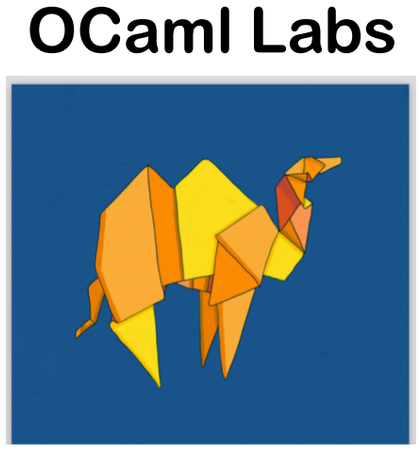
Retrofitting Parallelism onto OCaml

KC Sivaramakrishnan, Stephen Dolan, Leo white, Sadiq Jaffer, Tom Kelly, Anmol Sahoo, Sudha Parimala, Atul Dhiman, Anil Madhavapeddy



Industry

ahrefs

Jane Street

docker

FACEBOOK

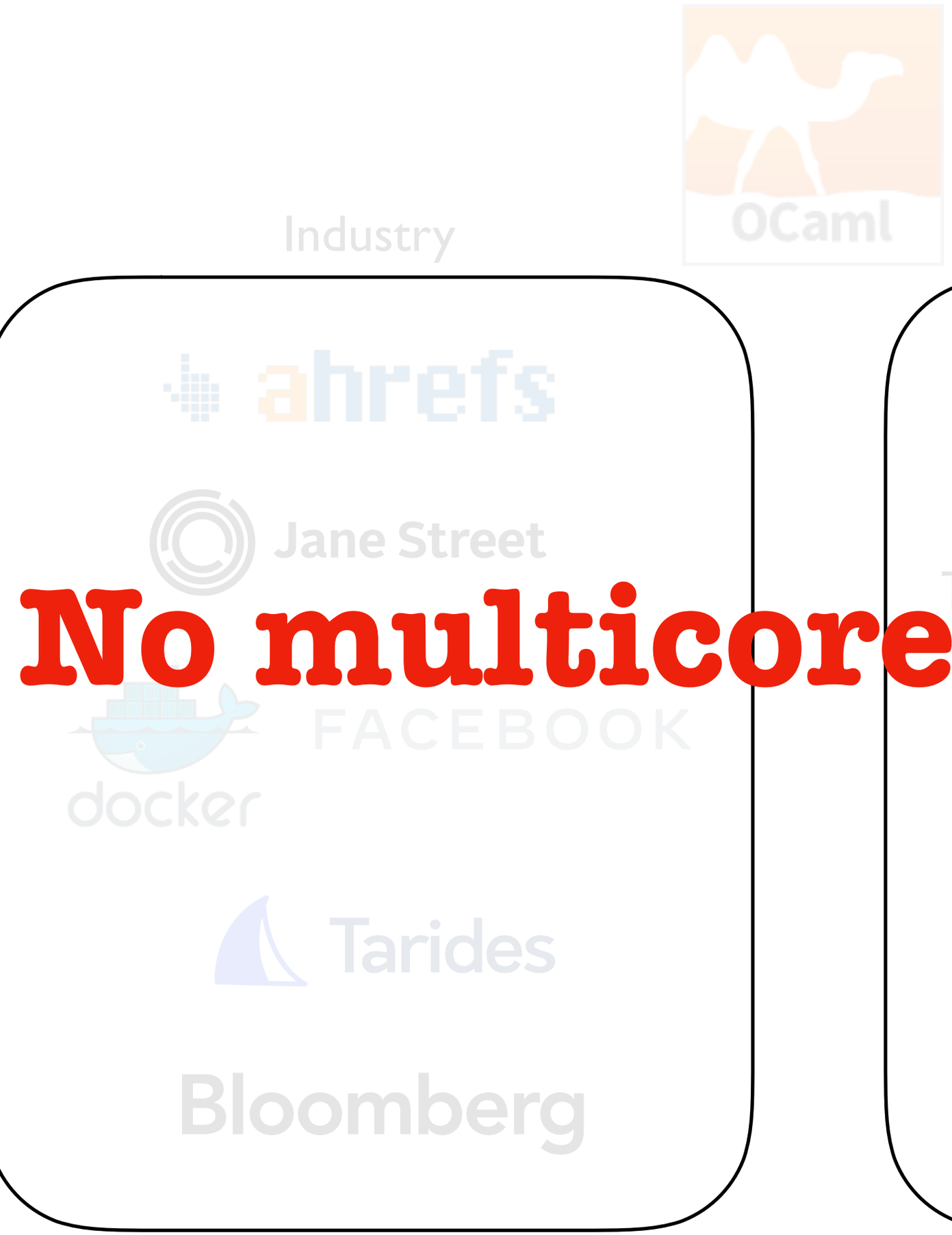
Tarides

Bloomberg

Projects

The **Astrée** Static Analyzer

COMPCERT


OCaml
Industry
Projects
ahrefs
Jane Street
FACEBOOK
docker
Tarides
Bloomberg
The Astrée Static Analyzer
flow
hack
COMPCERT
No multicore support!

Multicore OCaml

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

Multicore OCaml

- Adds native support for *concurrency* and *shared-memory parallelism* to OCaml
- Focus of this work is *parallelism*
  - ✦ Building a multicore GC for OCaml

Multicore OCaml

- Adds native support for *concurrency* and *shared-memory parallelism* to OCaml
- Focus of this work is *parallelism*
  - ✦ Building a multicore GC for OCaml
- Key parallel GC design principle
  - ✦ *Backwards compatibility before parallel scalability*

Challenges

- Millions of lines of legacy code
  - ✦ Weak references, ephemerons, lazy values, finalisers
  - ✦ Low-level C API that *bakes in* GC invariants
  - ✦ Cost of refactoring sequential code itself is prohibitive

Challenges

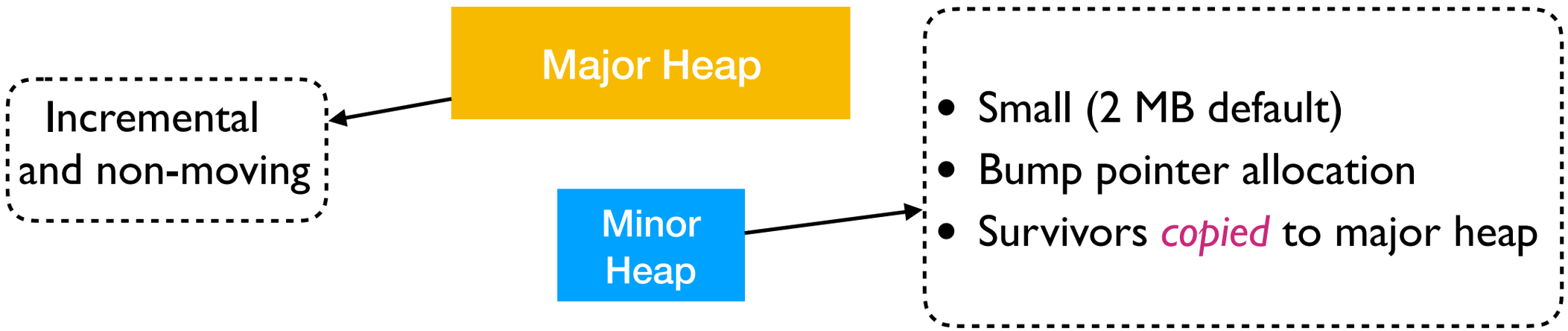
- Millions of lines of legacy code
  - ✦ Weak references, ephemerons, lazy values, finalisers
  - ✦ Low-level C API that *bakes in* GC invariants
  - ✦ Cost of refactoring sequential code itself is prohibitive
- Type safety
  - ✦ Dolan et al, "*Bounding Data Races in Space and Time*", PLDI'18
  - ✦ Strong guarantees (including type safety) under data races

Challenges

- Millions of lines of legacy code
  - ✦ Weak references, ephemerons, lazy values, finalisers
  - ✦ Low-level C API that *bakes in* GC invariants
  - ✦ Cost of refactoring sequential code itself is prohibitive
- Type safety
  - ✦ Dolan et al, "*Bounding Data Races in Space and Time*", PLDI'18
  - ✦ Strong guarantees (including type safety) under data races
- Low-latency and predictable performance
  - ✦ Thanks to the GC design

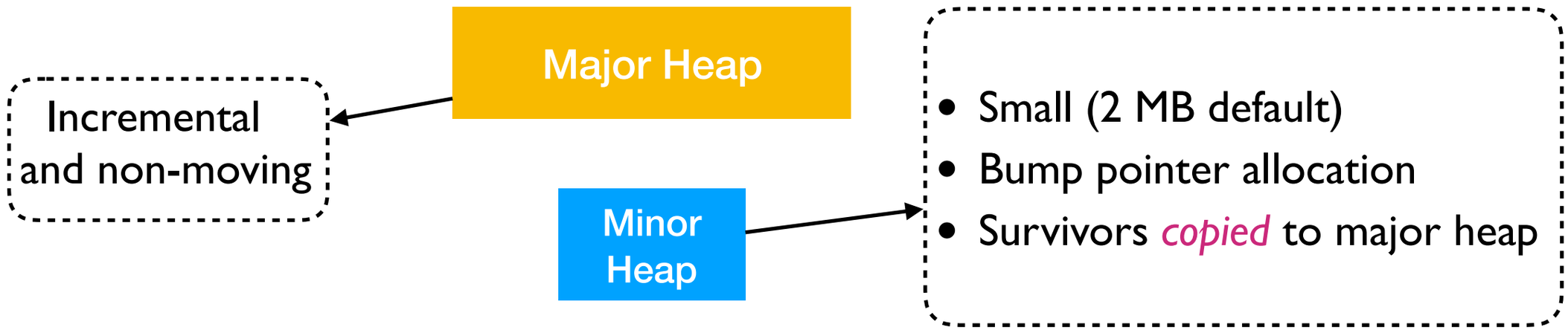
Stock OCaml GC

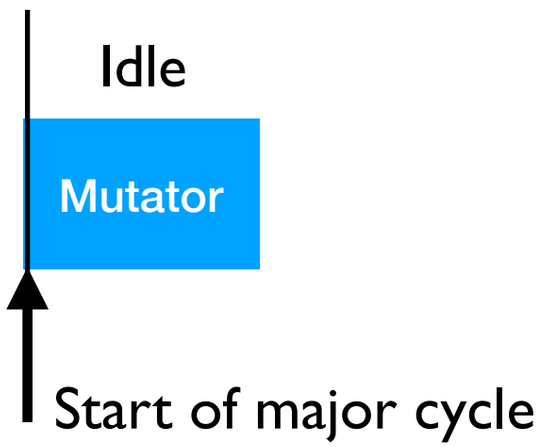
- A generational, non-moving, incremental, mark-and-sweep GC



