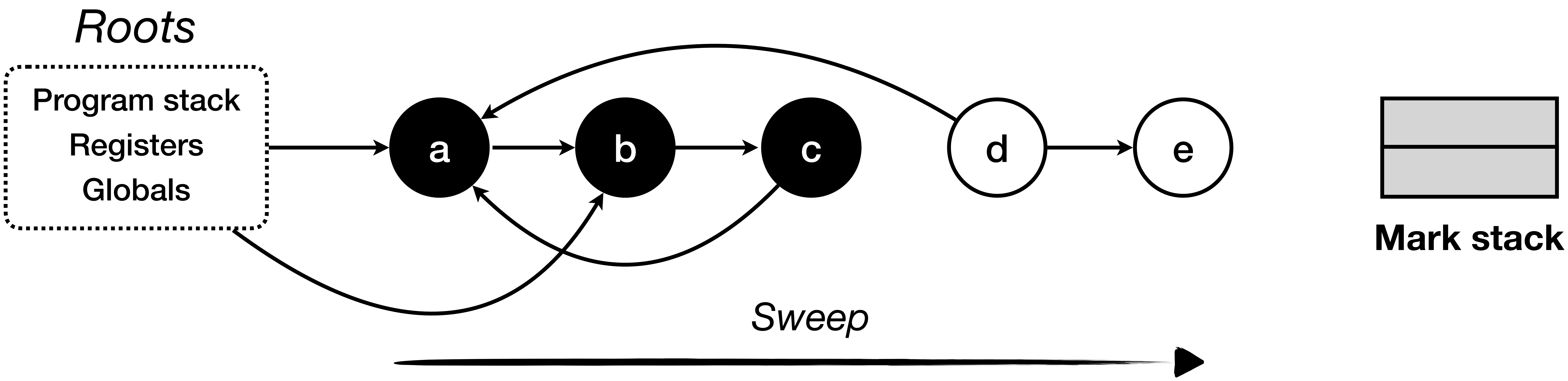
- Tri-color marking — White, Grey and Black
  - Phase 1 — Mark the roots

Mark and Sweep GC



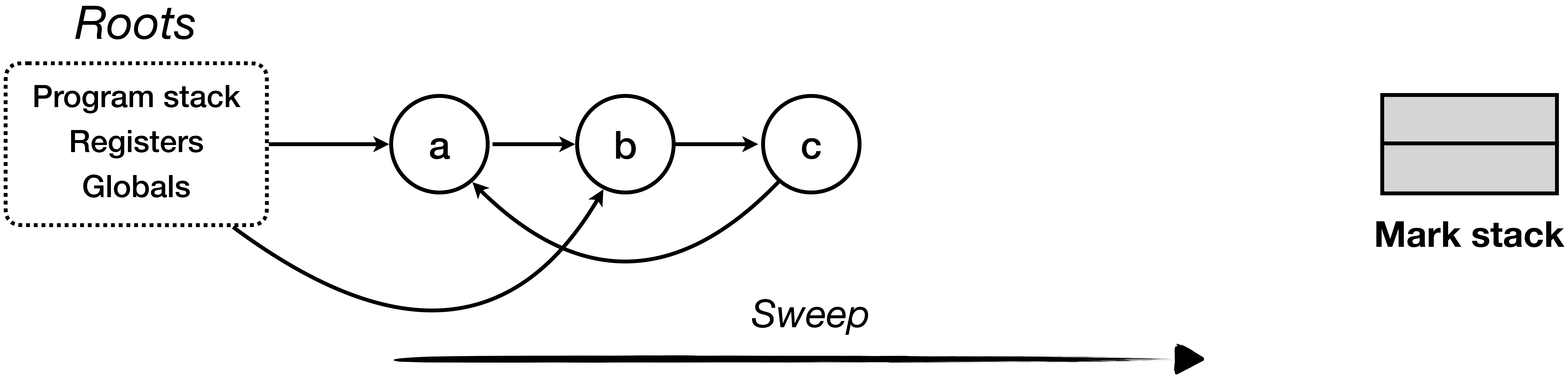
- Tri-color marking — White, Grey and Black
  - Phase 1 — Mark the roots
  - Phase 2 — DFS Mark

Mark and Sweep GC



- Tri-color marking — White, Grey and Black
  - Phase 1 — Mark the roots
  - Phase 2 — DFS Mark
  - Phase 3 — Sweep

Mark and Sweep GC



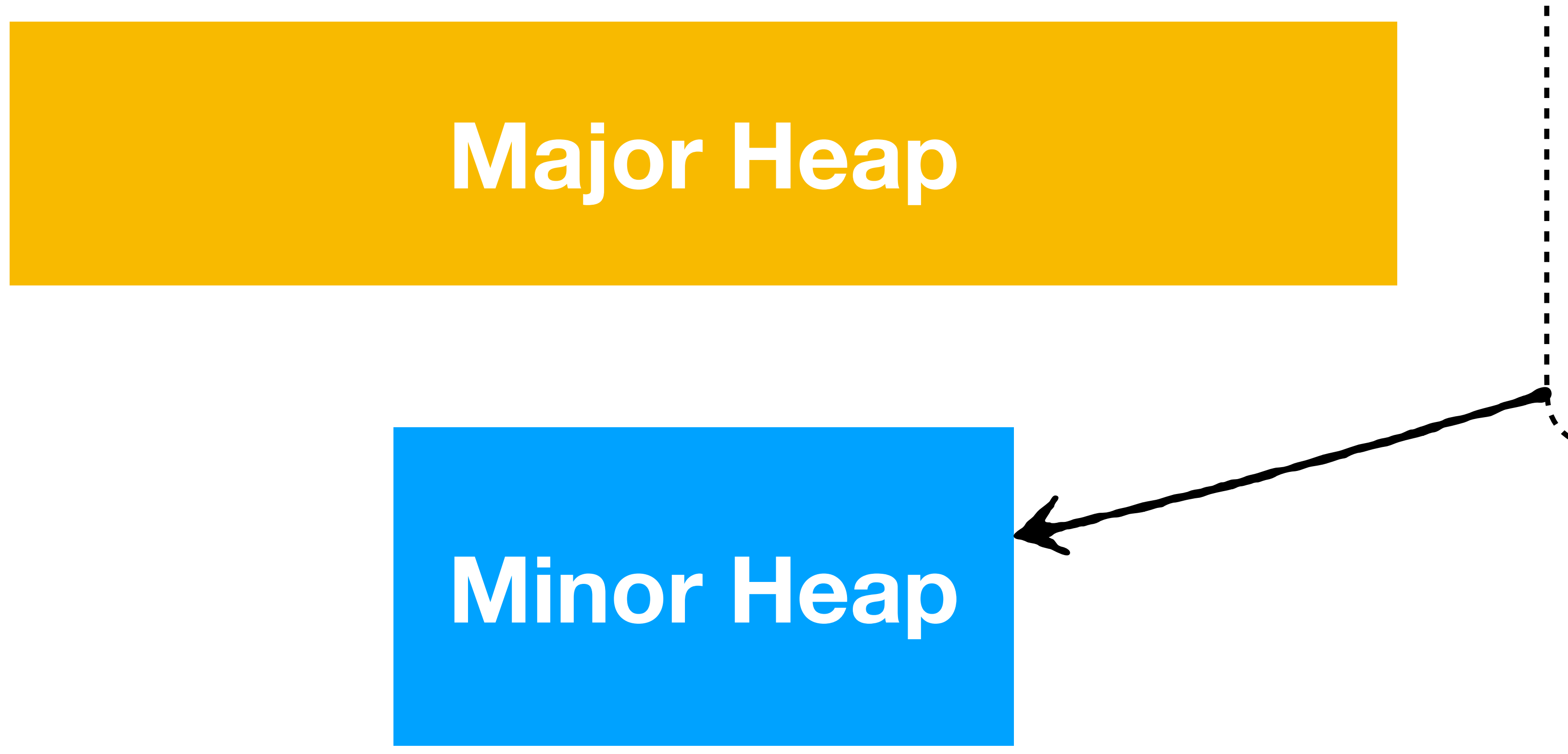
- Trigger GC when allocator doesn’t find a space (or some other metric)
- Time complexity
  - Marking is **O(reachable)**
  - Sweeping is **O(allocated)**

Mark and Sweep GC

- The biggest downside is ***latency***
- OCaml code (mutator) cannot run when the GC is running
    - Leads to ***multi-second pausetimes*** for GB-sized heaps
- How can we improve this?
    - ***Do not touch the entire heap for GC***
    - Avoid **O(reachable)** marking and **O(allocated)** sweeping

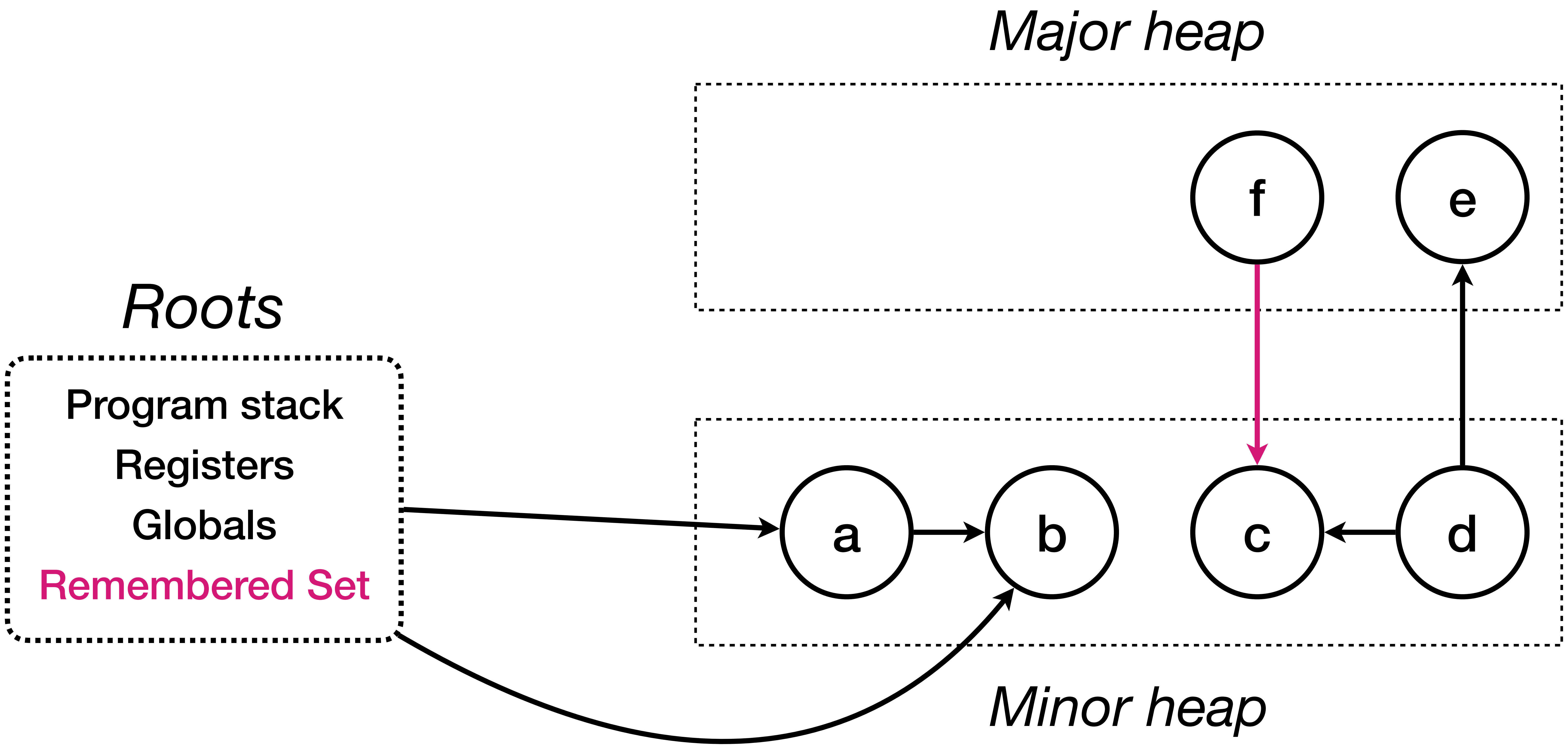
Generational GC

- Generational hypothesis
  - In a functional programming language, most objects die young

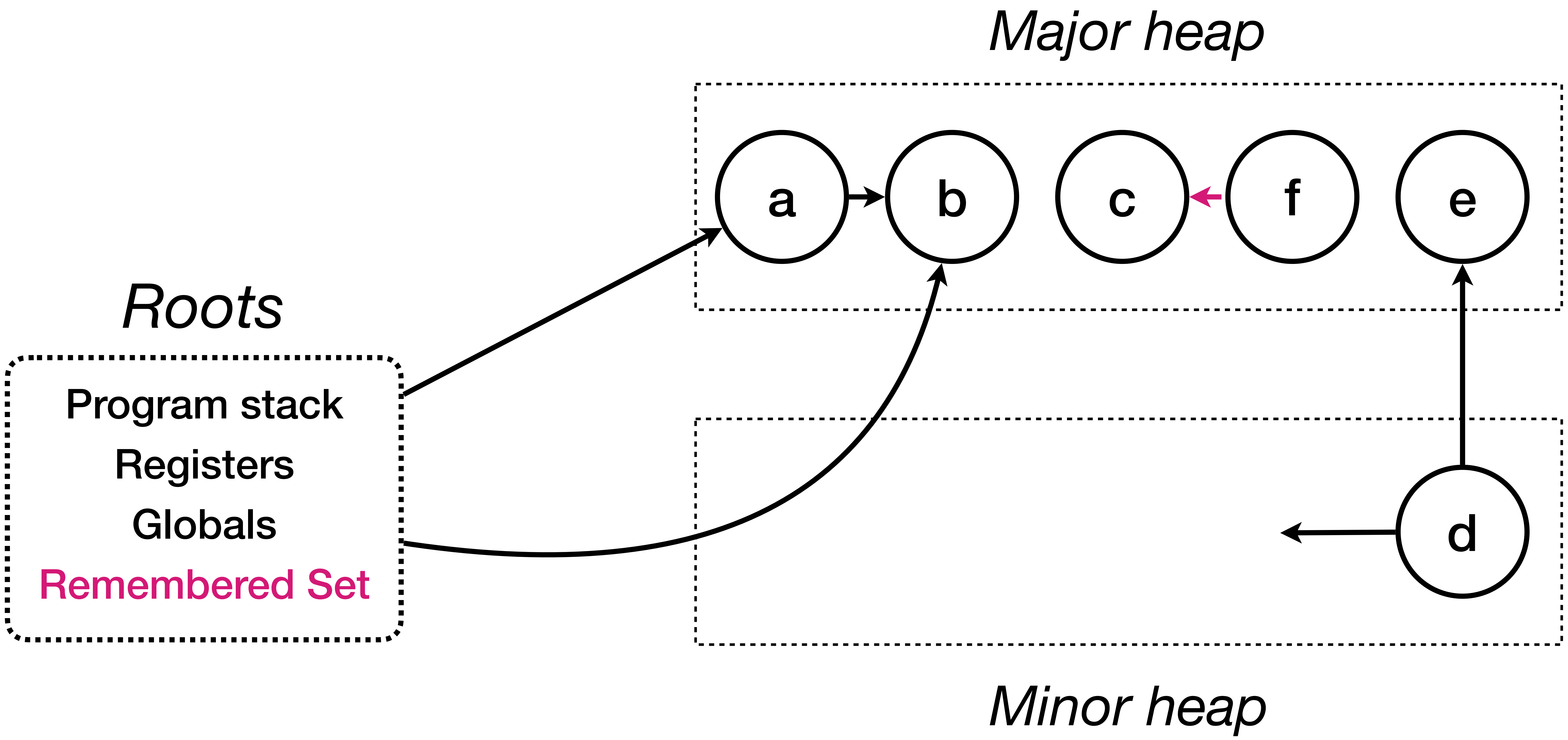


- Small (2 MB default)
- Bump pointer allocation
- Survivors *copied* to major heap