Stock OCaml GC

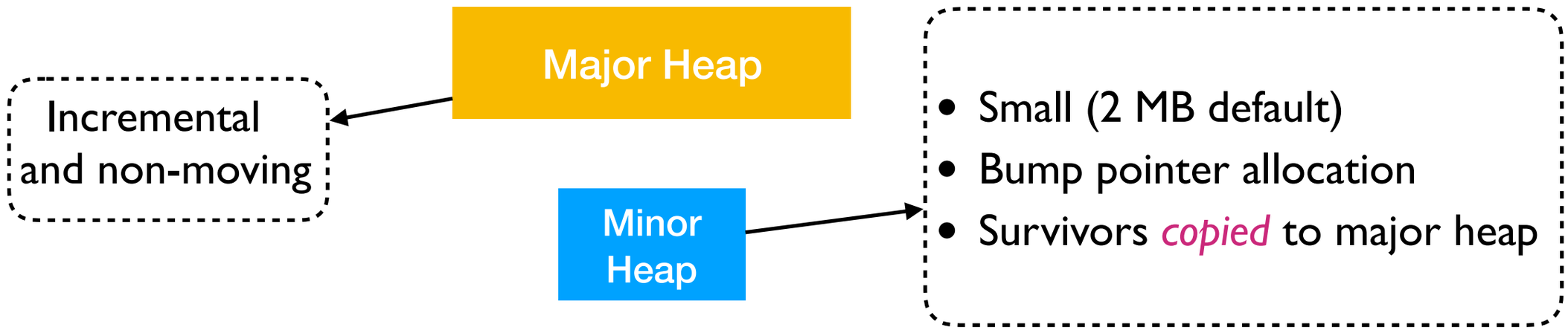
- A generational, non-moving, incremental, mark-and-sweep GC

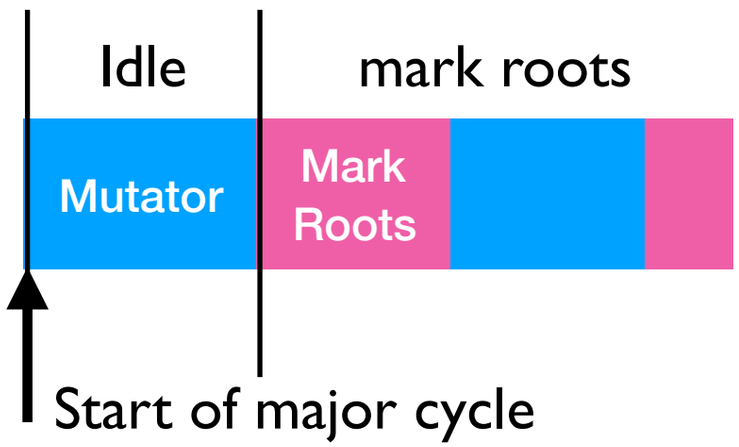




Stock OCaml GC

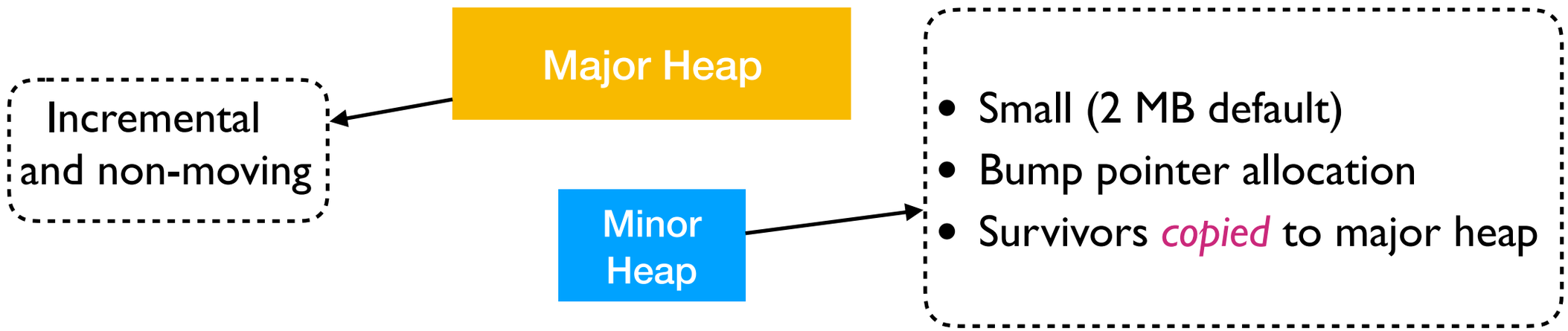
- A generational, non-moving, incremental, mark-and-sweep GC

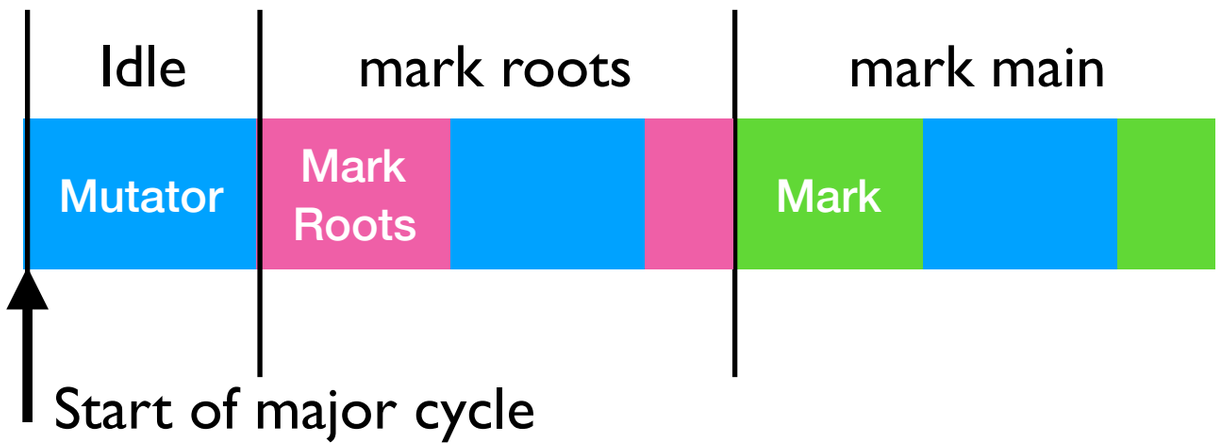




Stock OCaml GC

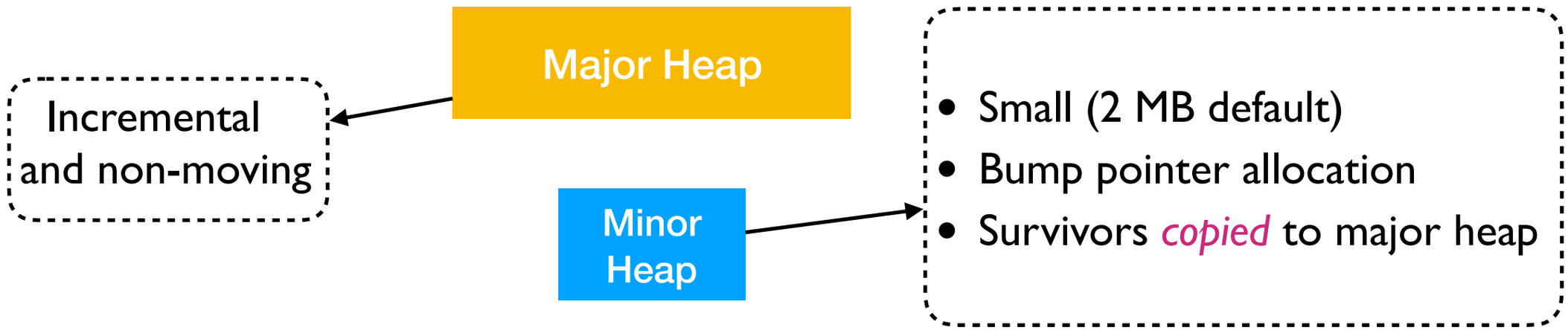
- A generational, non-moving, incremental, mark-and-sweep GC

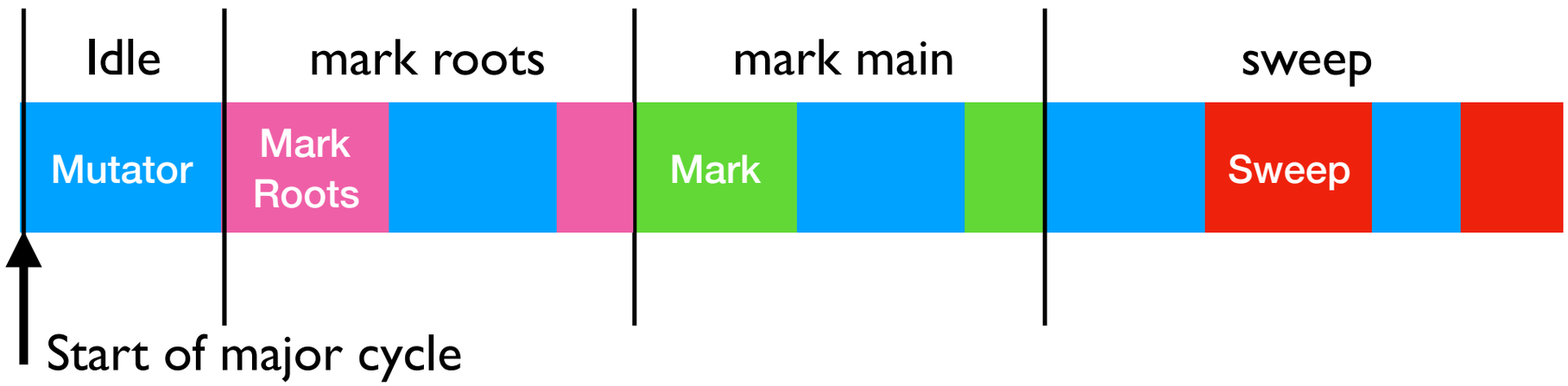




Stock OCaml GC

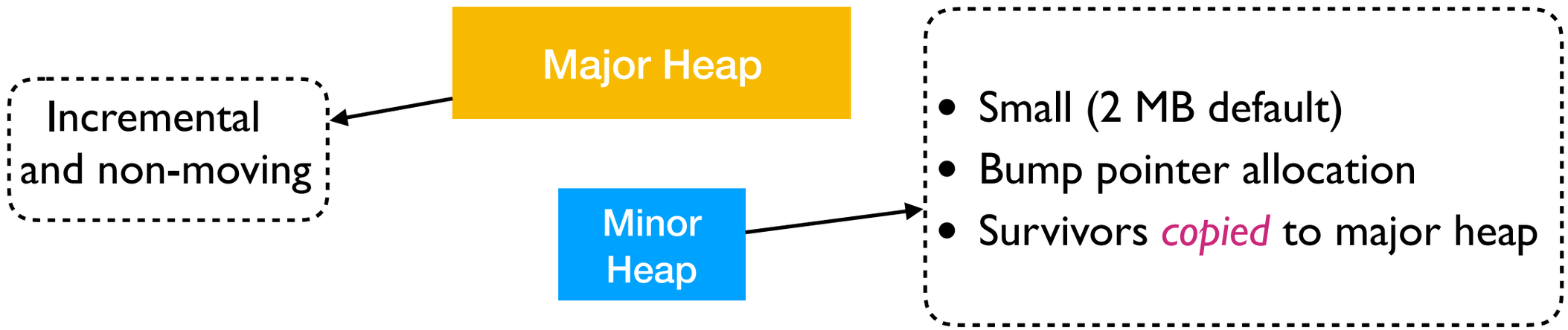
- A generational, non-moving, incremental, mark-and-sweep GC

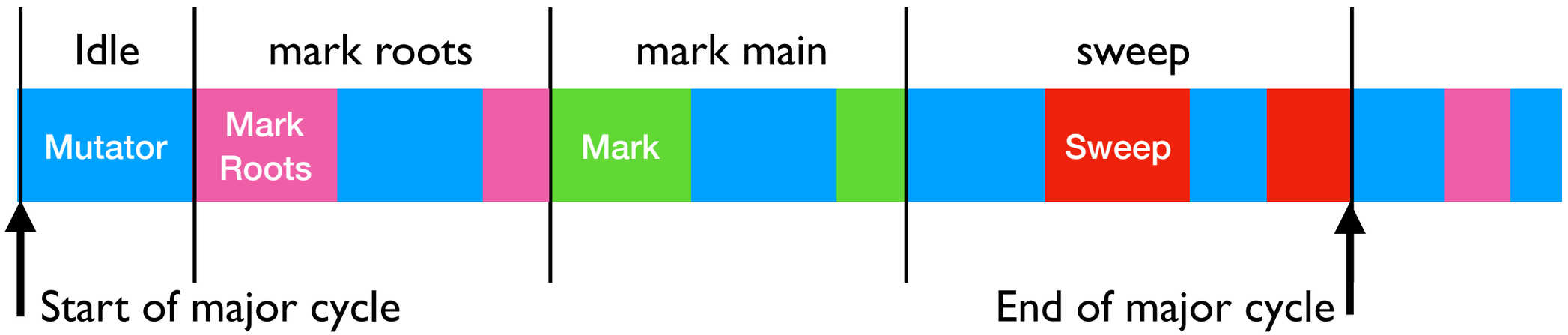




Stock OCaml GC

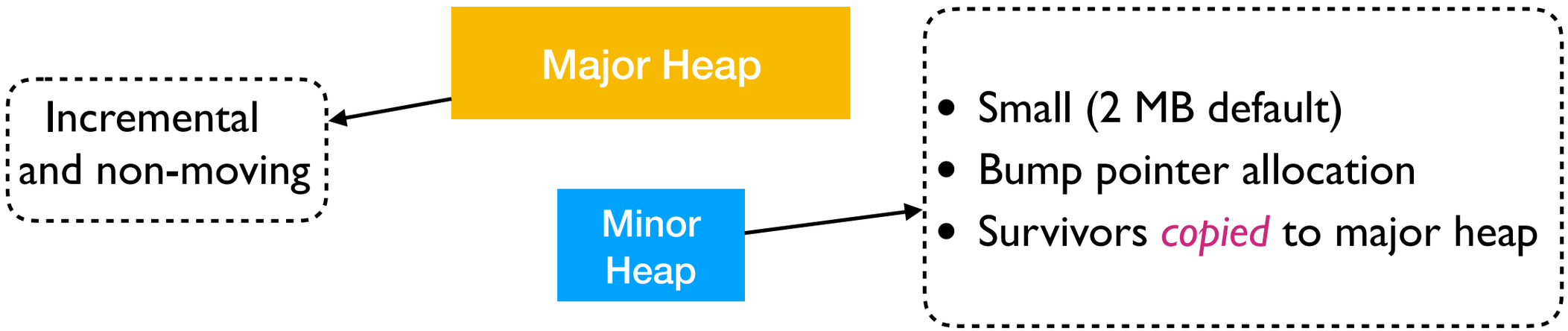
- A generational, non-moving, incremental, mark-and-sweep GC

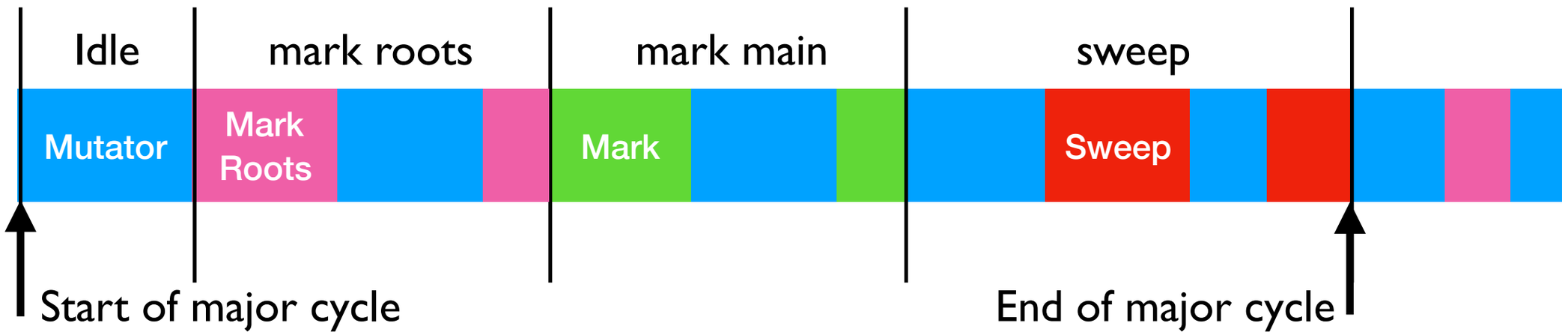




Stock OCaml GC

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

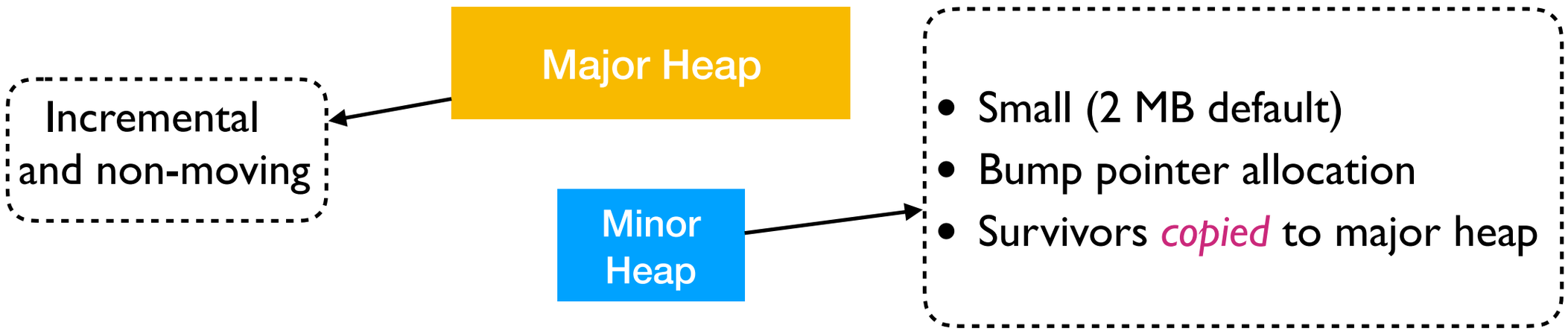


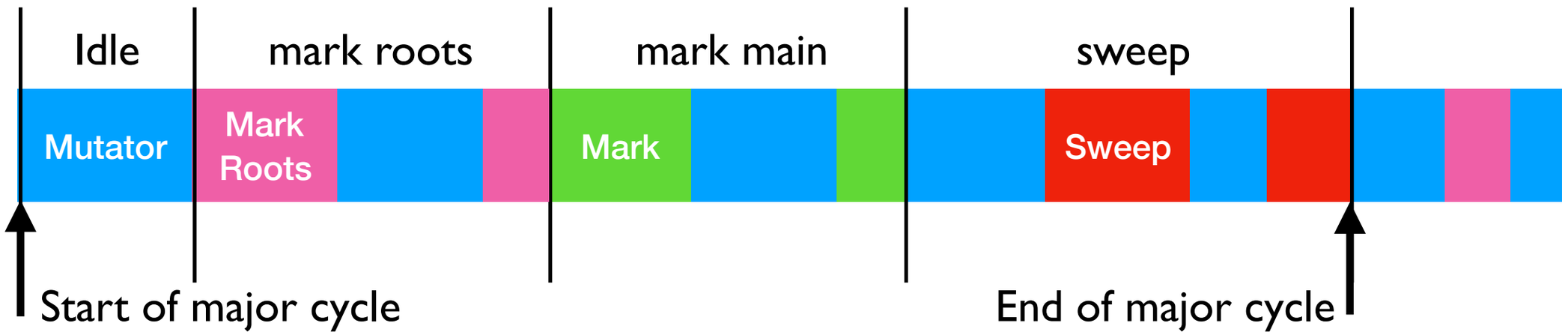


- Fast allocations, no read barriers

Stock OCaml GC

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





- Fast allocations, no read barriers
- Max GC latency **< 10 ms**, 99th percentile latency **< 1 ms**

Requirements

1. **Feature backwards compatibility**
    - Serial programs do not break on parallel runtime
    - No separate serial and parallel modes

Requirements

1. **Feature backwards compatibility**
   - Serial programs do not break on parallel runtime
   - No separate serial and parallel modes
2. **Performance backwards compatibility**
   - Serial programs behave similarly on parallel runtime in terms of running time, GC pausetime and memory usage.

Requirements

1. **Feature backwards compatibility**
   - Serial programs do not break on parallel runtime
   - No separate serial and parallel modes
2. **Performance backwards compatibility**
   - Serial programs behave similarly on parallel runtime in terms of running time, GC pausetime and memory usage.
3. **Parallel responsiveness and scalability**
   - Parallel programs remain responsive
   - Parallel programs scale with additional cores

Multicore OCaml: Major GC

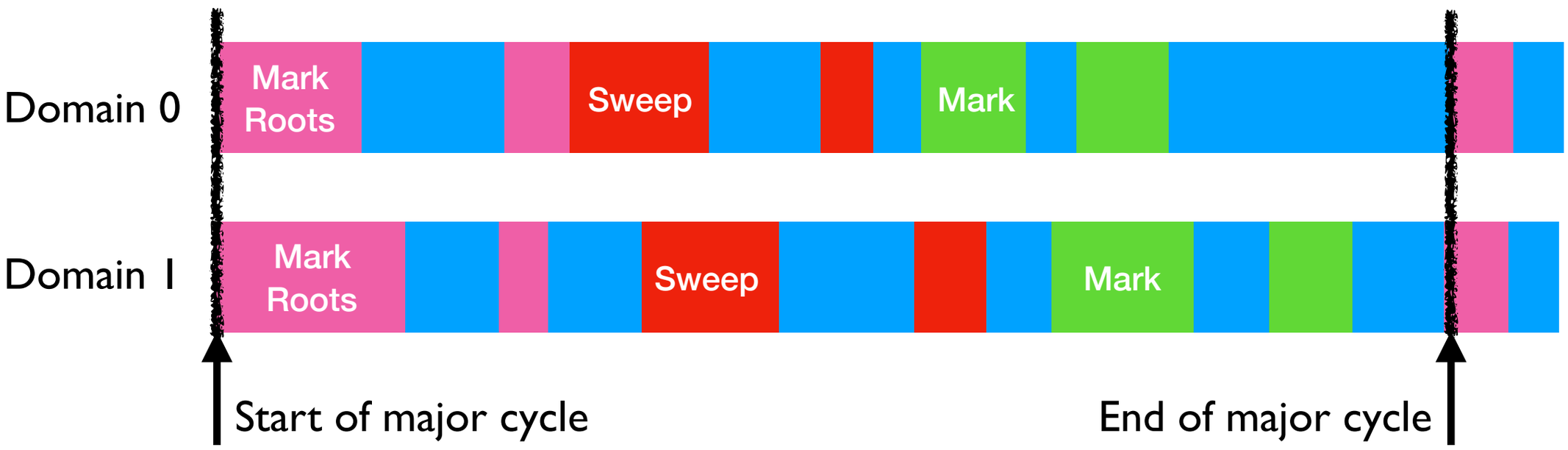
- Multicore-aware allocator
  - ✦ Based on Streamflow [Schneider et al. 2006]
  - ✦ *Thread-local, size-segmented free lists* for small objects + malloc for large allocations
  - ✦ Sequential performance on par with OCaml's allocators