Generational GC



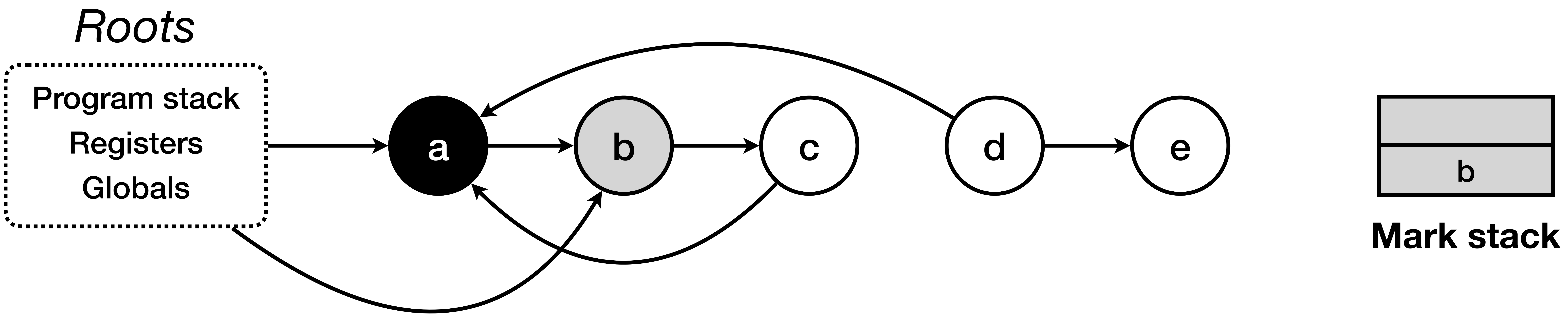
Generational GC



- Time complexity — ***O(reachable)***
  - Entire minor heap is free after copying
  - 10% survival rate
- Write barrier for remembered set
- Working set → Cache locality
- ***Major heap GC latency still remains :-(***

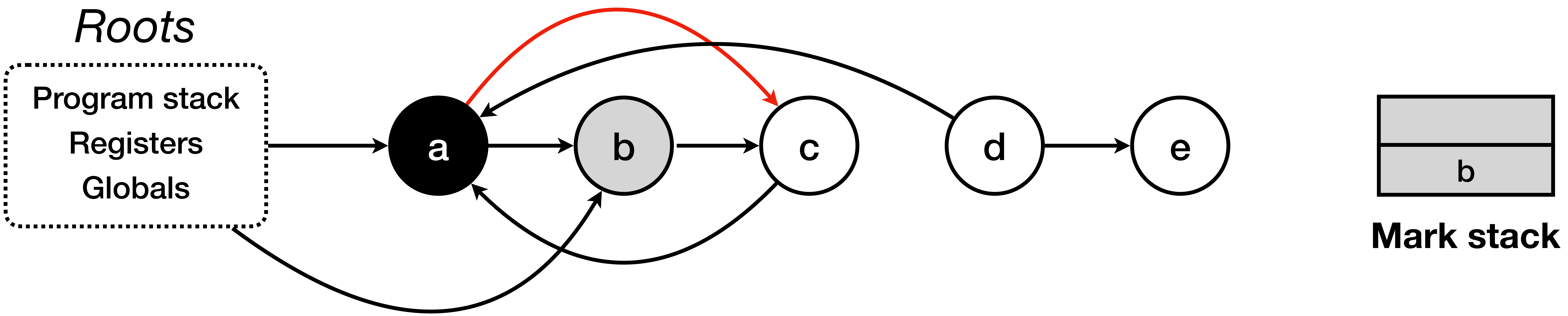
Incremental Mark-and-sweep GC

- Instead of marking and sweeping in one go, alternate between GC and mutator
  - *Graph will be changed by the mutator!*



Incremental Mark-and-sweep GC

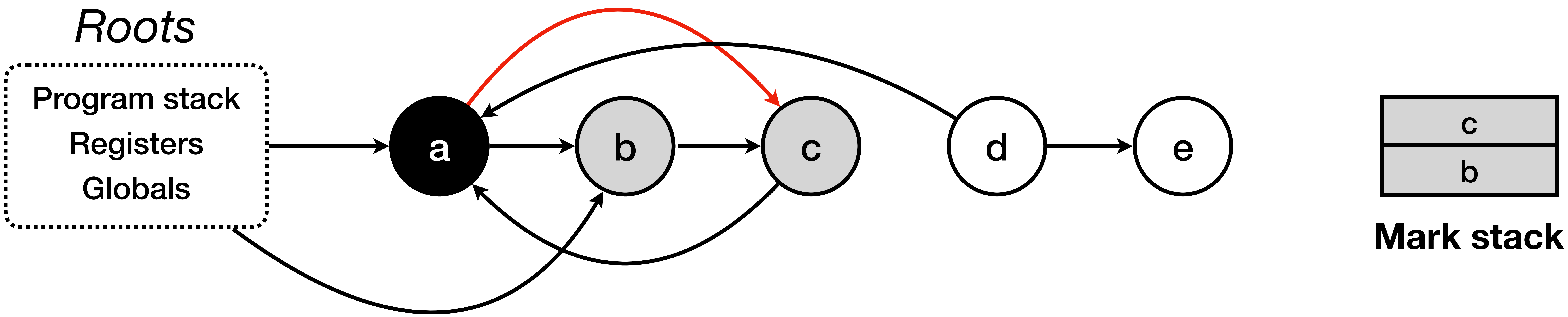
- Instead of marking and sweeping in one go, alternate between GC and mutator
  - *Graph will be changed by the mutator!*



- Finishing the GC at this state ***will free "c"*** leaving a ***dangling pointer***

Incremental Mark-and-sweep GC

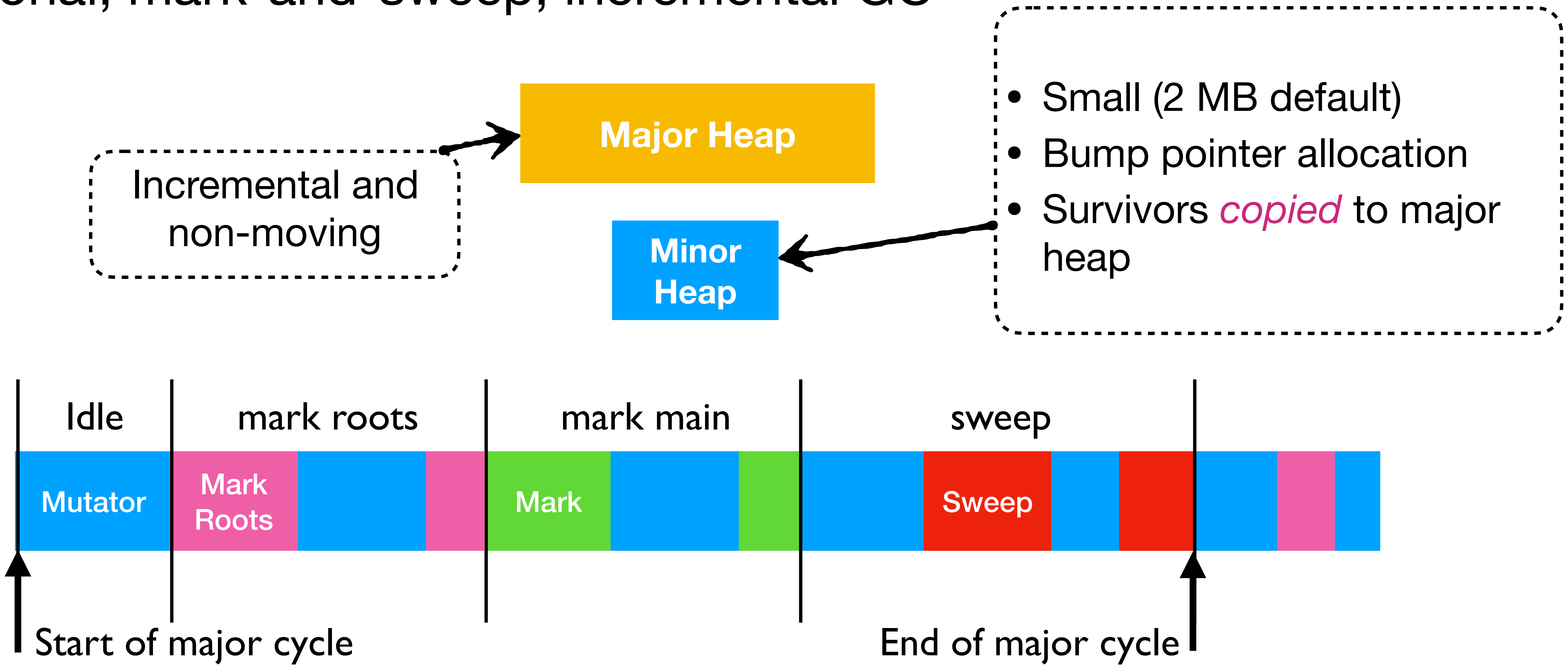
- Snapshot-at-the-beginning (SATB) GC
  - Use a ***deletion barrier*** to do a bit of work in the mutator



- Grey c when b → c pointer is deleted
- Snapshot-at-the-beginning property
  - Ensures that all objects reachable at the beginning of the cycle are reachable at the end

Summary — OCaml 4 GC

- Generational, mark-and-sweep, incremental GC

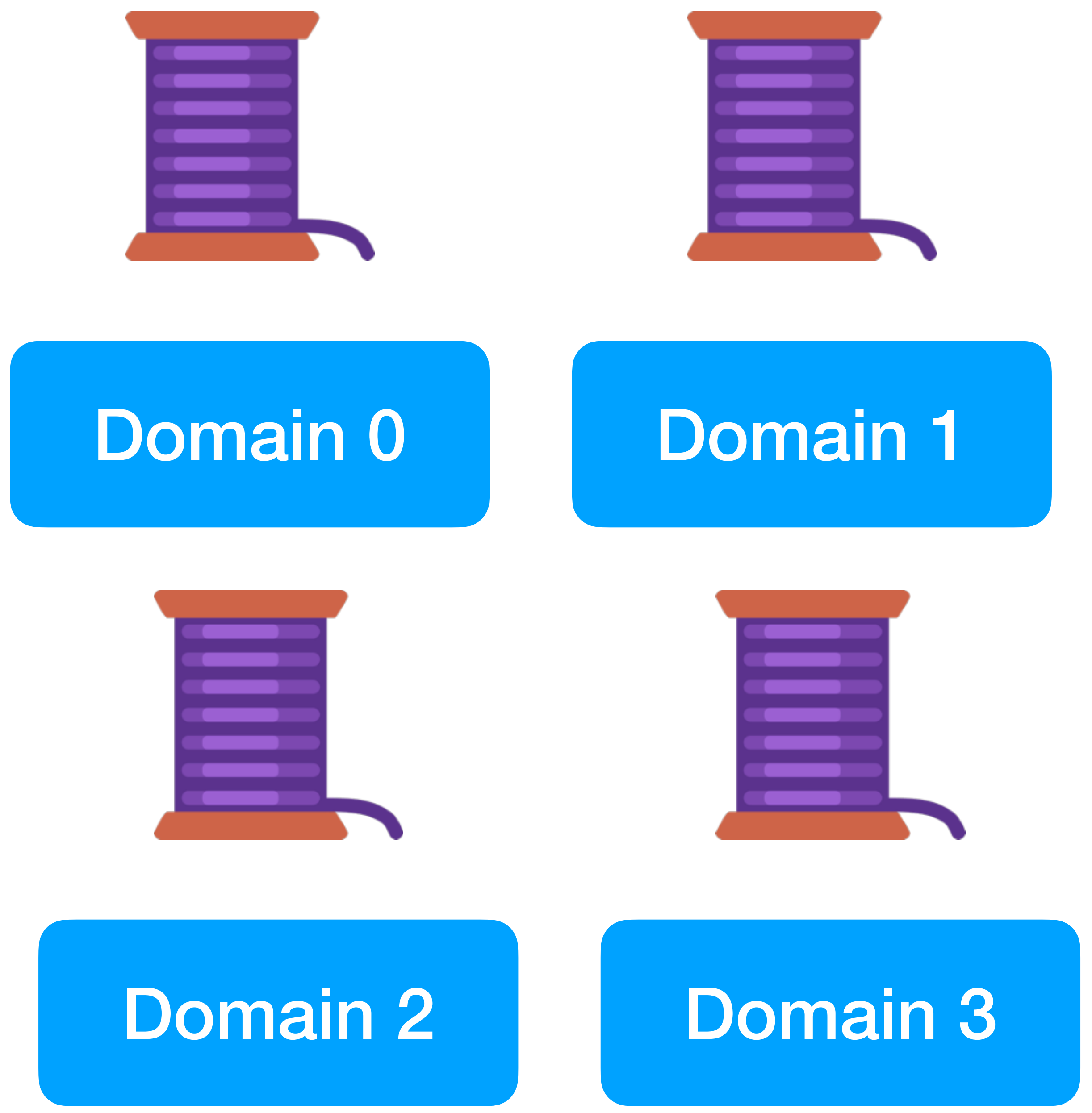


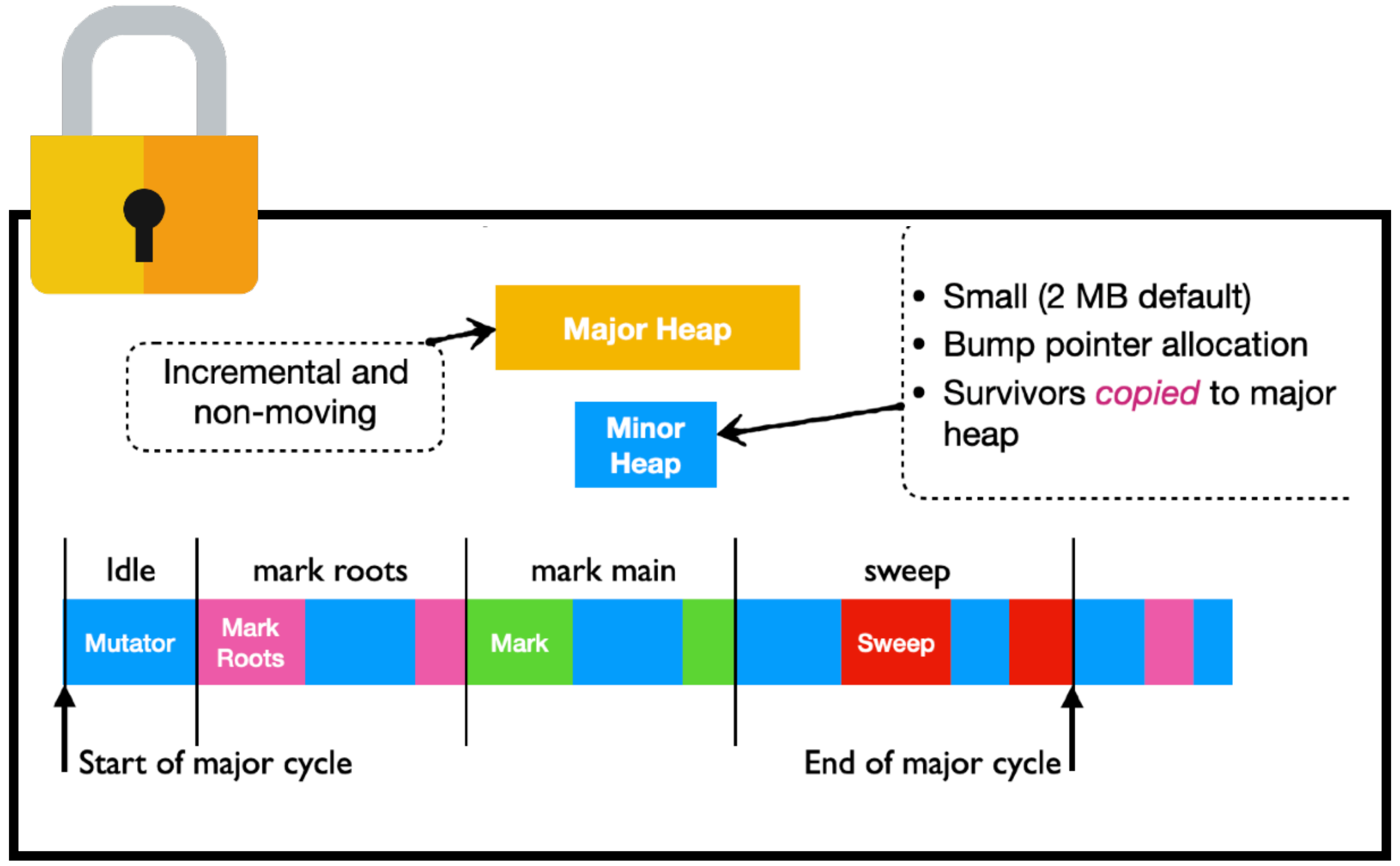
- Fast local allocations
- Max GC latency **< 10 ms**, 99th percentile latency **< 1 ms**

Going Multicore

Domains

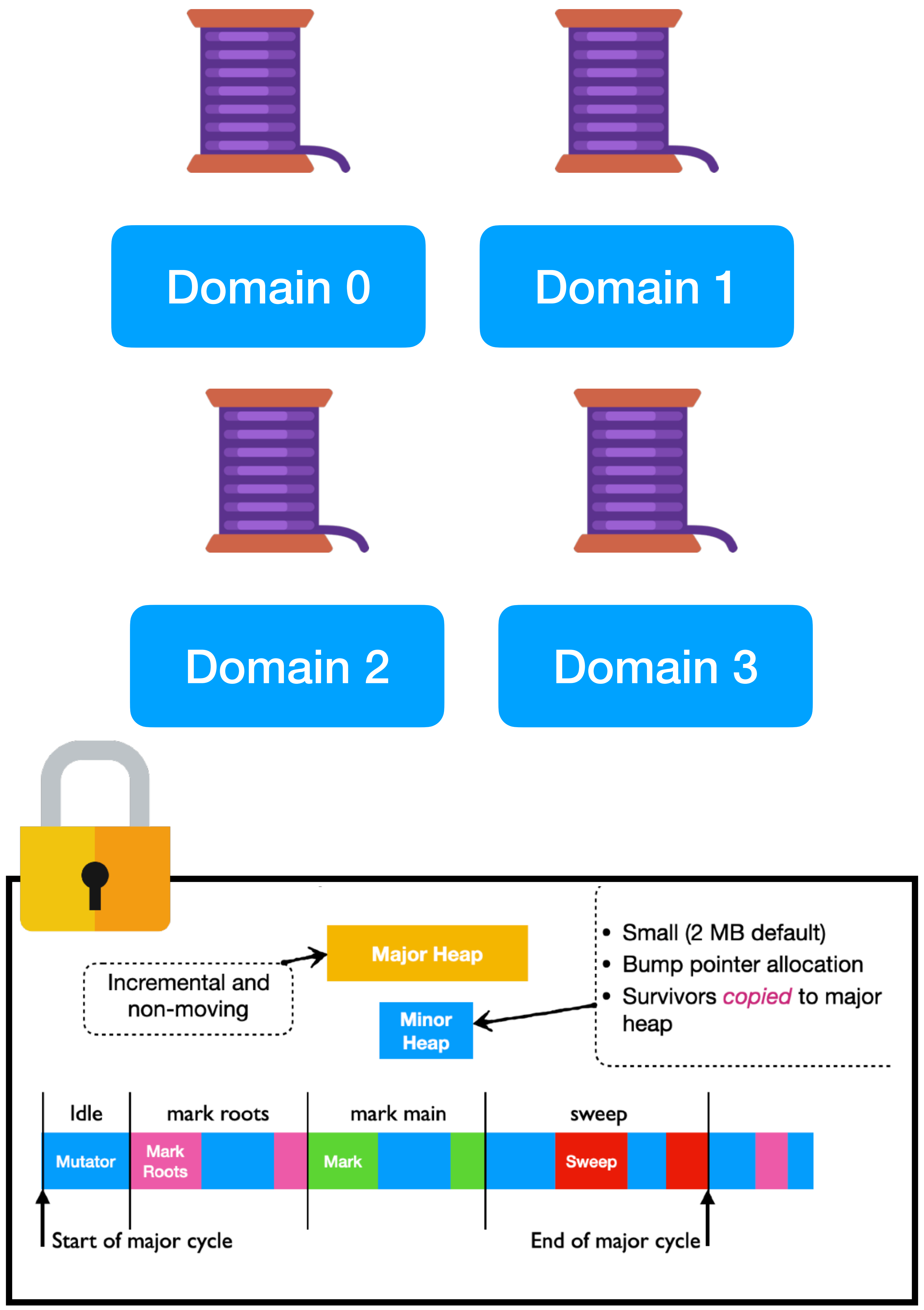
- A unit of parallelism
- ***Heavyweight*** — maps onto an OS thread
  - Aim to have 1 domain per physical core
- Stdlib exposes
  - Spawn & join, Mutex, Condition, domain-local storage
  - Atomic references
- A multicore language needs a multicore runtime!
  - A naive ***Stop-the-world GC*** would limit parallel scalability





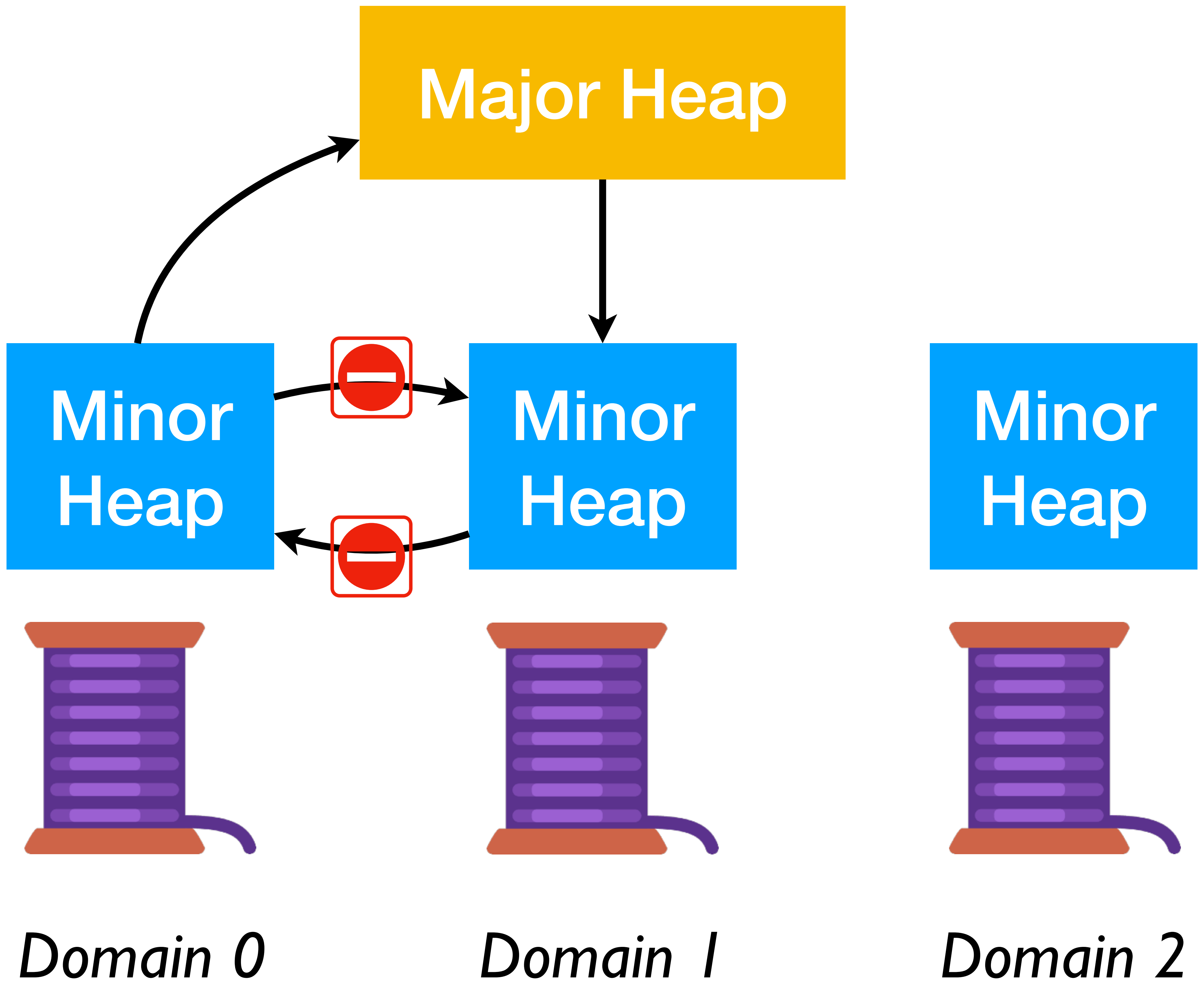
Stop-the-world GC

- Assume
  - GC overhead of 20%
  - Program is perfectly parallelizable
- On 1 core,
  - Mutator 80s + GC 20s = 100s
- On 8 cores,
  - Mutator 10s + GC 20s = 30s
  - Parallel Speedup = 100/30 = 3.3x on 8 cores
- On ∞ cores,
  - Mutator 0s + GC 20s = 20s
  - Parallel Speedup = 100/20 = 5x on ∞ cores

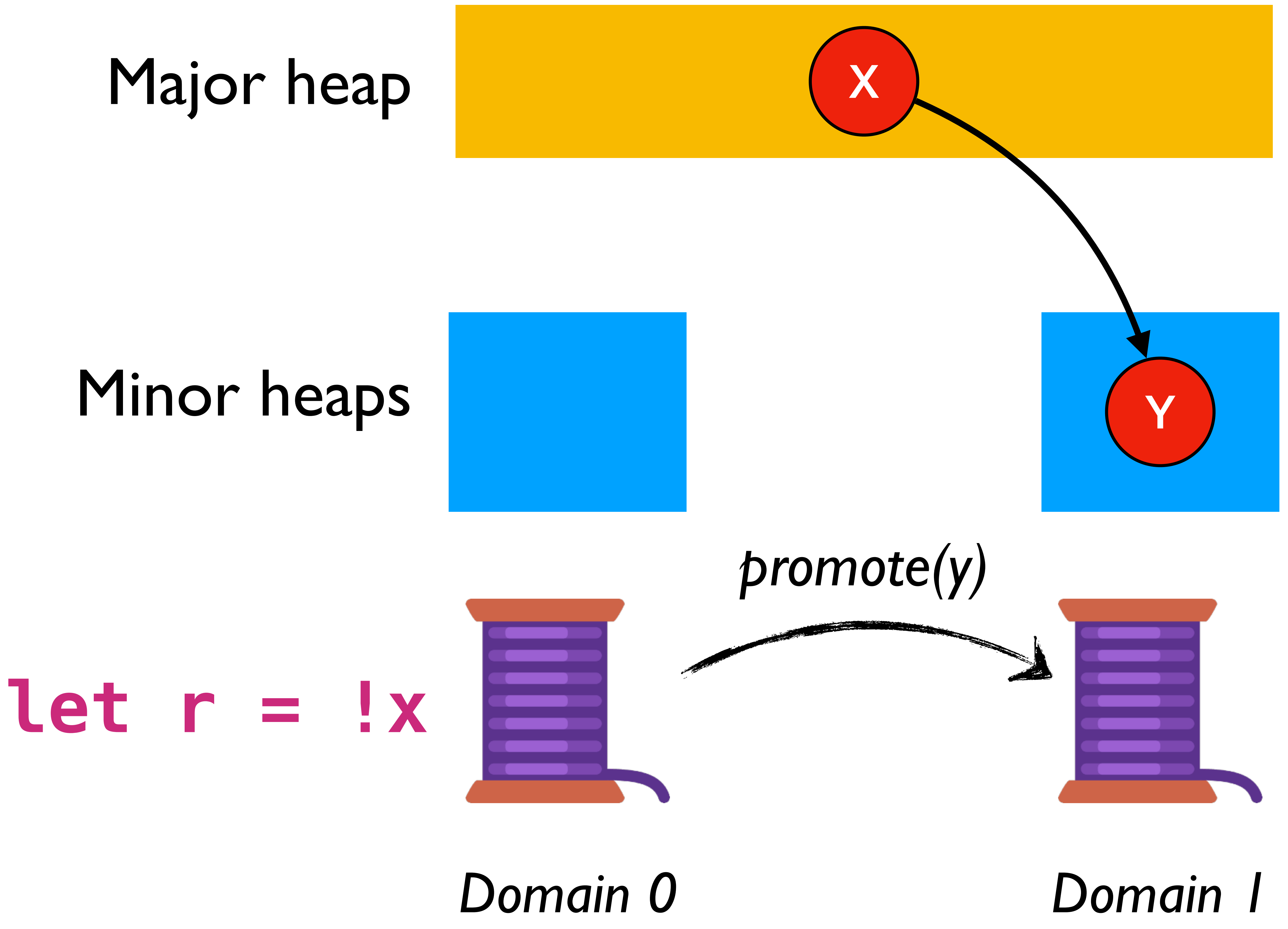


A concurrent minor GC

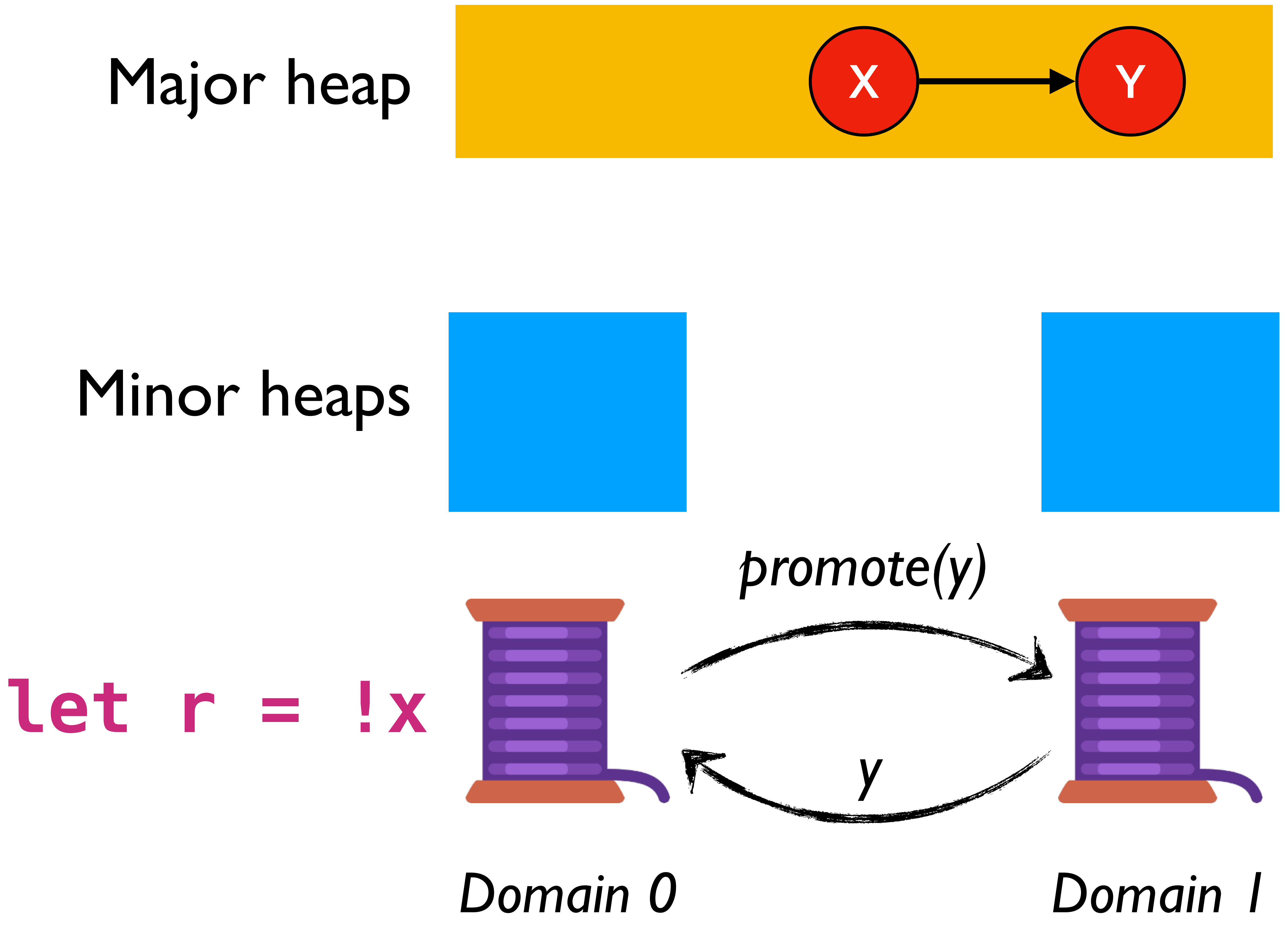
- Allow each domain's minor heap to be independently GCed



Promotion



Promotion



Concurrent Minor GC — Prior Art

A concurrent, generational garbage collector
for a multithreaded implementation of ML