Multicore OCaml: Major GC

- Multicore-aware allocator
  - ✦ Based on Streamflow [Schneider et al. 2006]
  - ✦ *Thread-local, size-segmented free lists* for small objects + malloc for large allocations
  - ✦ Sequential performance on par with OCaml's allocators
- A mostly-concurrent, non-moving, mark-and-sweep collector
  - ✦ Based on VCGC [Huelsbergen and Winterbottom 1998]

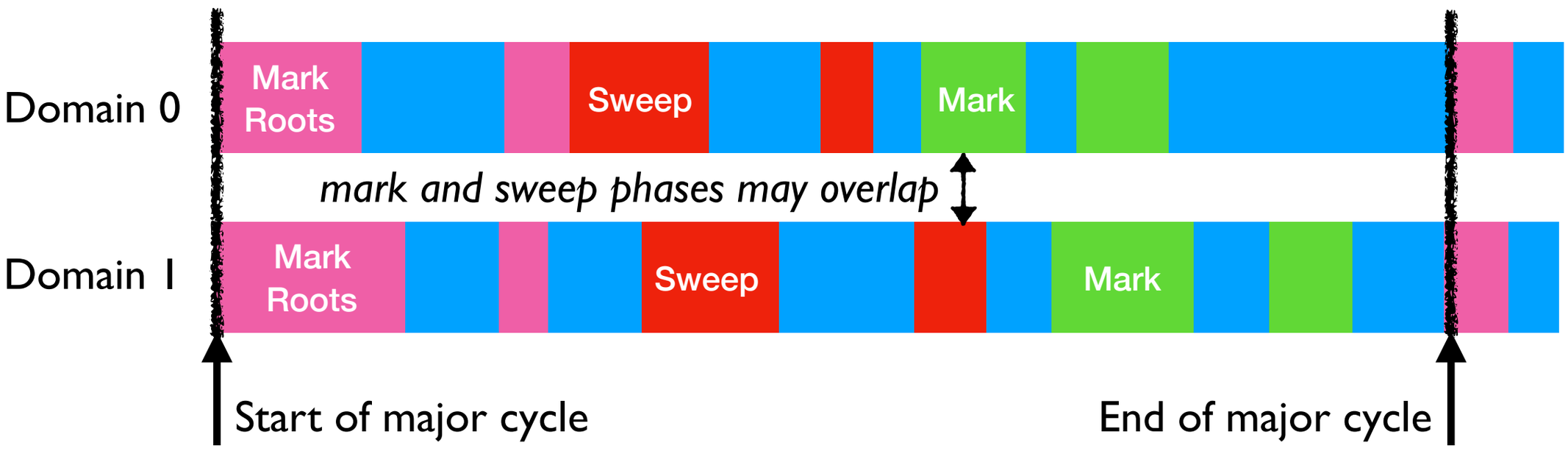
Multicore OCaml: Major GC

- Multicore-aware allocator
  - ✦ Based on Streamflow [Schneider et al. 2006]
  - ✦ *Thread-local, size-segmented free lists* for small objects + malloc for large allocations
  - ✦ Sequential performance on par with OCaml's allocators
- A mostly-concurrent, non-moving, mark-and-sweep collector
  - ✦ Based on VCGC [Huelsbergen and Winterbottom 1998]



Multicore OCaml: Major GC

- Multicore-aware allocator
  - ✦ Based on Streamflow [Schneider et al. 2006]
  - ✦ *Thread-local, size-segmented free lists* for small objects + malloc for large allocations
  - ✦ Sequential performance on par with OCaml's allocators
- A mostly-concurrent, non-moving, mark-and-sweep collector
  - ✦ Based on VCGC [Huelsbergen and Winterbottom 1998]



Multicore OCaml: Major GC

Multicore OCaml: Major GC

- Extend support weak references, ephemerons, (2 different kinds of) finalizers, fibers, lazy values

Multicore OCaml: Major GC

- Extend support weak references, ephemerons, (2 different kinds of) finalizers, fibers, lazy values
- Ephemerons are tricky in a concurrent multicore GC
  - ✦ A generalisation of weak references
  - ✦ Introduce *conjunction* in the reachability property
  - ✦ Requires multiple rounds of ephemeron marking
  - ✦ Cycle-delimited handshaking *without global barrier*

Multicore OCaml: Major GC

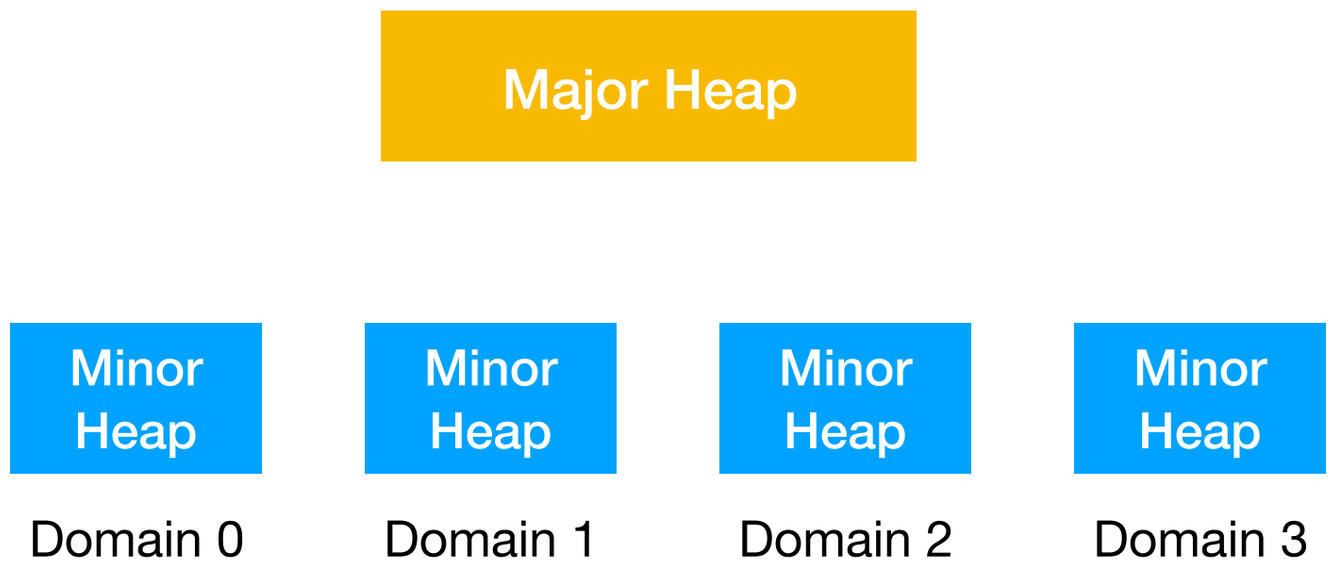
- Extend support weak references, ephemerons, (2 different kinds of) finalizers, fibers, lazy values
- Ephemerons are tricky in a concurrent multicore GC
  - ✦ A generalisation of weak references
  - ✦ Introduce *conjunction* in the reachability property
  - ✦ Requires multiple rounds of ephemeron marking
  - ✦ Cycle-delimited handshaking *without global barrier*
- A barrier each for the two kinds of finalisers
  - ✦ 3 barriers / cycle worst case

Multicore OCaml: Major GC

- Extend support weak references, ephemerons, (2 different kinds of) finalizers, fibers, lazy values
- Ephemerons are tricky in a concurrent multicore GC
  - ✦ A generalisation of weak references
  - ✦ Introduce *conjunction* in the reachability property
  - ✦ Requires multiple rounds of ephemeron marking
  - ✦ Cycle-delimited handshaking *without global barrier*
- A barrier each for the two kinds of finalisers
  - ✦ 3 barriers / cycle worst case
- *Verified in the SPIN model checker*

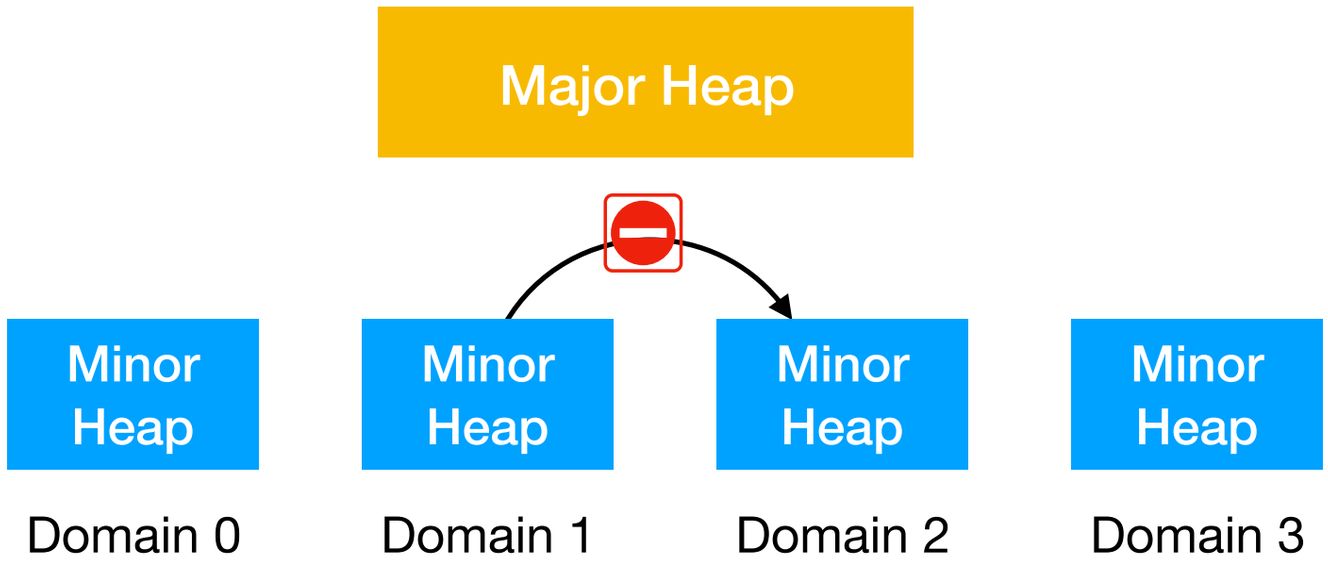
Concurrent Minor GC

- Based on [Doligez and Leroy 1993] but lazier as in [Marlow and Peyton Jones 2011] collector for GHC