Damien Doligez Xavier Leroy
École Normale Supérieure and INRIA Rocquencourt*

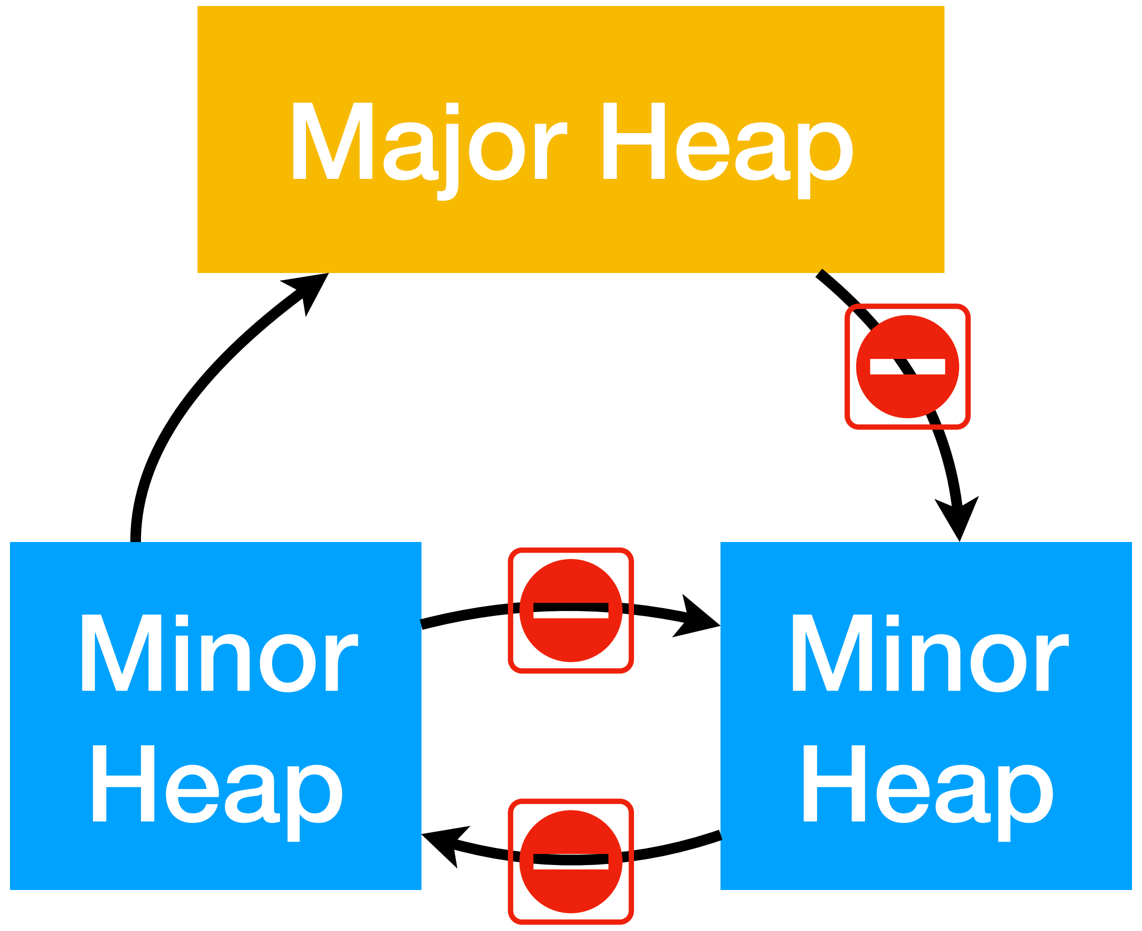
POPL '93

Abstract

This paper presents the design and implementation of a "quasi real-time" garbage collector for Concurrent Caml Light, an implementation of ML with threads. This two-generation system combines a fast, asyn-

the threads that execute the user's program, with as little synchronization as possible between the collector and the mutators (the threads executing the user's program).

A number of concurrent collectors have been de-



Concurrent Minor GC — Prior Art

Multicore Garbage Collection with Local Heaps

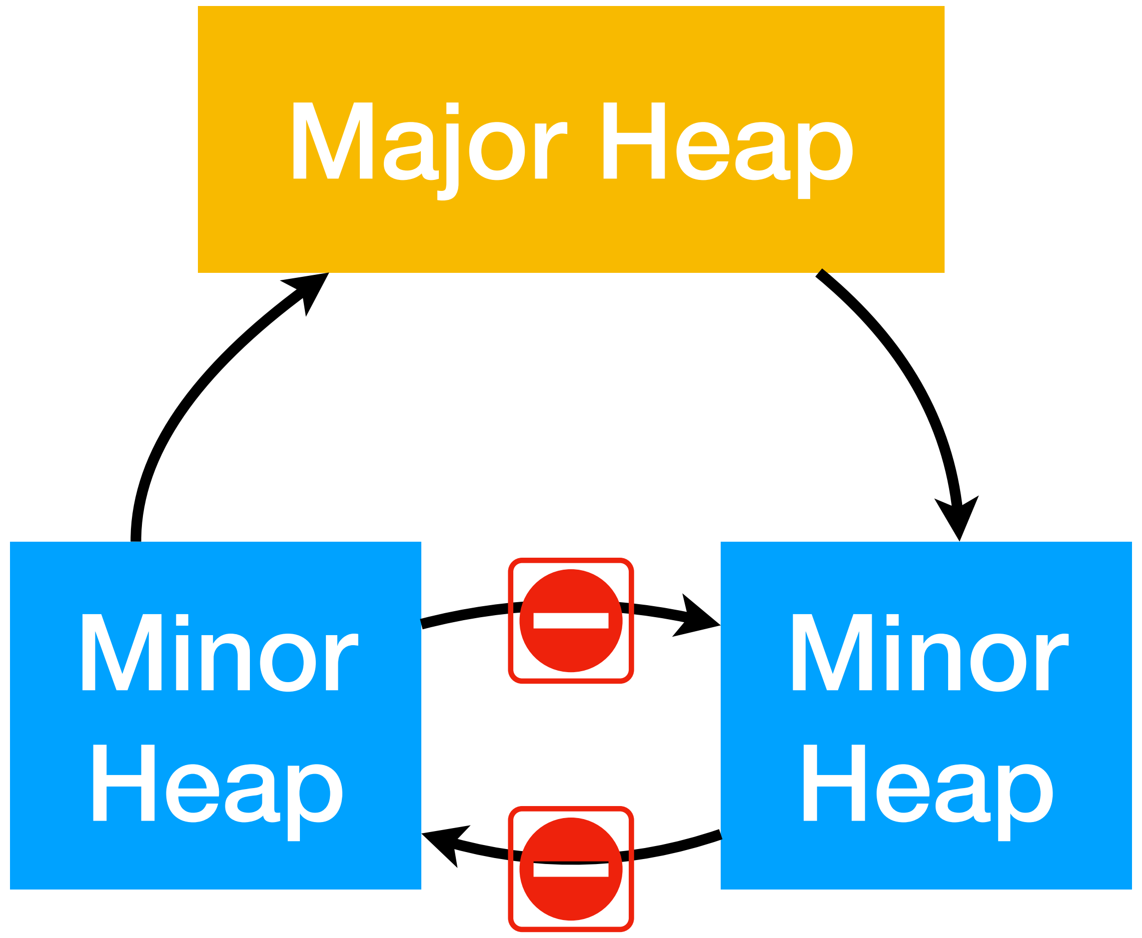
Simon Marlow
Microsoft Research, Cambridge, U.K.
simonmar@microsoft.com

Simon Peyton Jones
Microsoft Research, Cambridge, U.K.
simonpj@microsoft.com

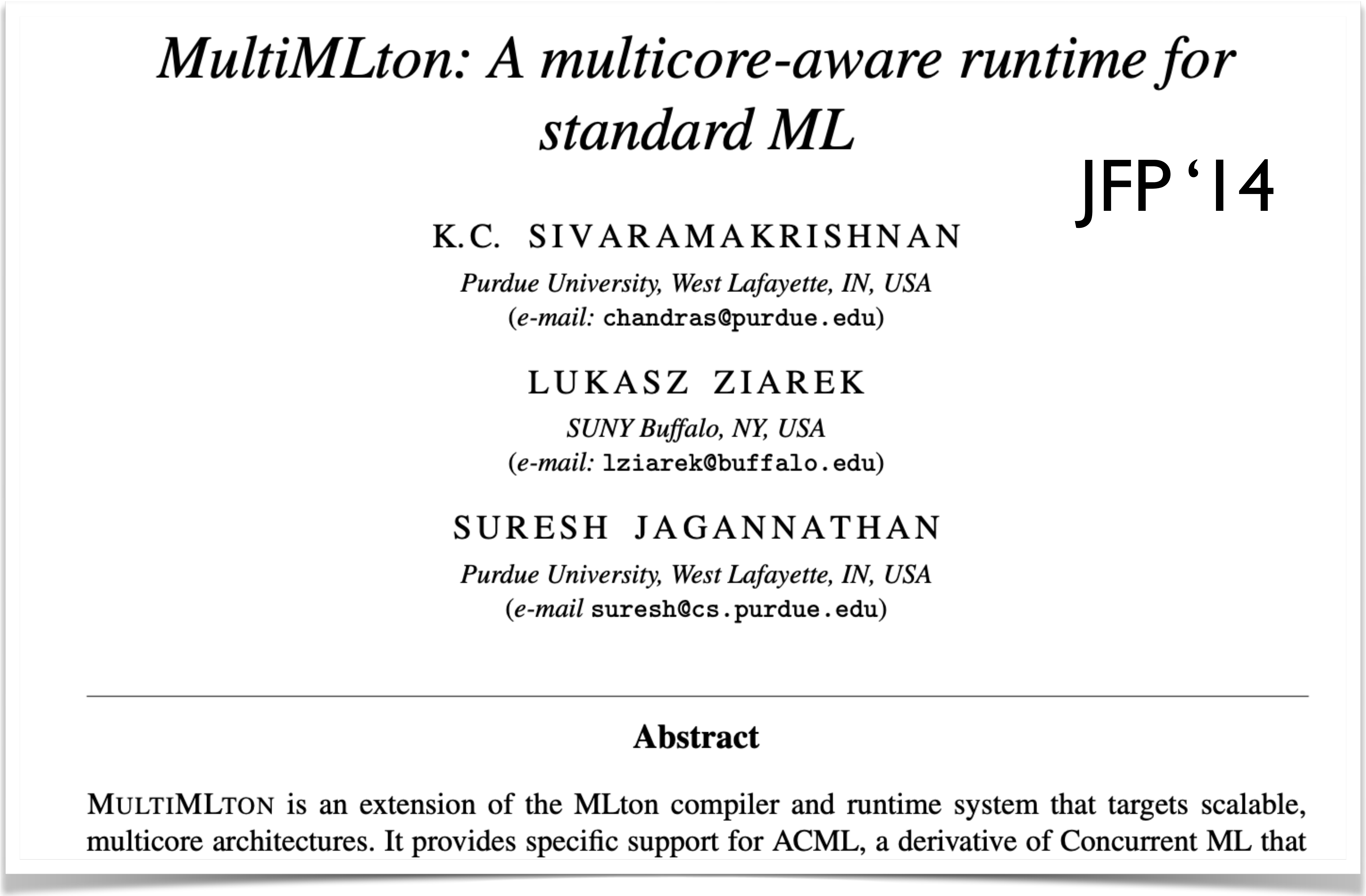
ISMM '11

Abstract

In a parallel, shared-memory, language with a garbage collected heap, it is desirable for each processor to perform minor garbage collections independently. Although obvious, it is difficult to make this idea pay off in practice, especially in languages where muta- to design collectors in which each processor has a private heap that can be collected independently without synchronising with the other processors; there is also a global heap for shared data. Some of the existing designs are based on static analyses to identify objects whose references never escape the current thread and



Concurrent Minor GC — Prior Art



MultiMLton: A multicore-aware runtime for standard ML

K.C. SIVARAMAKRISHNAN
Purdue University, West Lafayette, IN, USA
(*e-mail:* chandras@purdue.edu)

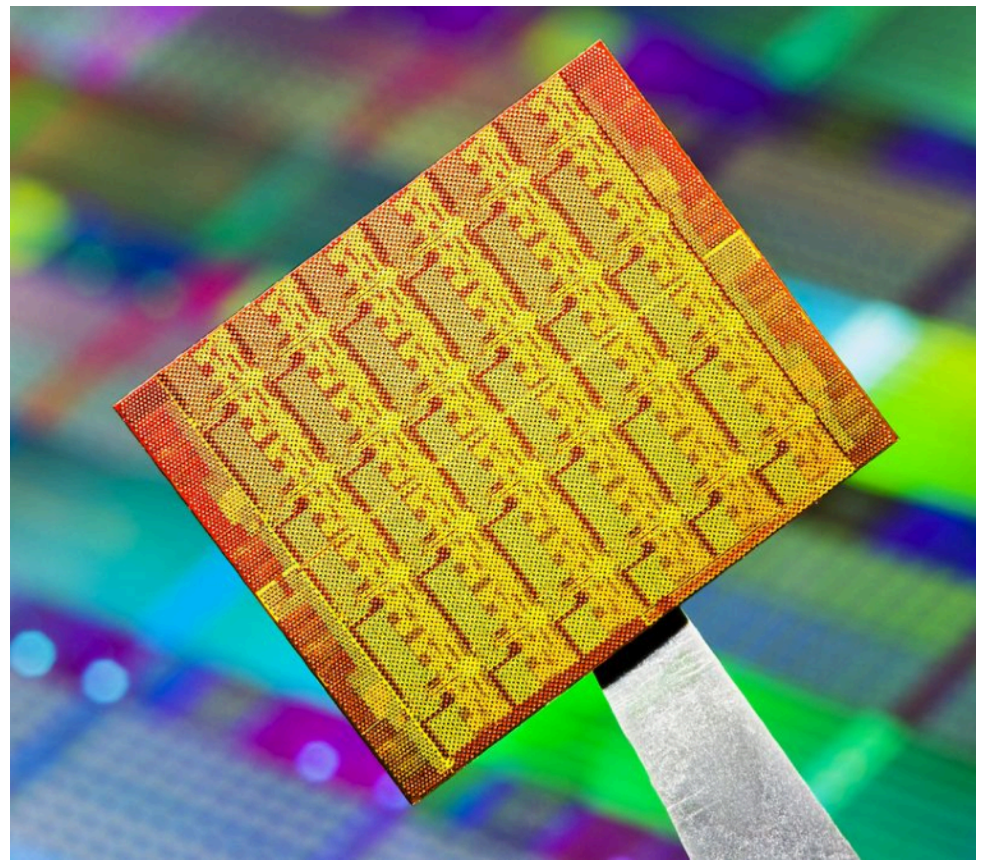
LUKASZ ZIAREK
SUNY Buffalo, NY, USA
(*e-mail:* lziarek@buffalo.edu)

SURESH JAGANNATHAN
Purdue University, West Lafayette, IN, USA
(*e-mail* suresh@cs.purdue.edu)

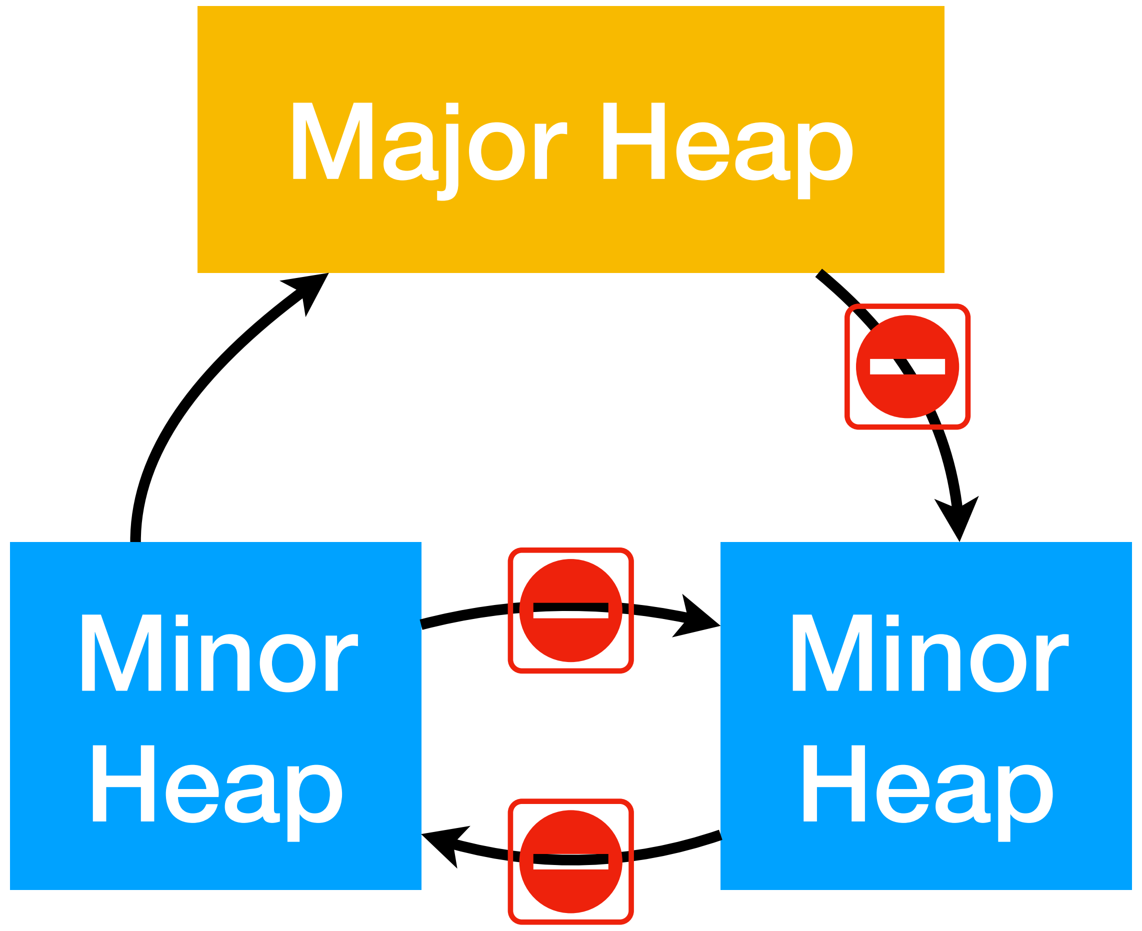
Abstract

MULTIMLTON is an extension of the MLton compiler and runtime system that targets scalable, multicore architectures. It provides specific support for ACML, a derivative of Concurrent ML that

JFP '14



Intel Single-chip Cloud Computer (SCC)



Concurrent Minor GC — Prior Art

Hierarchical Memory Management for Mutable State

Extended Technical Appendix

PPoPP '18

Adrien Guatto
Carnegie Mellon University
adrien@guatto.org

Sam Westrick
Carnegie Mellon University
swestric@cs.cmu.edu

Ram Raghunathan
Carnegie Mellon University
ram.r@cs.cmu.edu

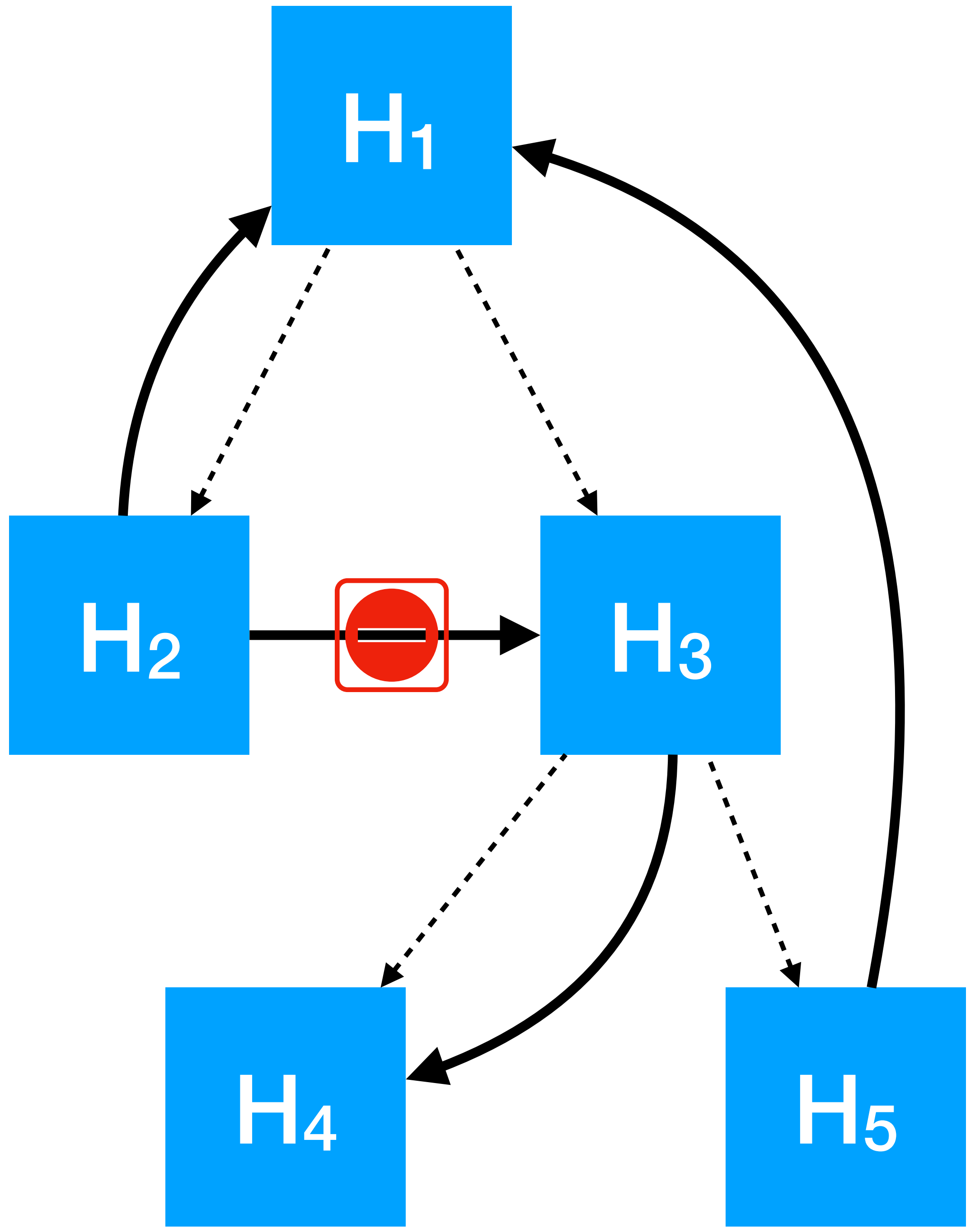
Umut Acar
Carnegie Mellon University
umut@cs.cmu.edu

Matthew Fluet
Rochester Institute of Technology
mtf@cs.rit.edu

Abstract

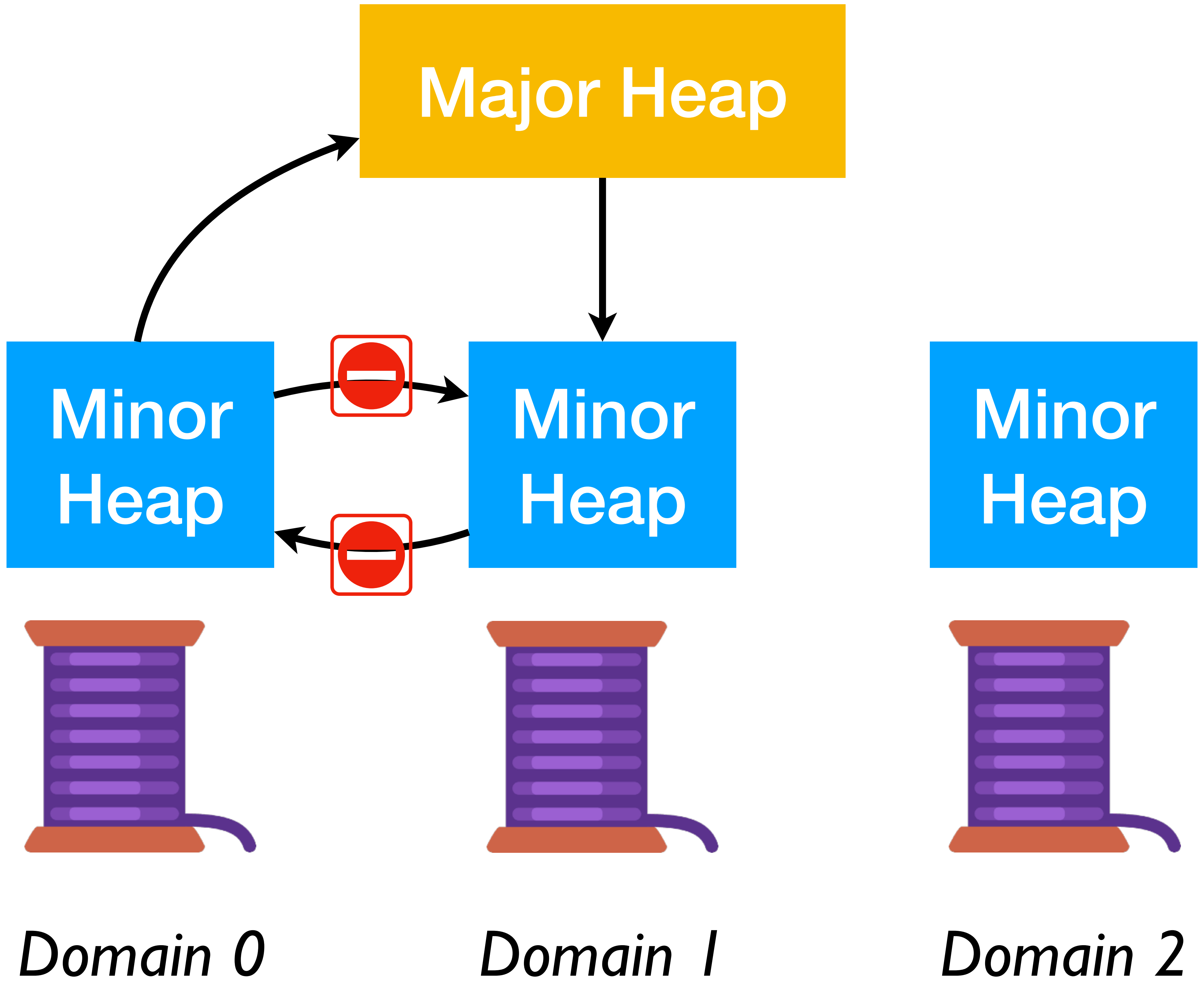
It is well known that modern functional programming languages are naturally amenable to parallel programming. strongly typed functional languages is their ability to distinguish between pure and impure code. This aids in writing correct parallel programs by making it easier to avoid race

MaPLe

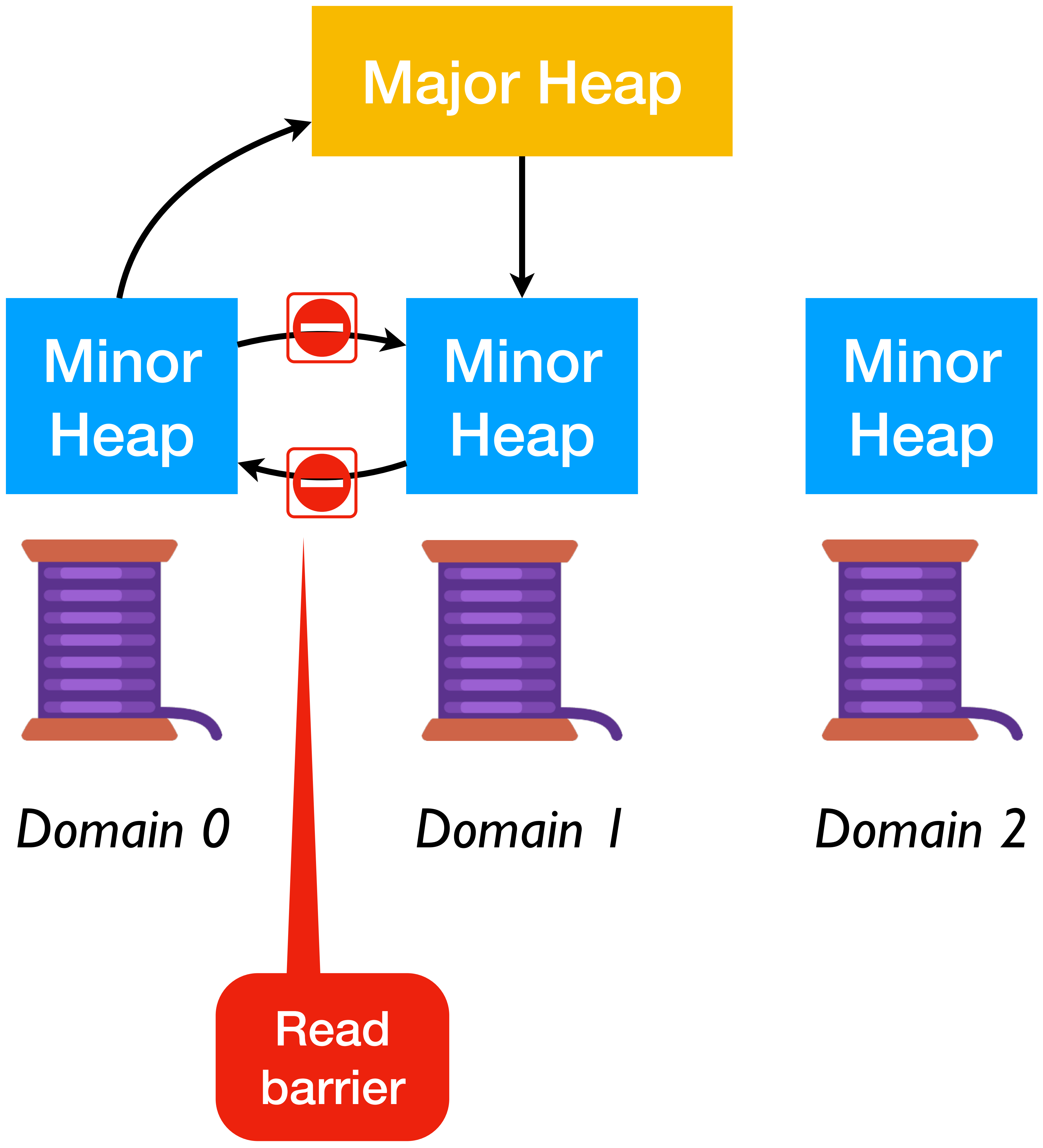


disentanglement

A concurrent minor GC



A concurrent minor GC



- OCaml does not have read barriers
- A new branch on reads in OCaml
  - *Cheap and fast*
- ***Read is now GC safe point***
  - OCaml C FFI makes assumptions about when GC can run
  - In OCaml 4, GC cannot run at ***field reads***

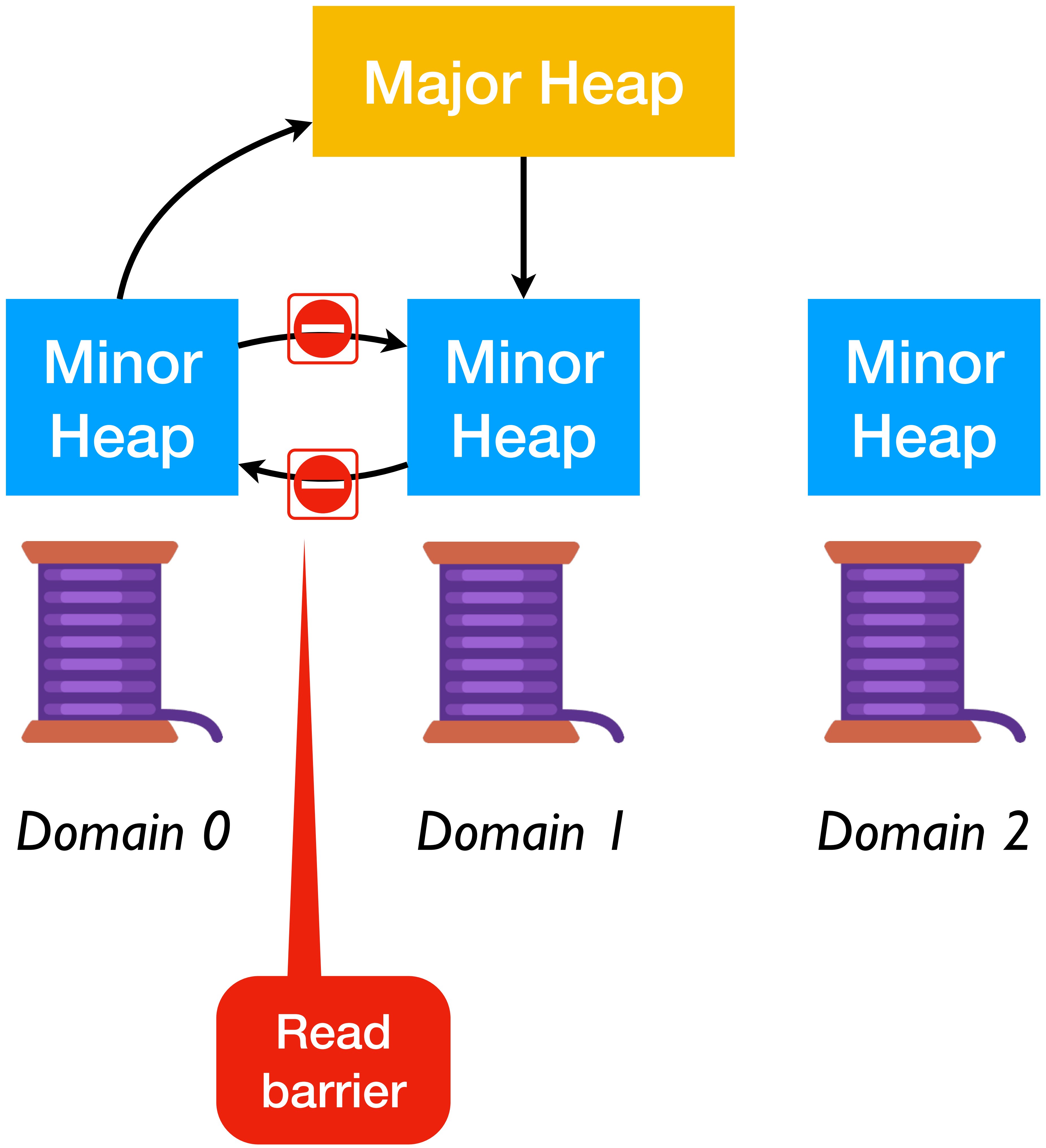
C FFI — Breaking GC Invariant

```c
value caml_test2(value v)
{
  CAMLparam1(v); // Register the parameter as a GC root
  CAMLlocal1(result); // Register the local variable as a GC root
  result = caml_alloc_2 (Tag_0, Field(v, 0), Field(v, 1)); //BUG!
  CAMLreturn(result);
}
```