Concurrent Minor GC

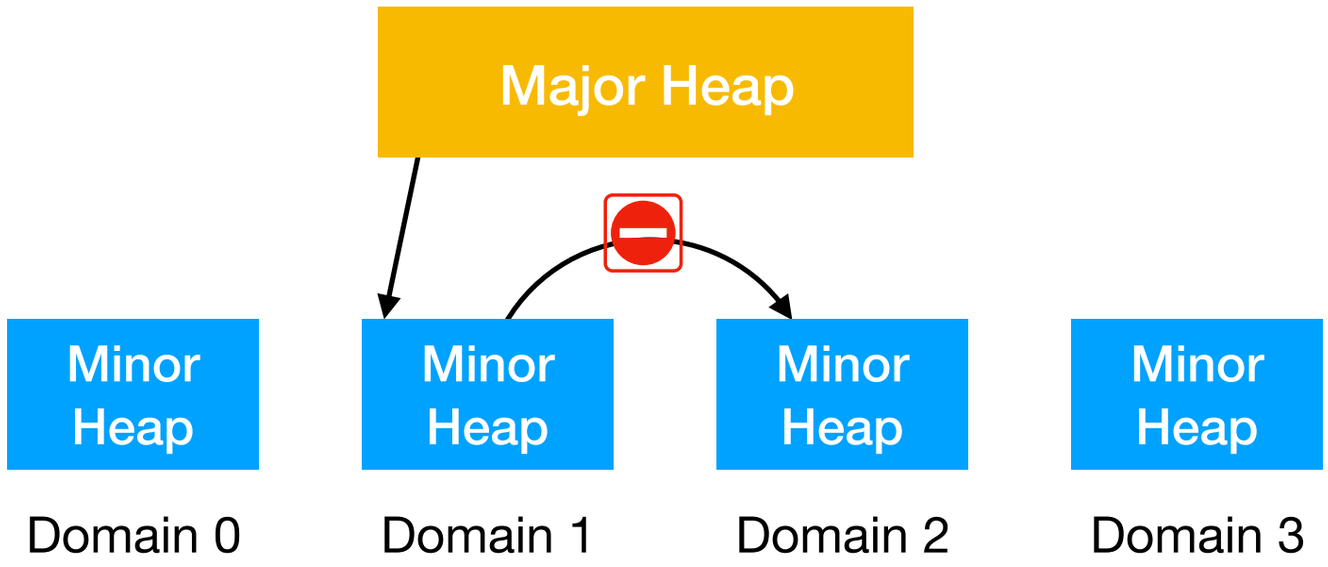
- Based on [Doligez and Leroy 1993] but lazier as in [Marlow and Peyton Jones 2011] collector for GHC



- Each domain can *independently* collect its minor heap

Concurrent Minor GC

- Based on [Doligez and Leroy 1993] but lazier as in [Marlow and Peyton Jones 2011] collector for GHC



- Each domain can *independently* collect its minor heap
- Major to minor pointers allowed
  - ✦ *Prevents early promotion & mirrors sequential behaviour*
  - ✦ *Read barrier required for mutable field + promotion*

Read Barriers

- Stock OCaml does not have read barriers
  - ✦ Read barriers need to be efficient for performance backwards compatibility

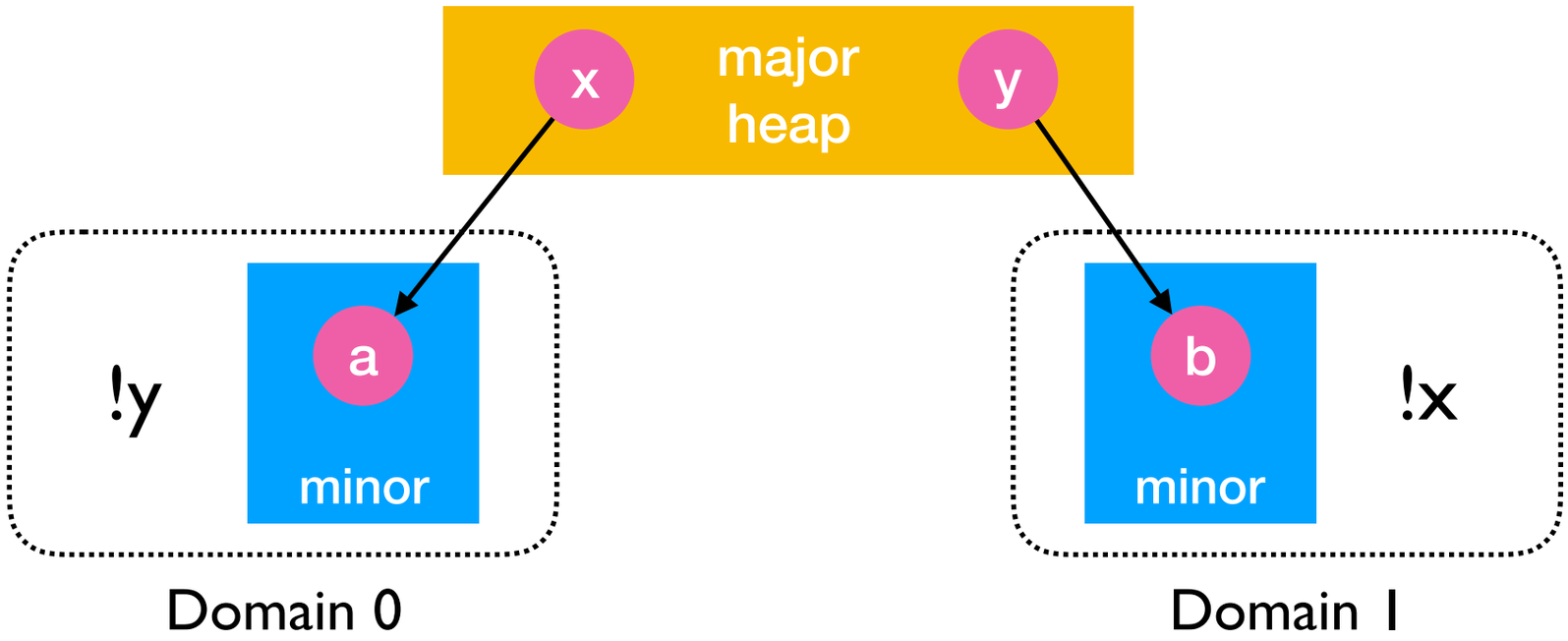
Read Barriers

- Stock OCaml does not have read barriers
  - ✦ Read barriers need to be efficient for performance backwards compatibility
- *3 instructions in x86* - VMM + bit-twiddling tricks
  - ✦ *Proof of correctness available in the paper*
  - ✦ Minimal performance impact on sequential code

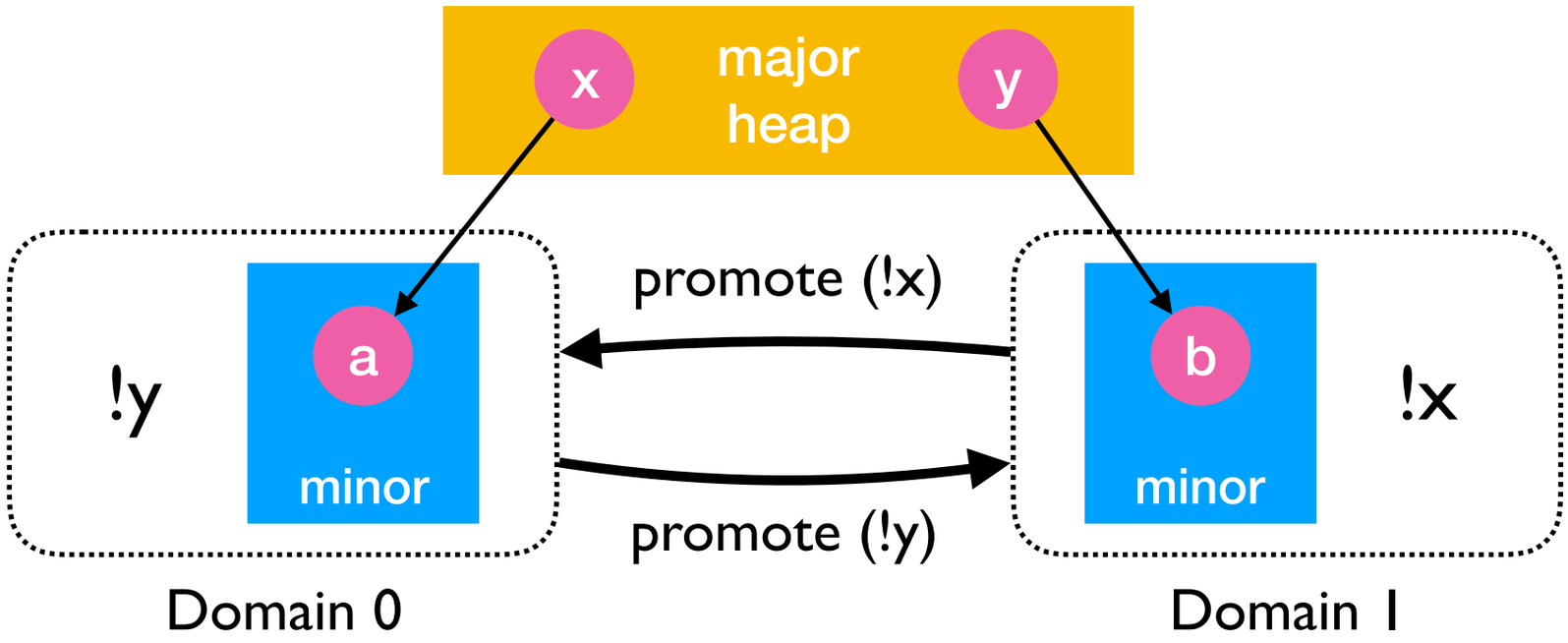
Read Barriers

- Stock OCaml does not have read barriers
  - ✦ Read barriers need to be efficient for performance backwards compatibility
- *3 instructions in x86* - VMM + bit-twiddling tricks
  - ✦ *Proof of correctness available in the paper*
  - ✦ Minimal performance impact on sequential code
- Unfortunately, read barriers *break the C API* (feature backwards compatibility)