C FFI — Breaking GC Invariant

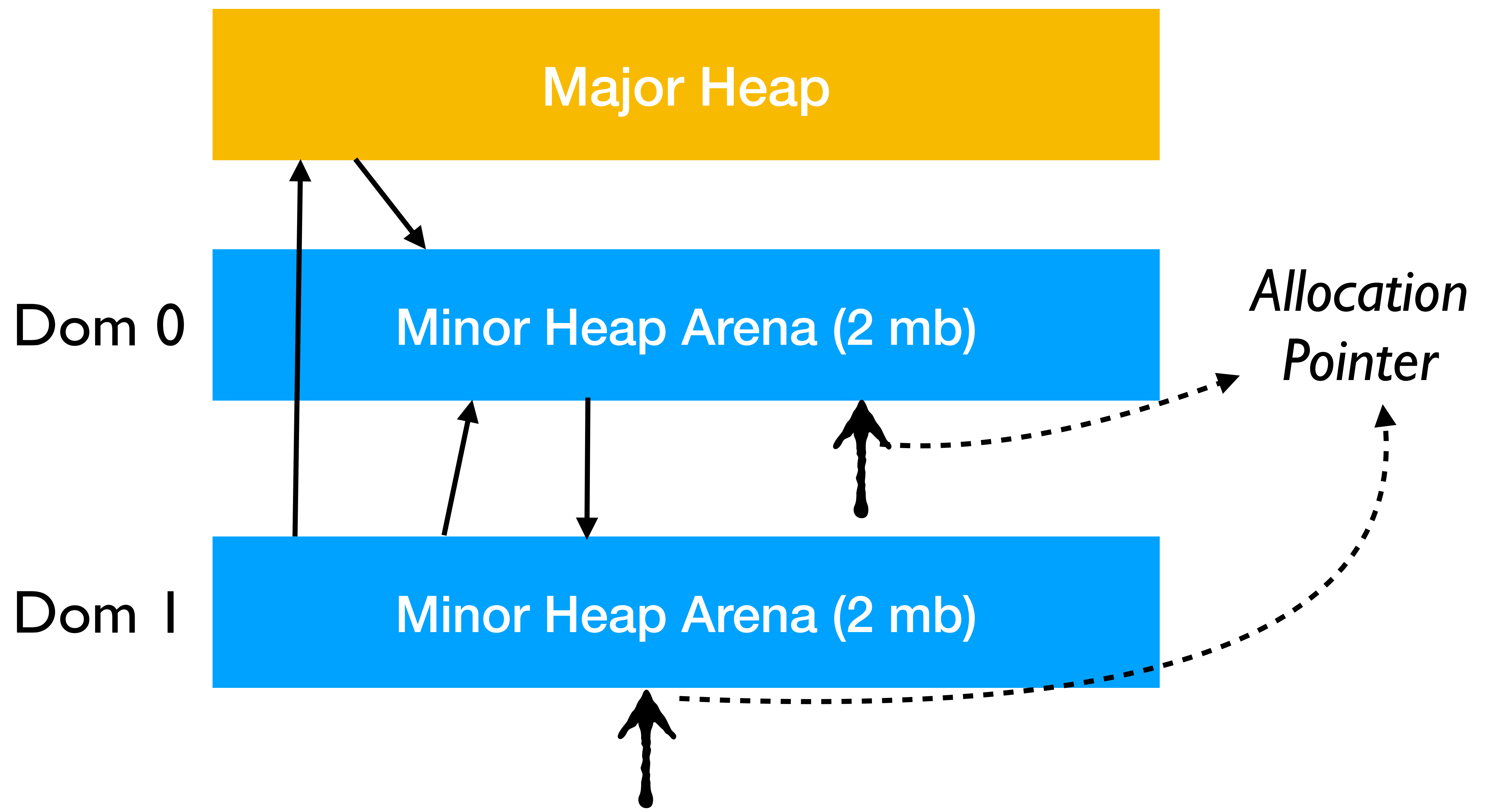
```
value caml_test2(value v)
{
  CAMLparam1(v); // Register the parameter as a GC root
  CAMLlocal1(result); // Register the local variable as a GC root
  value r1 = Field(v, 0);
  value r2 = Field(v, 1); //GC can occur here and move object at [r1]
  result = caml_alloc_2 (Tag_0, r1, r2);
  CAMLreturn(result);
}
```

A concurrent minor GC



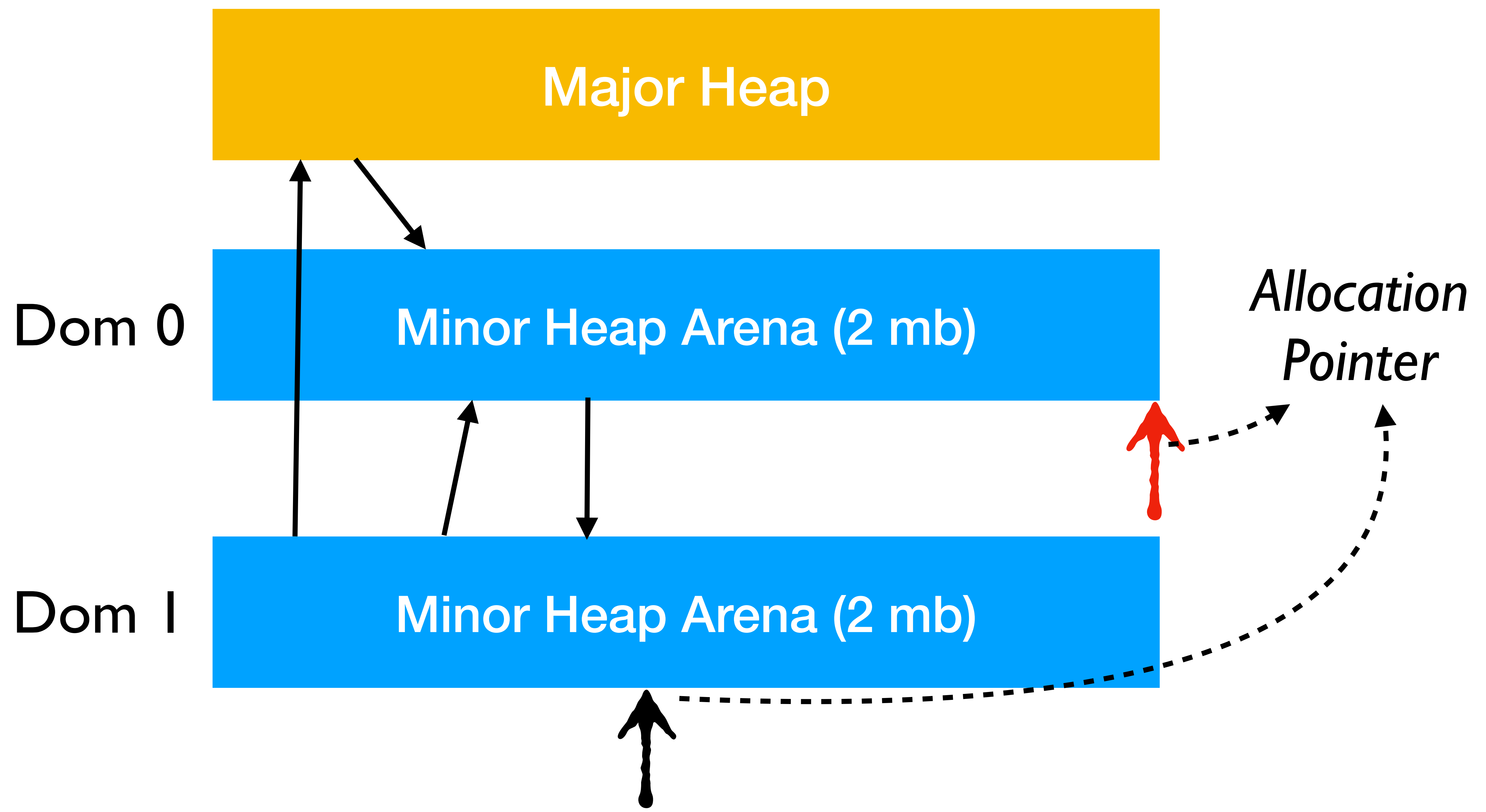
- OCaml does not have read barriers
- A new branch on reads in OCaml
  - *Cheap and fast*
- ***Read is now GC safe point***
  - OCaml C FFI makes assumptions about when GC can run
  - In OCaml 4, GC cannot run at ***field reads***
- ***GC invariants are broken by this design :-(***

Take 2 — A parallel minor GC



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

Take 2 — A parallel minor GC



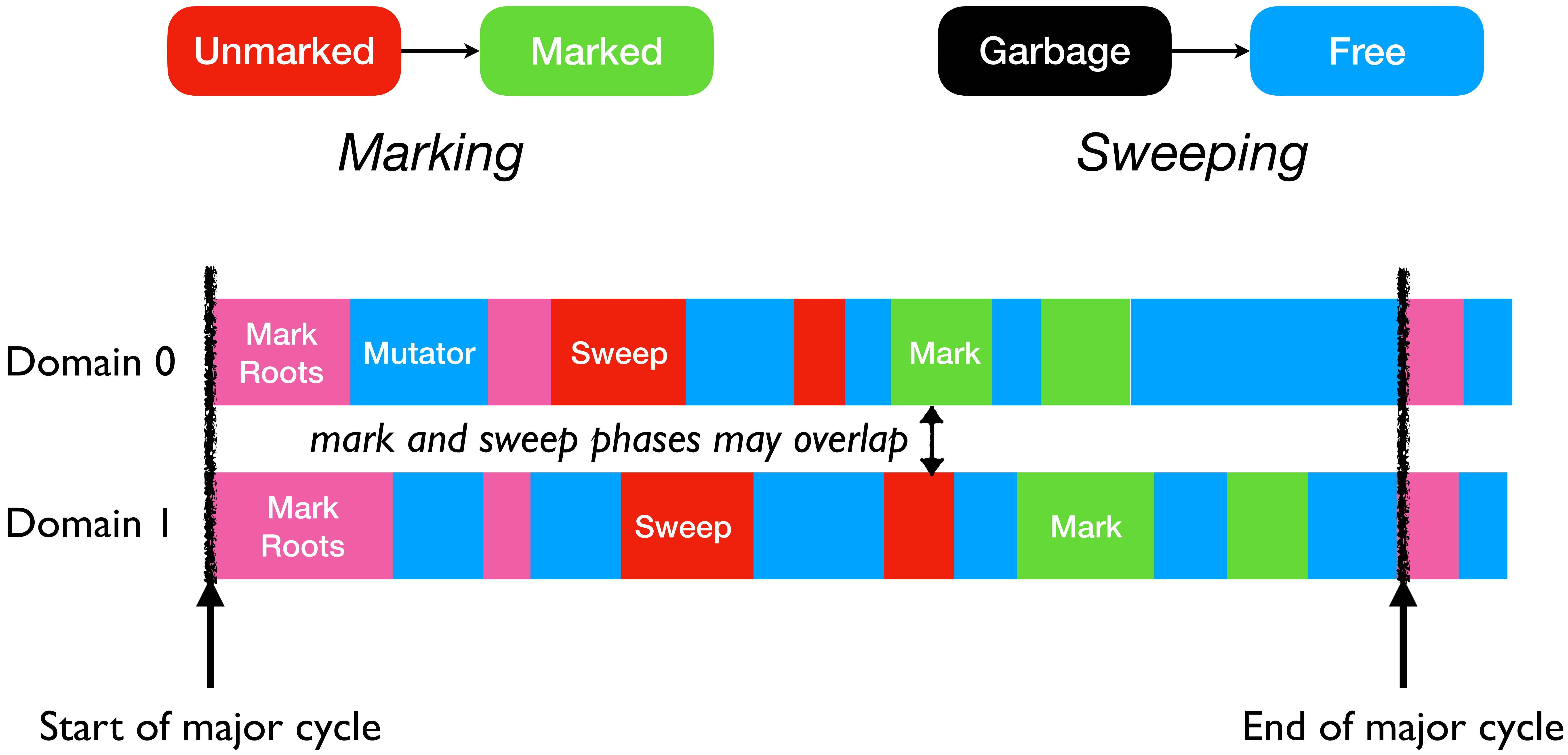
- *Stop-the-world* *parallel* collection for minor heaps
  - 2 barriers / minor gc; (some) work sharing between gc threads
- On 24 cores, w/ default heap size (2MB / arena), ***< 5 ms*** pause for completing minor GC

Multicore Allocator

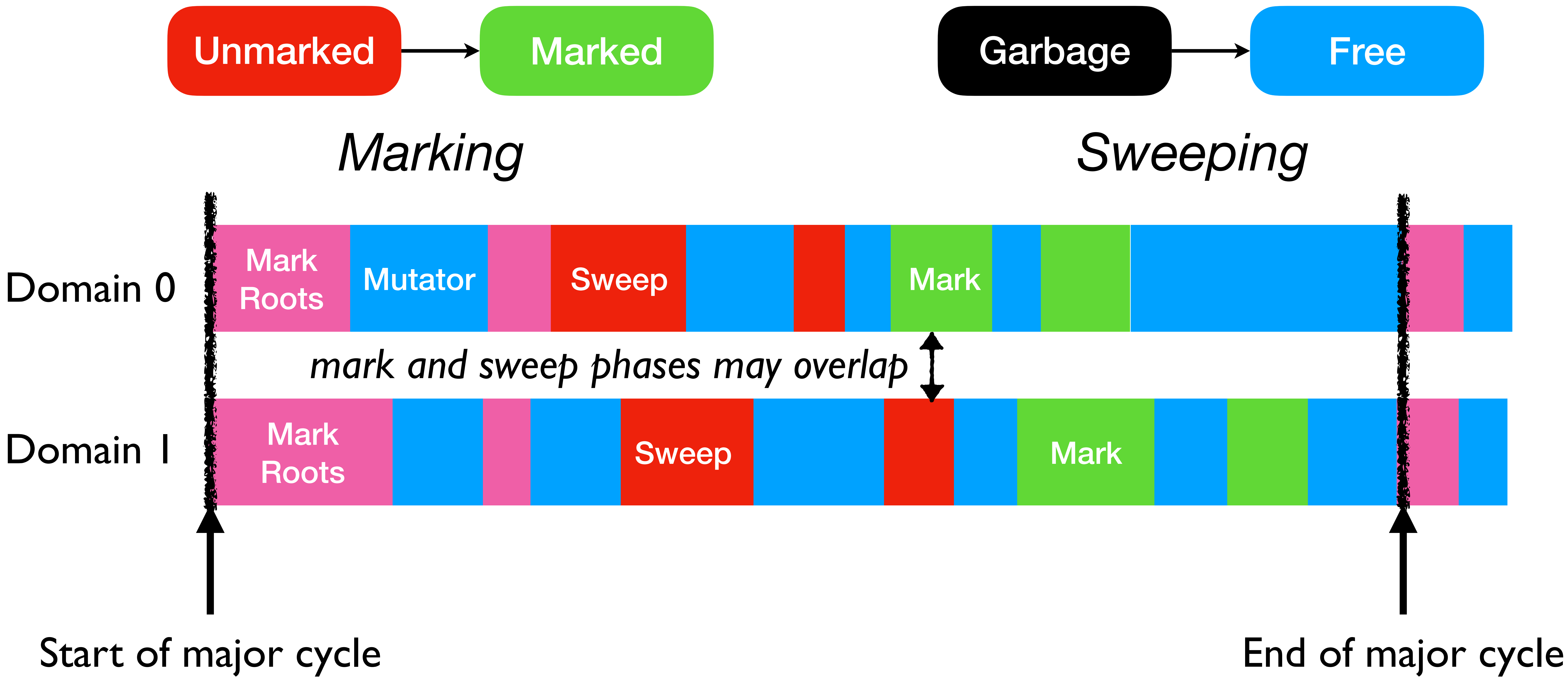
- Multicore-aware allocator must avoid synchronisation for allocations
- Major heap uses size-segmented, thread-local, page-sized pools
  - Minor heap is domain-local and bump pointer (no synchronisation)
- Pool size distribution
  - More pools for small sizes; exponentially decreases
  - Most allocations in OCaml are small (99% of objects < 5 words in size)
  - Malloc for large allocations
- Global pool of unused pages
  - Take away: most allocations don’t need synchronization

OCaml 5 major GC

- Mostly concurrent mark-and-sweep GC

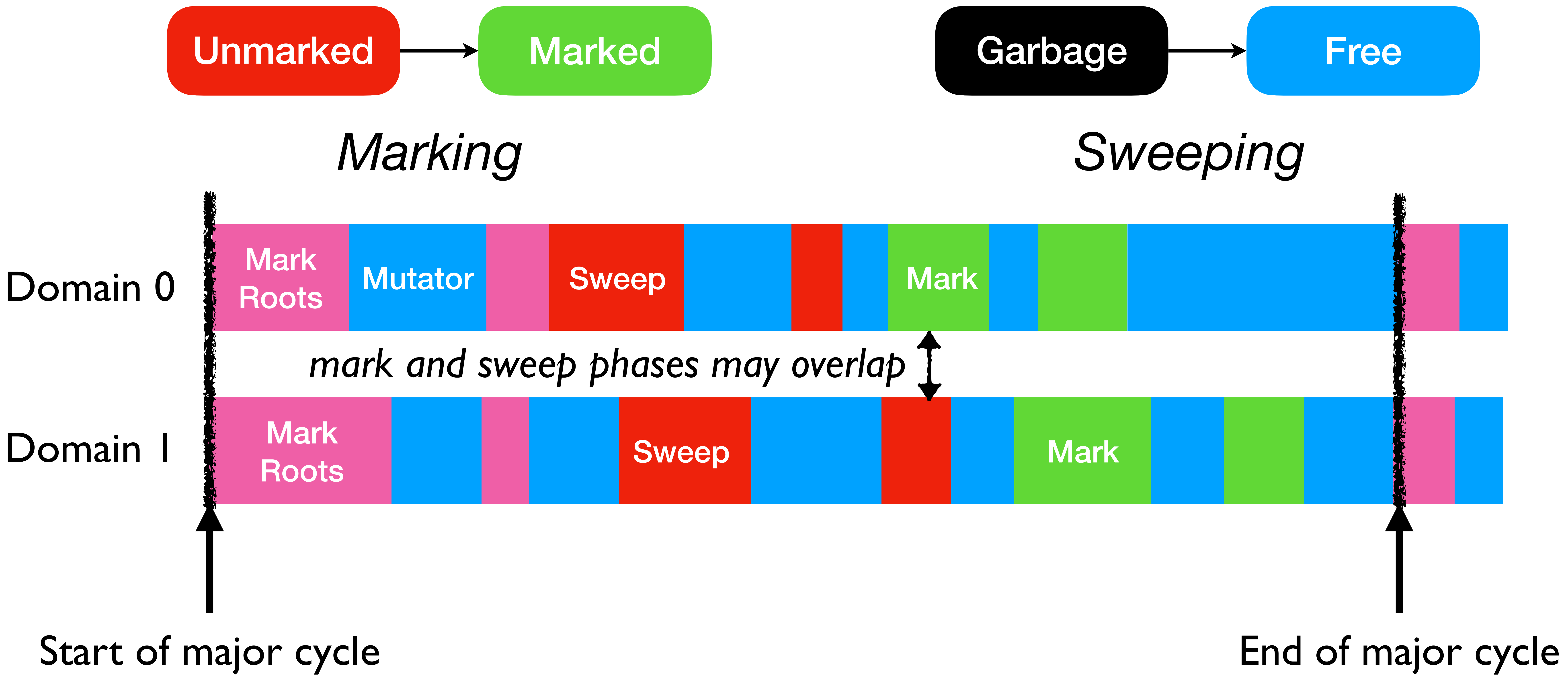


OCaml 5 major GC



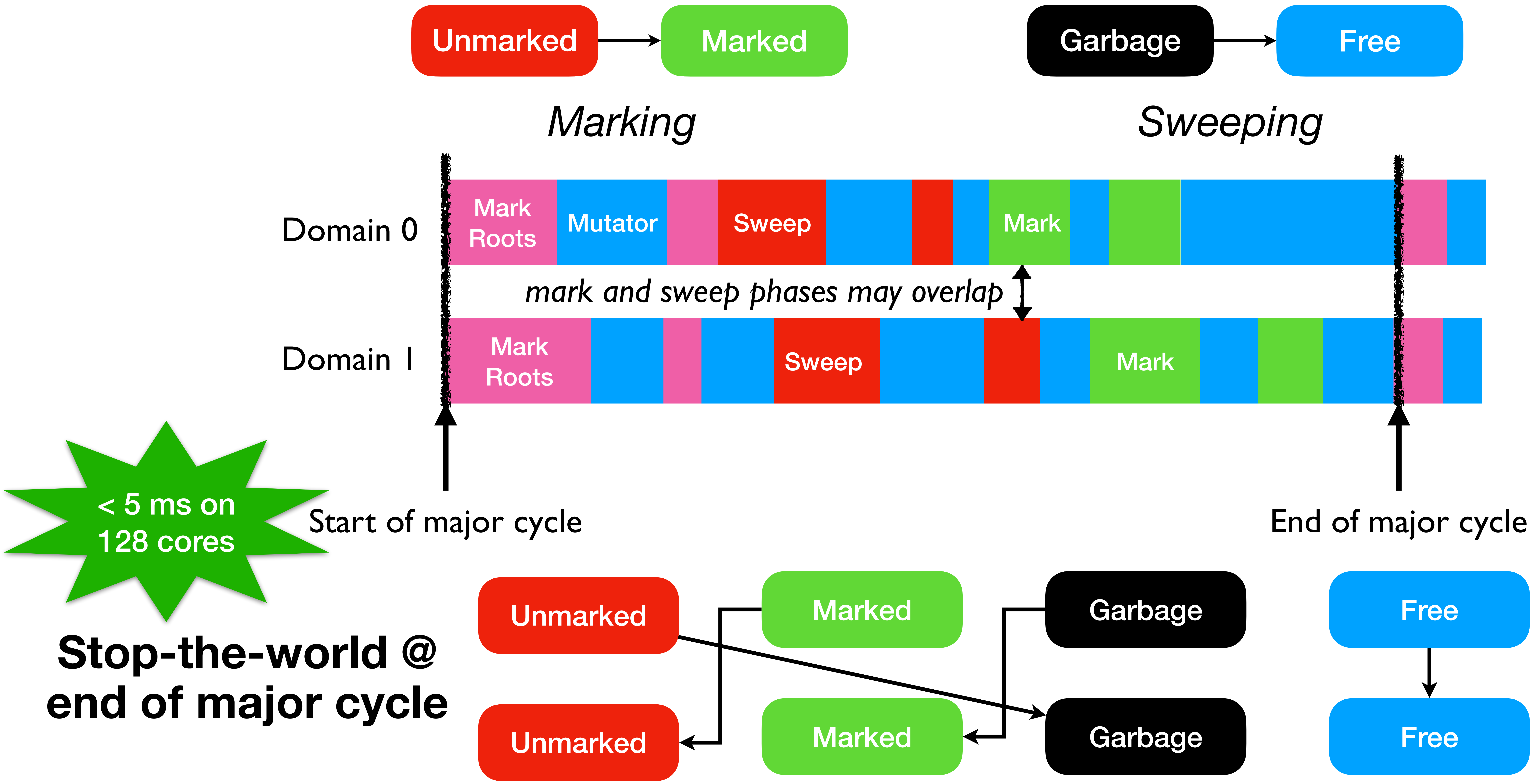
- How do we know when sweeping (per domain) is done?
  - When a domain has finished sweeping its own pools

OCaml 5 major GC



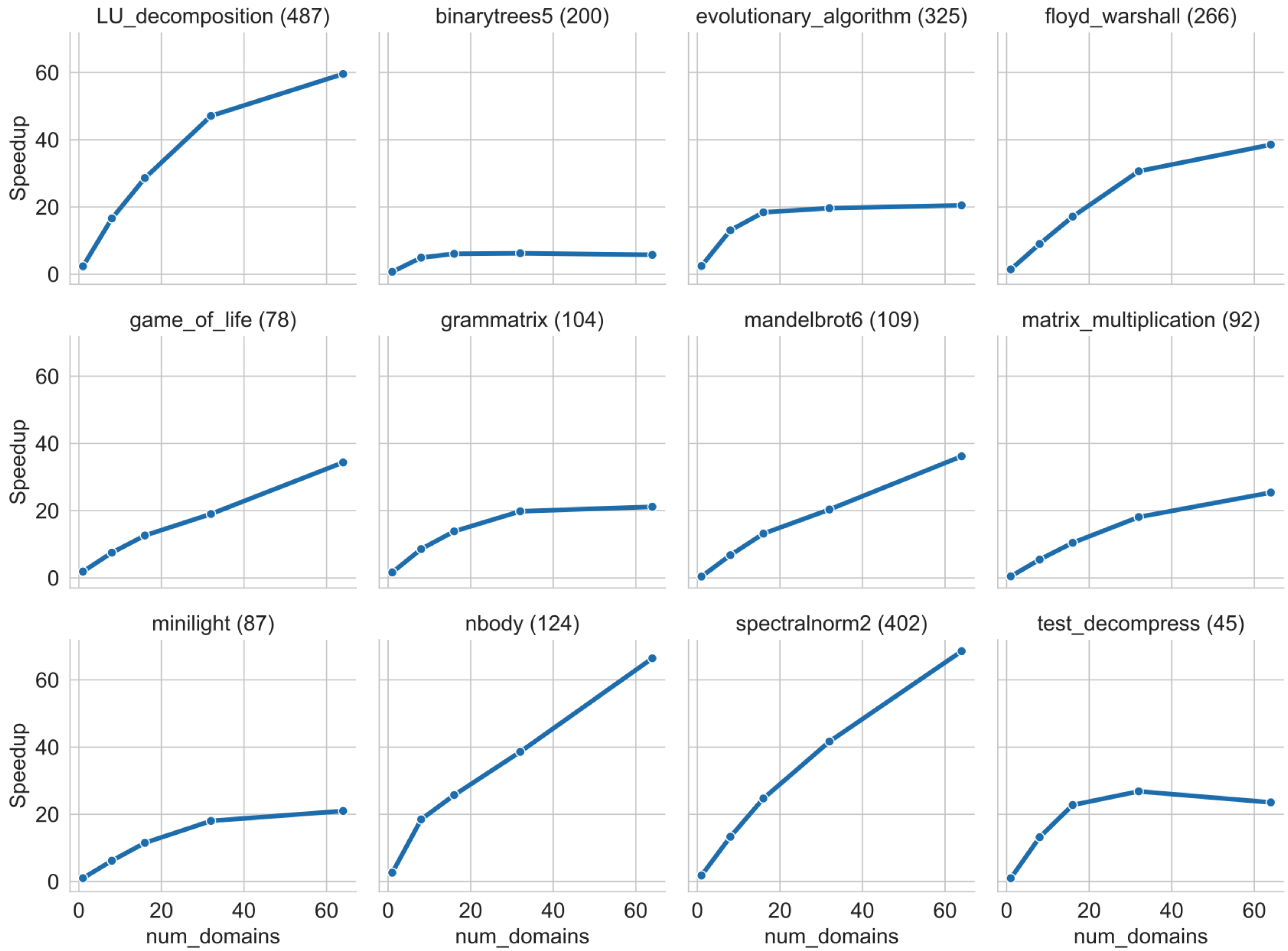
- How do we know when marking is done?
  - When every domain’s mark stack is empty (thanks, SATB GC!)

OCaml 5 major GC



Stop-the-world @ end of major cycle

Performance Results

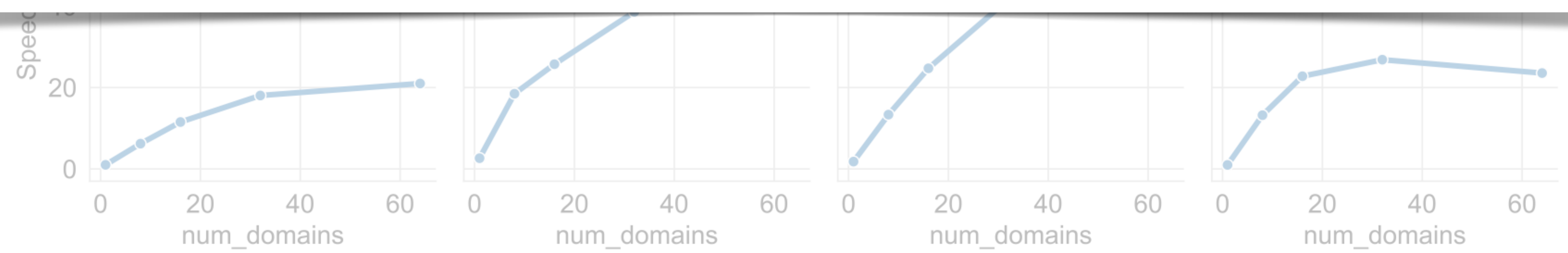


Retrofitting Parallelism onto OCaml

KC SIVARAMAKRISHNAN, IIT Madras, India
STEPHEN DOLAN, OCaml Labs, UK
LEO WHITE, Jane Street, UK
SADIQ JAFFER, Opsian, UK and OCaml Labs, UK
TOM KELLY, OCaml Labs, UK
ANMOL SAHOO, IIT Madras, India
SUDHA PARIMALA, IIT Madras, India
ATUL DHIMAN, IIT Madras, India
ANIL MADHAVAPEDDY, University of Cambridge Computer Laboratory, UK and OCaml Labs, UK

ICFP 2020

OCaml is an industrial-strength, multi-paradigm programming language, widely used in industry and academia. OCaml is also one of the few modern managed system programming languages to lack support for shared memory parallel programming. This paper describes the design, a full-fledged implementation and evaluation



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
  - Eio, a new OCaml 5 concurrency library
  - 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 IPC logic
  - No function colouring!
    - Reclaim the use of `try…with`, `for` and `while` loops!
- Used TSan to ensure that ***data races*** are removed

Data races

- When two threads access the same memory location
    - Without synchronization
    - One of them is a write
- Data races are programming errors
    - Leads to *undefined behaviour* in C and C++
- OCaml programs with data races remain *well-typed*
    - May observe non-sequentially-consistent behaviour

```
let a = ref 0 and b = ref 0

let d1 () =
  a := 1;
  !b

let d2 () =
  b := 1;
  !a

let () =
  let h = Domain.spawn d2 in
  let r1 = d1 () in
  let r2 = Domain.join h in
  assert (not (r1 = 0 && r2 = 0))
```

Bounding Data Races in Space and Time

(Extended version, with appendices)

PLDI 2018

Stephen Dolan
University of Cambridge, UK

KC Sivaramakrishnan
University of Cambridge, UK

Anil Madhavapeddy
University of Cambridge, UK

Abstract

We propose a new semantics for shared-memory parallel programs that gives strong guarantees even in the presence of data races. Our *local data race freedom* property guarantees that all data-race-free portions of programs exhibit sequential semantics. We provide a straightforward operational semantics and an equivalent axiomatic model, and evaluate an implementation for the OCaml programming language. Our evaluation demonstrates that it is possible to

The primary reasoning tools provided to programmers by these models are the *data-race-freedom (DRF) theorems*. Programmers are required to mark as *atomic* all variables used for synchronisation between threads, and to avoid *data races*, which are concurrent accesses (except concurrent reads) to nonatomic variables. In return, the DRF theorems guarantee that no relaxed behaviour will be observed. Concisely, *data-race-free programs have sequential semantics*.