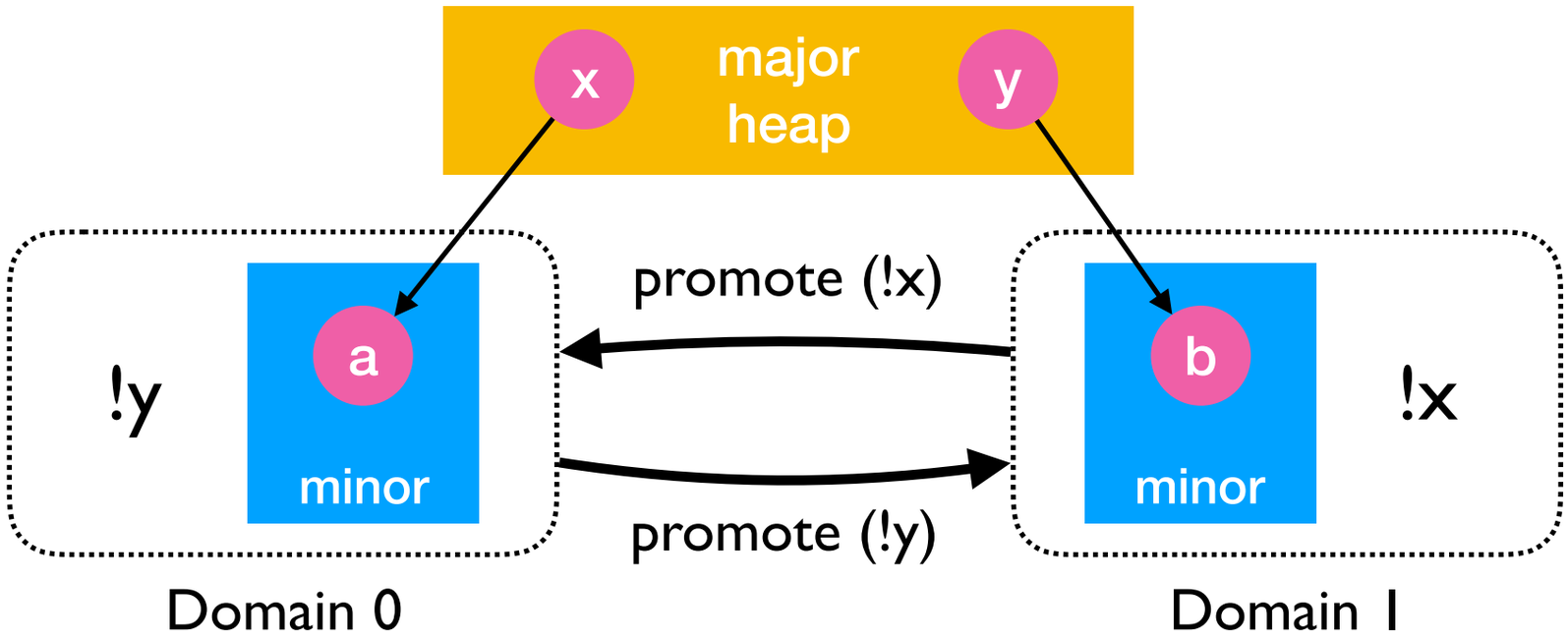
Read Barriers



Read Barriers

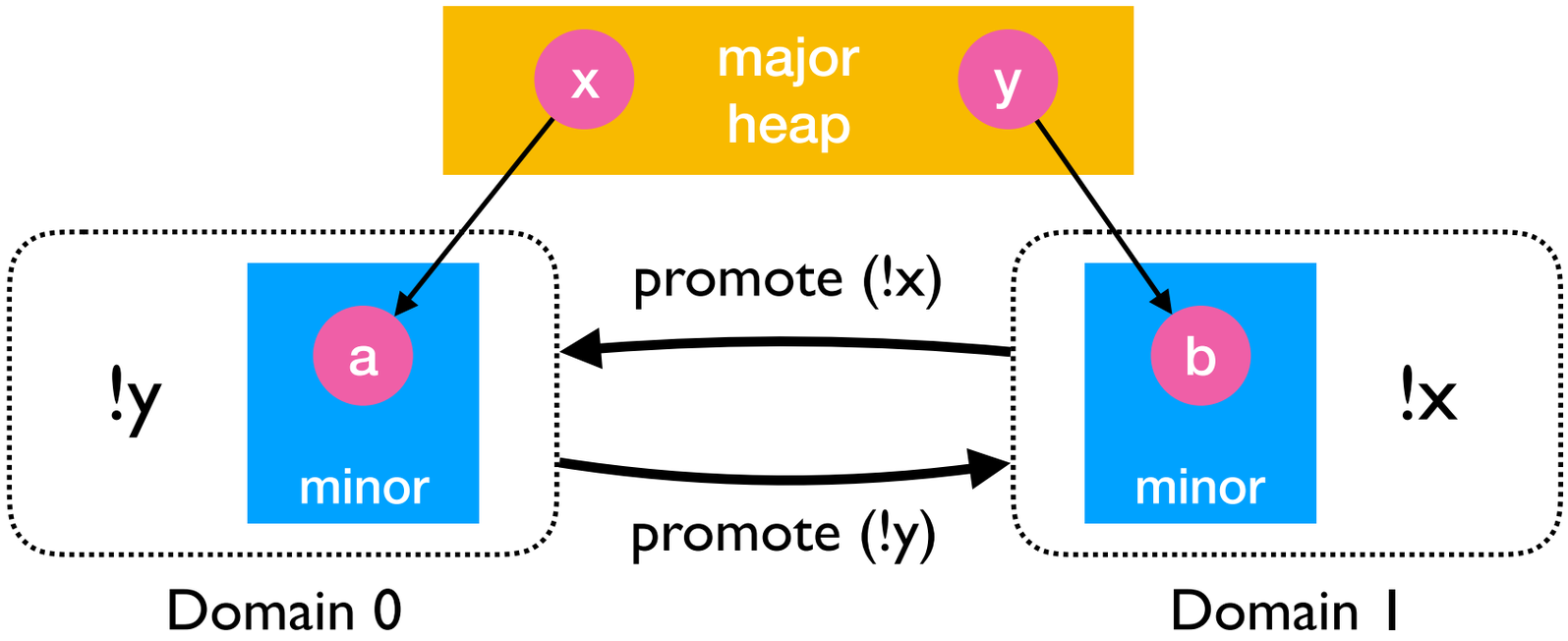


Read Barriers



- Service promotion requests on read faults to avoid *deadlock*
  - ✦ *Mutable reads are GC safe points!*

Read Barriers



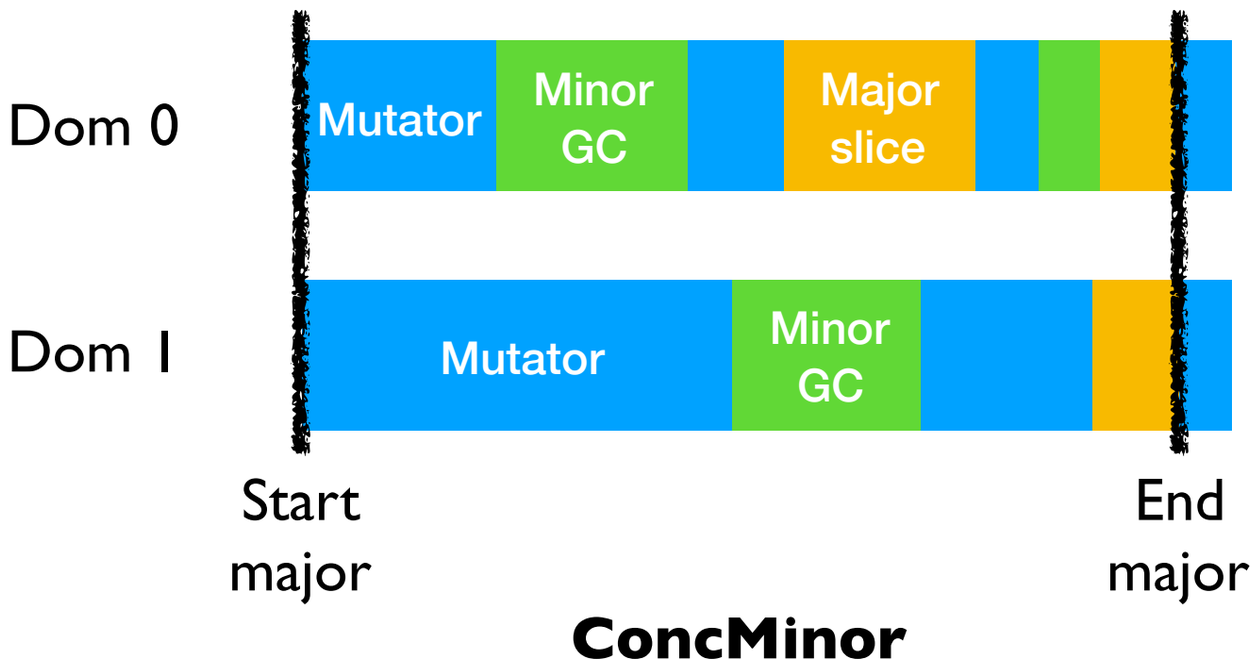
- Service promotion requests on read faults to avoid *deadlock*
  - ✦ *Mutable reads are GC safe points!*
- C API written with explicit knowledge of when GC may happen
  - ✦ *Need to manually refactor tricky code*

Parallel Minor GC

- Stop-the-world parallel minor collection
  - ✦ Similar to GHCs minor collection

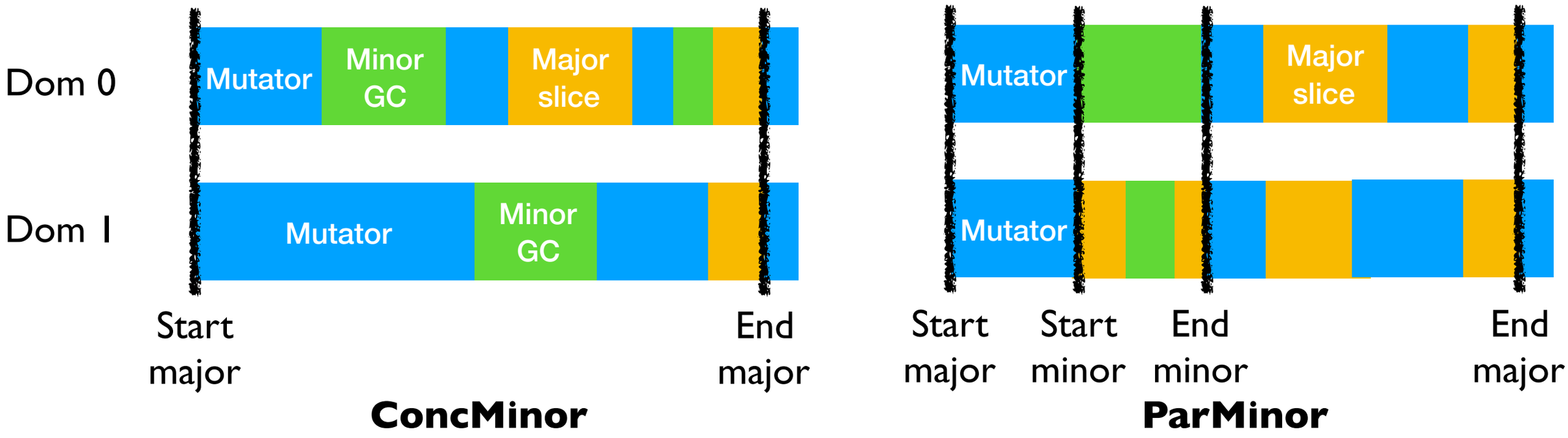
Parallel Minor GC

- Stop-the-world parallel minor collection
  - ✦ Similar to GHCs minor collection



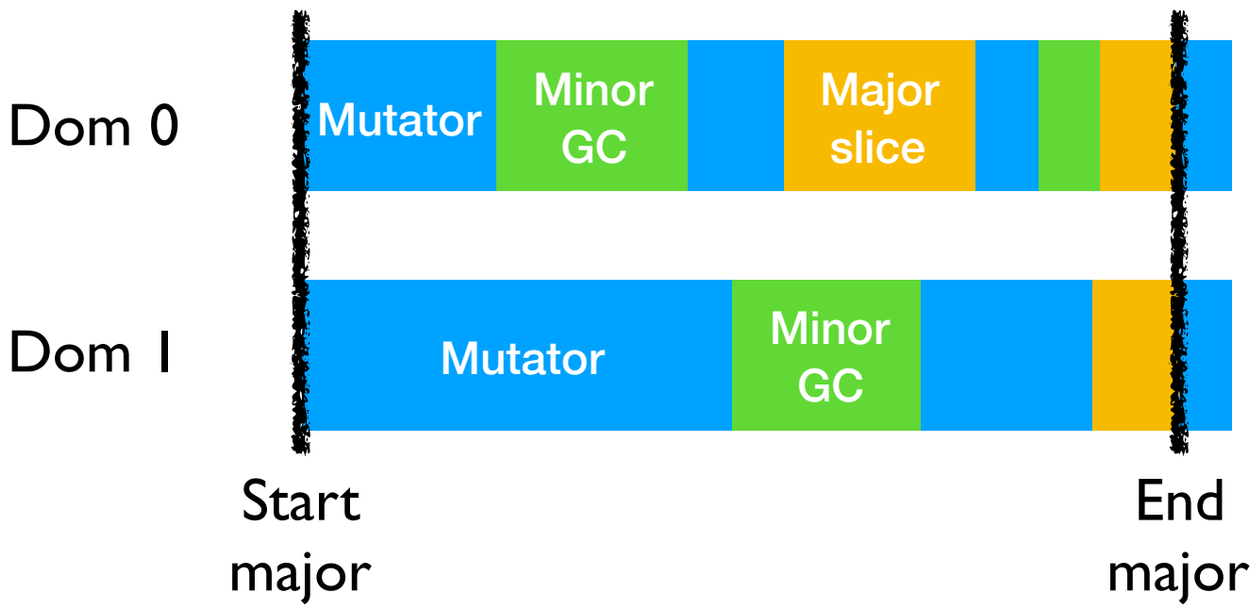
Parallel Minor GC

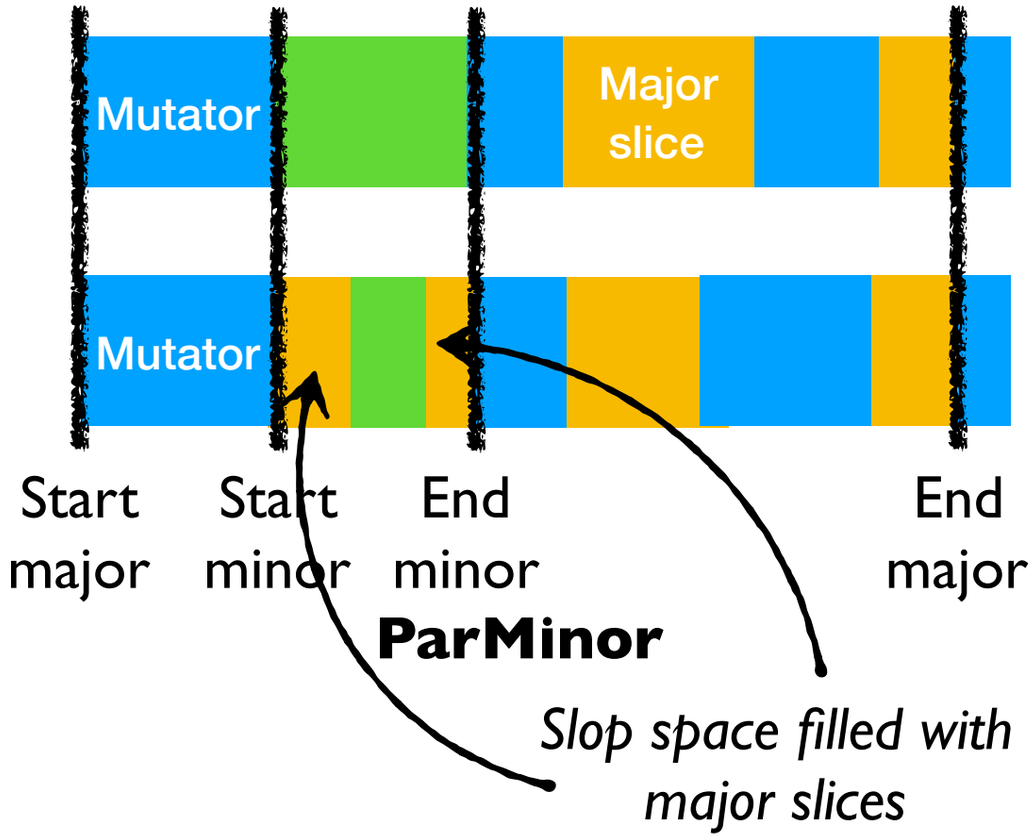
- Stop-the-world parallel minor collection
  - ✦ Similar to GHCs minor collection



Parallel Minor GC

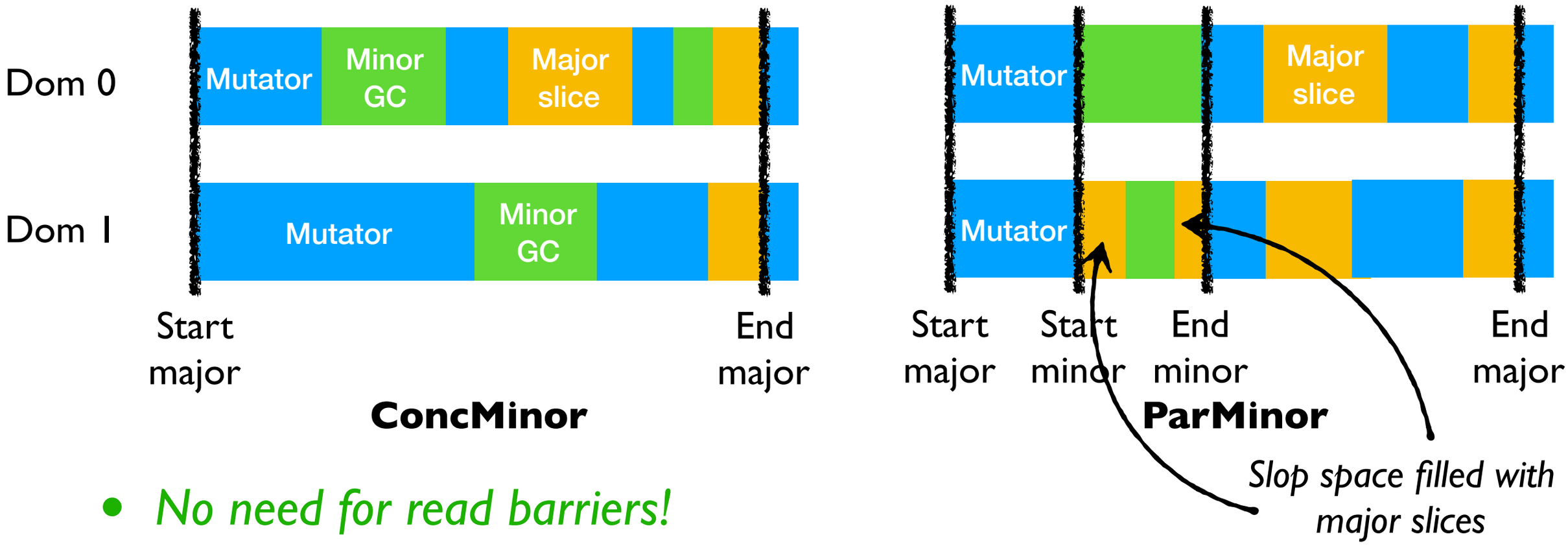
- Stop-the-world parallel minor collection
  - ✦ Similar to GHCs minor collection





Parallel Minor GC

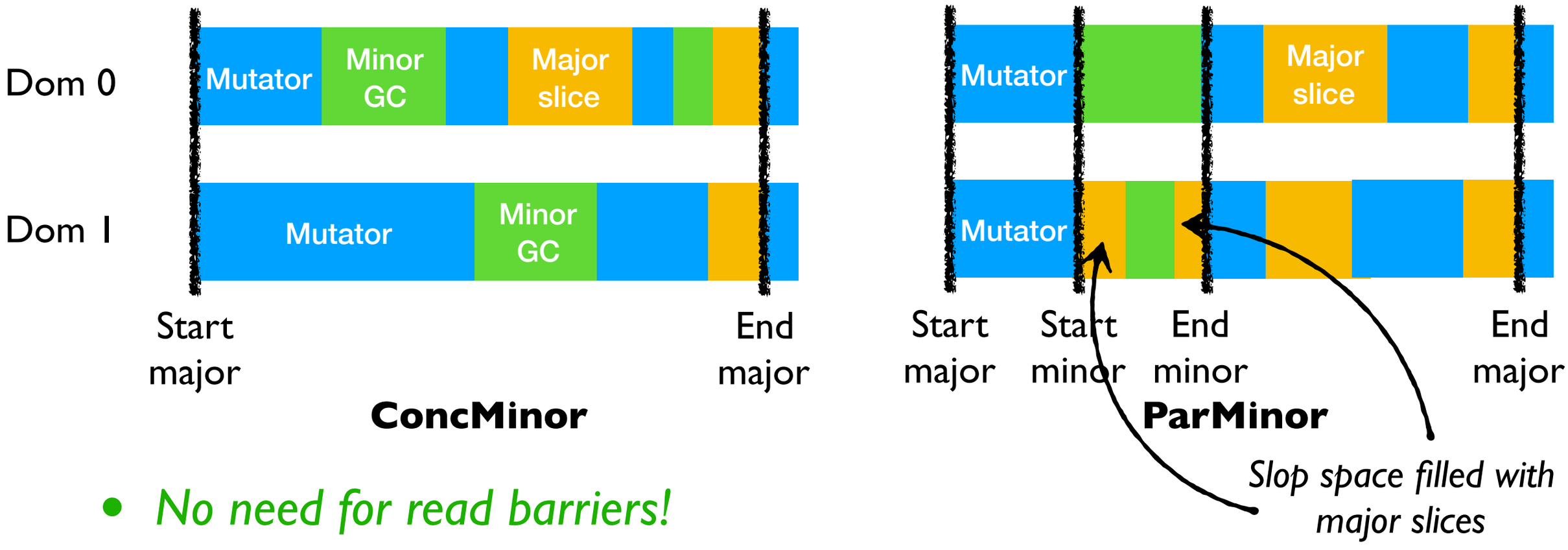
- Stop-the-world parallel minor collection
  - ✦ Similar to GHCs minor collection



- *No need for read barriers!*

Parallel Minor GC

- Stop-the-world parallel minor collection
  - ✦ Similar to GHCs minor collection



- *No need for read barriers!*
- Quickly bring all the domains to a barrier
  - ✦ Insert poll points in code for timely inter-domain interrupt handling [Feeley 1993]

Evaluation

- 2 x 14-core Intel(R) Xeon(R) Gold 5120 CPU @ 2.20GHz
  - ✦ 24 cores isolated for performance evaluation