When programs are not data-race-free, such models give

ThreadSanitizer (since 5.2)

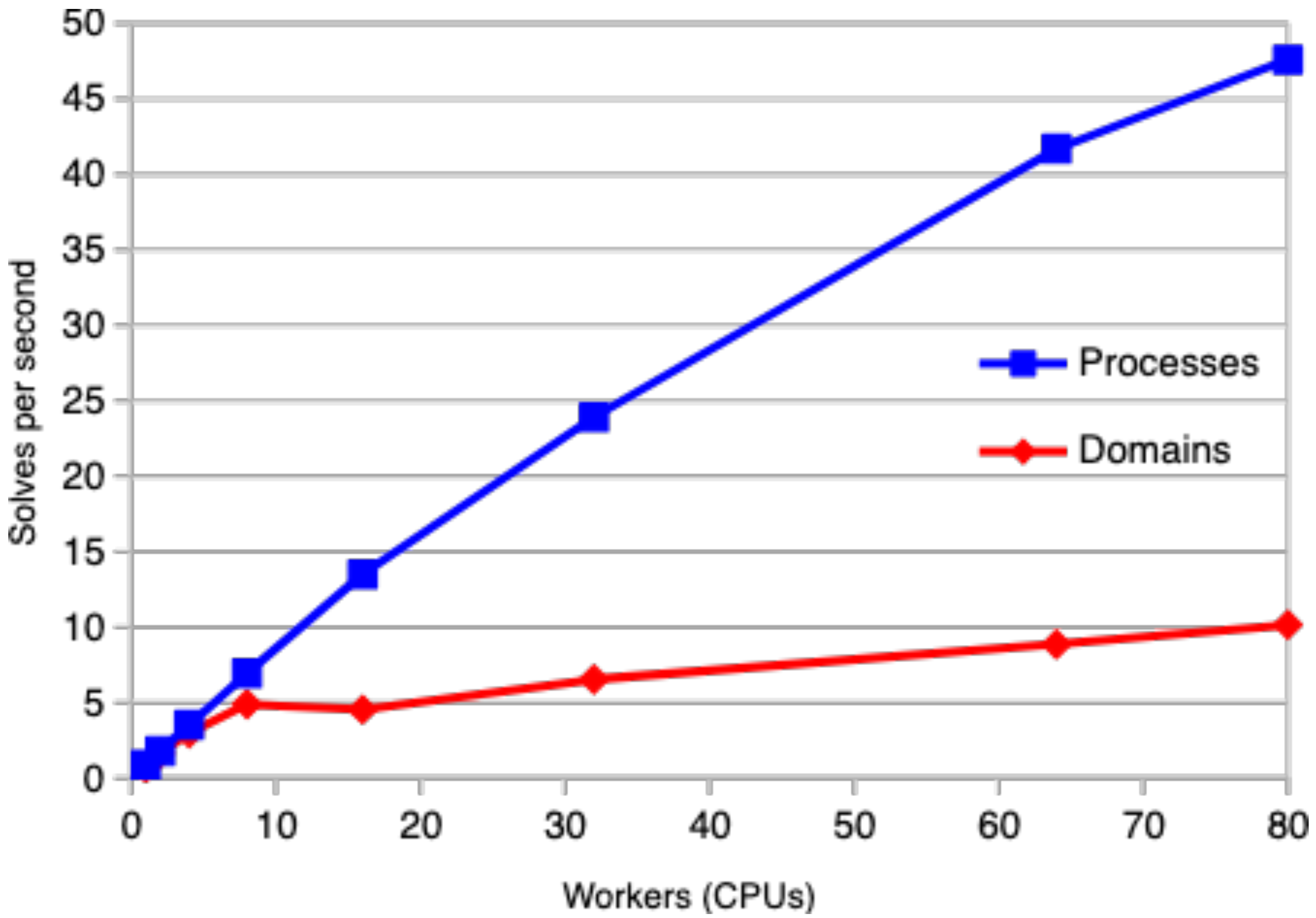
- Detect data races dynamically
- Part of the LLVM project — C++, Go, Swift

```
let a = ref 0 and b = ref 0

let d1 () =
  a := 1;
  !b

let d2 () =
  b := 1;
  !a

let () =
  let h = Domain.spawn d2 in
  let r1 = d1 () in
  let r2 = Domain.join h in
  assert (not (r1 = 0 && r2 = 0))
```

```
==================
WARNING: ThreadSanitizer: data race (pid=3808831)
  Write of size 8 at 0x8febe0 by thread T1 (mutexes: write M90)
    #0 camlSimple_race.d2_274 simple_race.ml:8 (simple_race.exe+
    #1 camlDomain.body_706 stdlib/domain.ml:211 (simple_race.exe
    #2 caml_start_program <null> (simple_race.exe+0x47cf37)
    #3 caml_callback_exn runtime/callback.c:197 (simple_race.exe
    #4 domain_thread_func runtime/domain.c:1167 (simple_race.exe

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

Eio solver service performance

- … was underwhelming ….initially

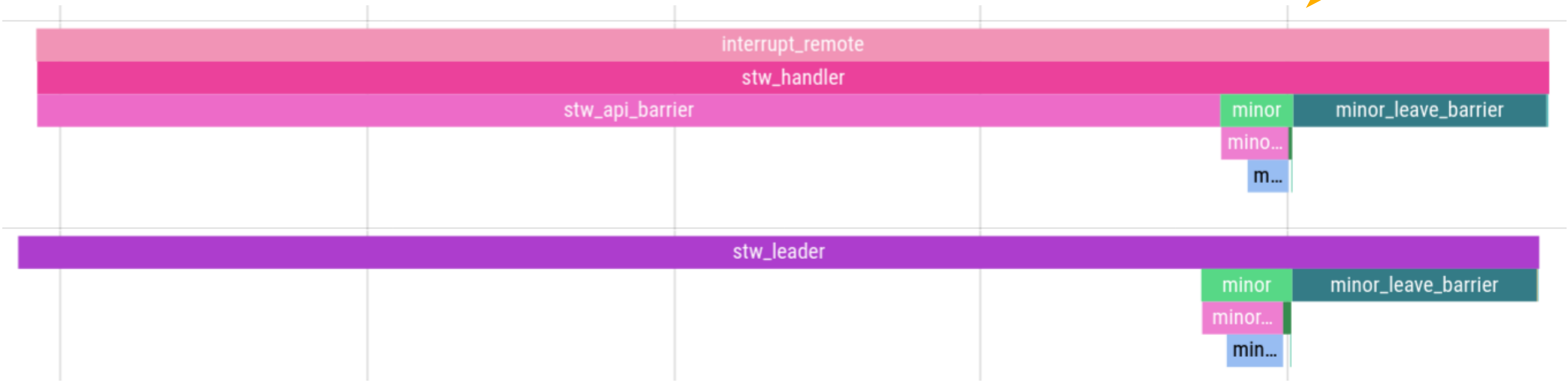


Performance analysis

- 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

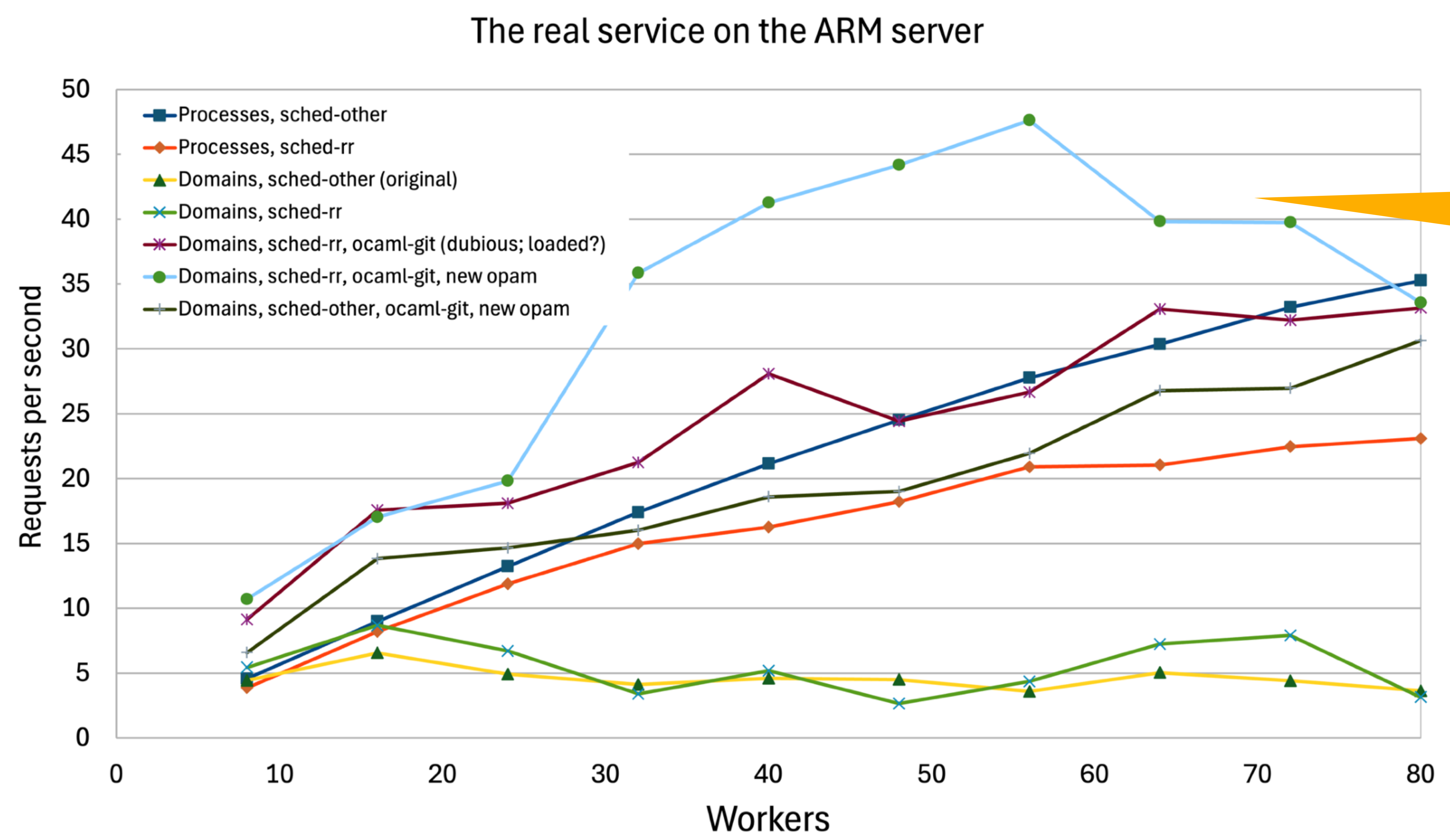
```
$ olly trace foo.trace foo.exe
```

https://perfetto.dev/



Problem indentified

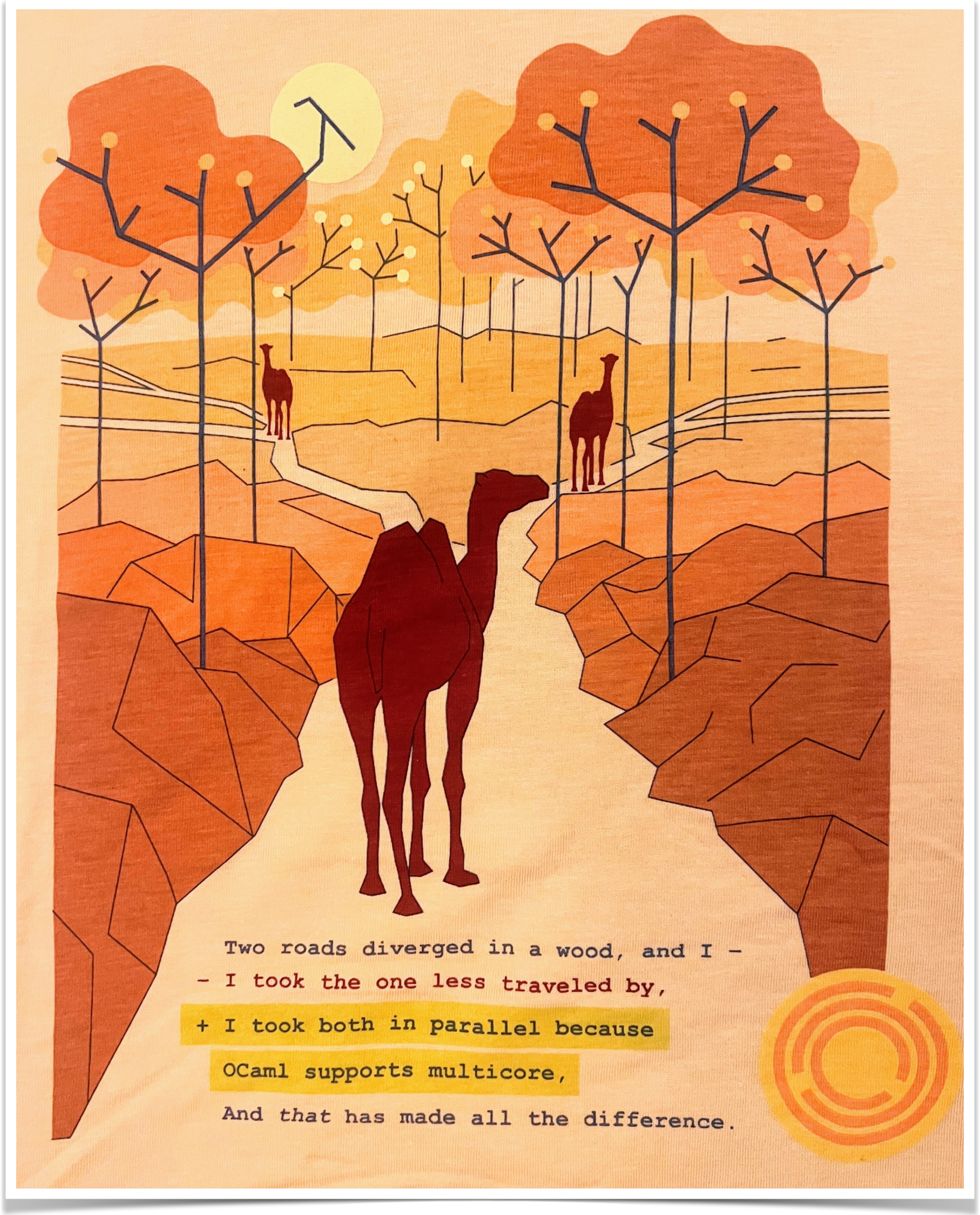
- Switch from `sched_other` to `sched_rr`
- `git log` for each solve to find earliest commit
  - 50ms penalty for STW subprocess spawn
  - Avoid by implementing it in OCaml



Still some work to do

Explore OCaml 5

- 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.



OCaml 6?

Oxidizing OCaml with Modal Memory Management

Data Rac

AÏNA LINN C
BENJAMIN P
LAILA ELBEH
LEO WHITE,
STEPHEN DO
RICHARD A.
CHRIS CASIN
FRANÇOIS P
DEREK DREY

We present DRF
threaded OCaml

A Mechanically Verified Garbage Collector for OCaml

JAR 2025

Sheera Shamsu[1], Dipesh Kafle[2], Dhruv Maroo[1], Kartik Nagar[1], Karthikeyan Bhargavan[3], KC Sivaramakrishnan[4]

[1]IIT Madras, Chennai, 600036, India.
[2]NIT Trichy, Trichy, 620015, India.
[3]Inria, Paris, 75014, France.
[4]Tarides and IIT Madras, Chennai, 600036, India.

A 2025

ational effects.
n, but using a traditional effect system would require adding extensive effect annotations to the millions of lines of existing code in these languages. Recent proposals seek to address this problem by removing the need for explicit effect polymorphism. However, they typically rely on fragile syntactic mechanisms or on introducing a separate notion of second-class function. We introduce a novel semantic approach based on modal effect types.