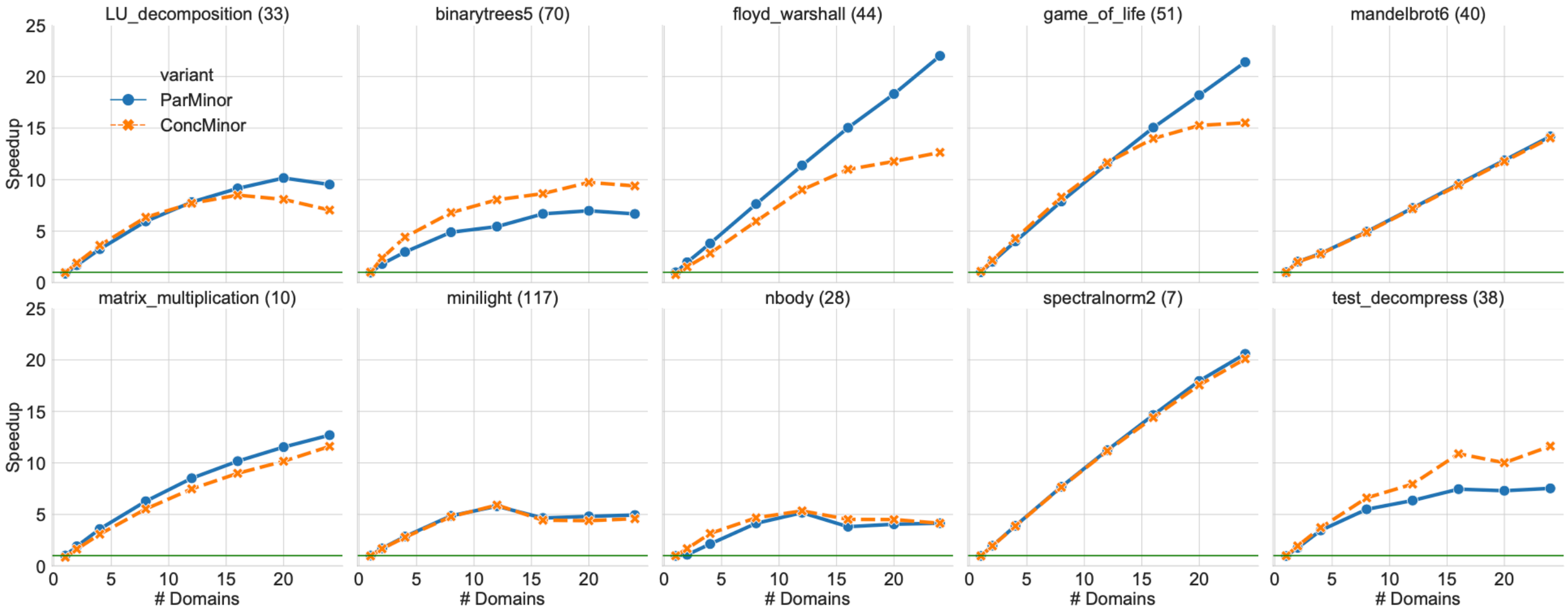
Evaluation

- 2 x 14-core Intel(R) Xeon(R) Gold 5120 CPU @ 2.20GHz
  - ✦ 24 cores isolated for performance evaluation
- Sequential Throughput — compared to stock OCaml
  - ✦ ConcMinor 4.9% slower and ParMinor 3.5% slower
  - ✦ ConcMinor 54% lower peak memory and ParMinor 61% lower peak memory

Evaluation

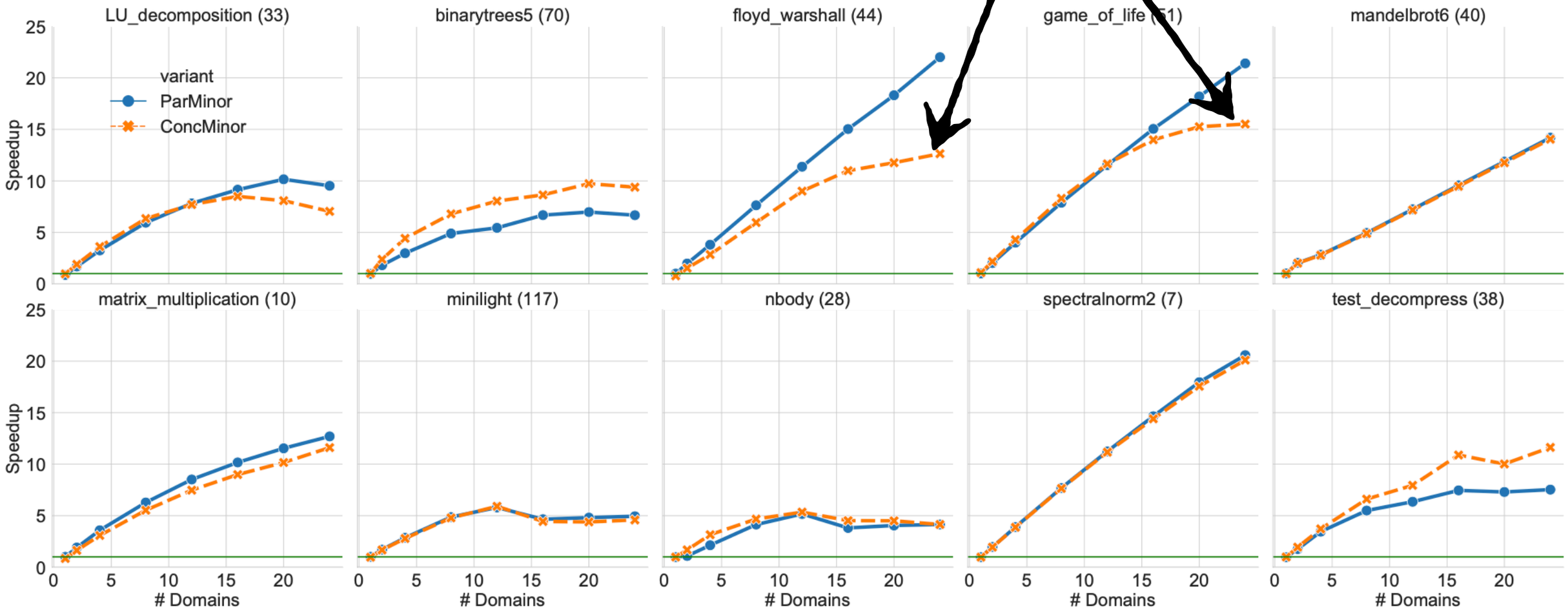
- 2 x 14-core Intel(R) Xeon(R) Gold 5120 CPU @ 2.20GHz
  - ✦ 24 cores isolated for performance evaluation
- Sequential Throughput — compared to stock OCaml
  - ✦ ConcMinor 4.9% slower and ParMinor 3.5% slower
  - ✦ ConcMinor 54% lower peak memory and ParMinor 61% lower peak memory
- Sequential GC pause times on par with stock OCaml

Parallel Scalability



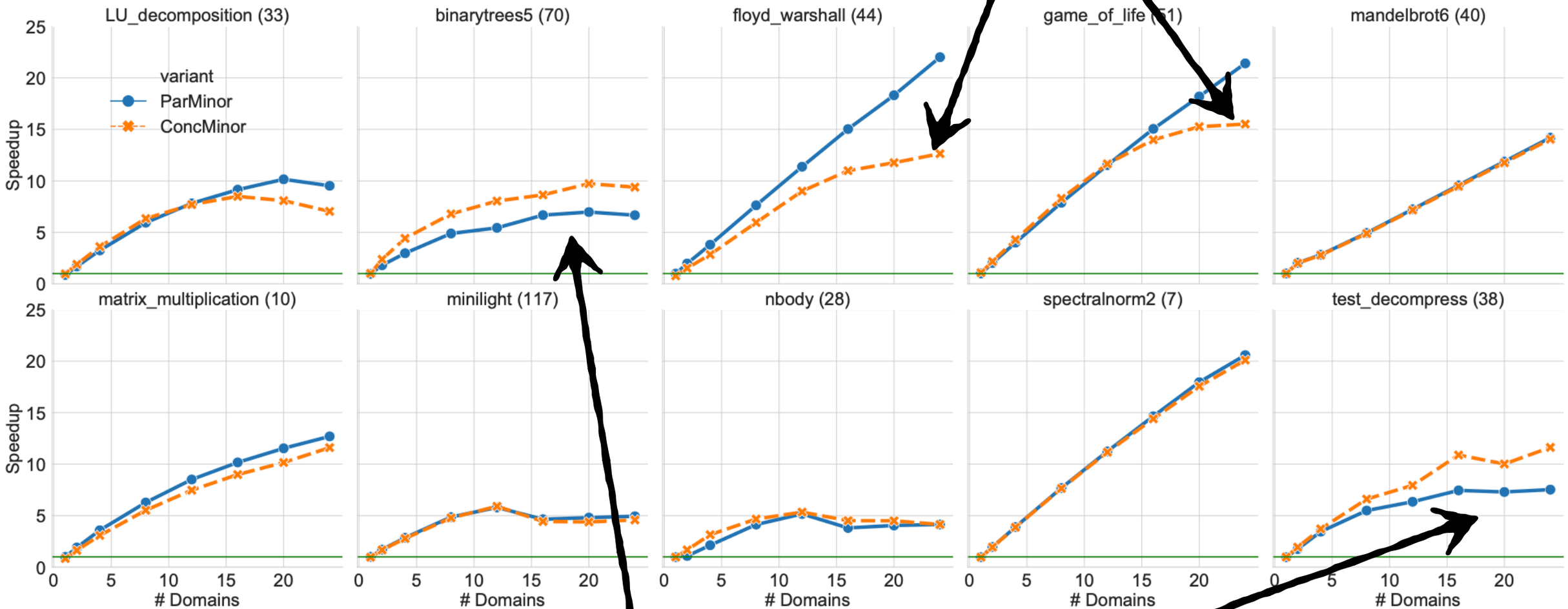
Parallel Scalability

ConcMinor suffers due to read faults



Parallel Scalability

ConcMinor suffers due to read faults



Unbalanced allocation leads to inopportune minor GCs in ParMinor

ParMinor vs ConcMinor

- Parallel GC latency roughly similar between ParMinor and ConcMinor

ParMinor vs ConcMinor

- Parallel GC latency roughly similar between ParMinor and ConcMinor
- ParMinor wins over ConcMinor
  - ✦ Does not break the C API
  - ✦ Performs similarly to the ConcMinor on 24 cores

ParMinor vs ConcMinor

- Parallel GC latency roughly similar between ParMinor and ConcMinor
- ParMinor wins over ConcMinor
  - ✦ Does not break the C API
  - ✦ Performs similarly to the ConcMinor on 24 cores
- **OCaml 5.00** will have multicore support and use ParMinor
  - ✦ May revisit ConcMinor later for manycore future

Thanks!

- Multicore OCaml
  - ✦ https://github.com/ocaml-multicore/ocaml-multicore
- Sandmark — benchmark suite for (Multicore) OCaml
  - ✦ https://github.com/ocaml-bench/sandmark/
- SPIN models
  - ✦ https://github.com/ocaml-multicore/multicore-ocaml-verify
- Parallel Programming with Multicore OCaml
  - ✦ https://github.com/ocaml-multicore/parallel-programming-in-multicore-ocaml