Shenandoah

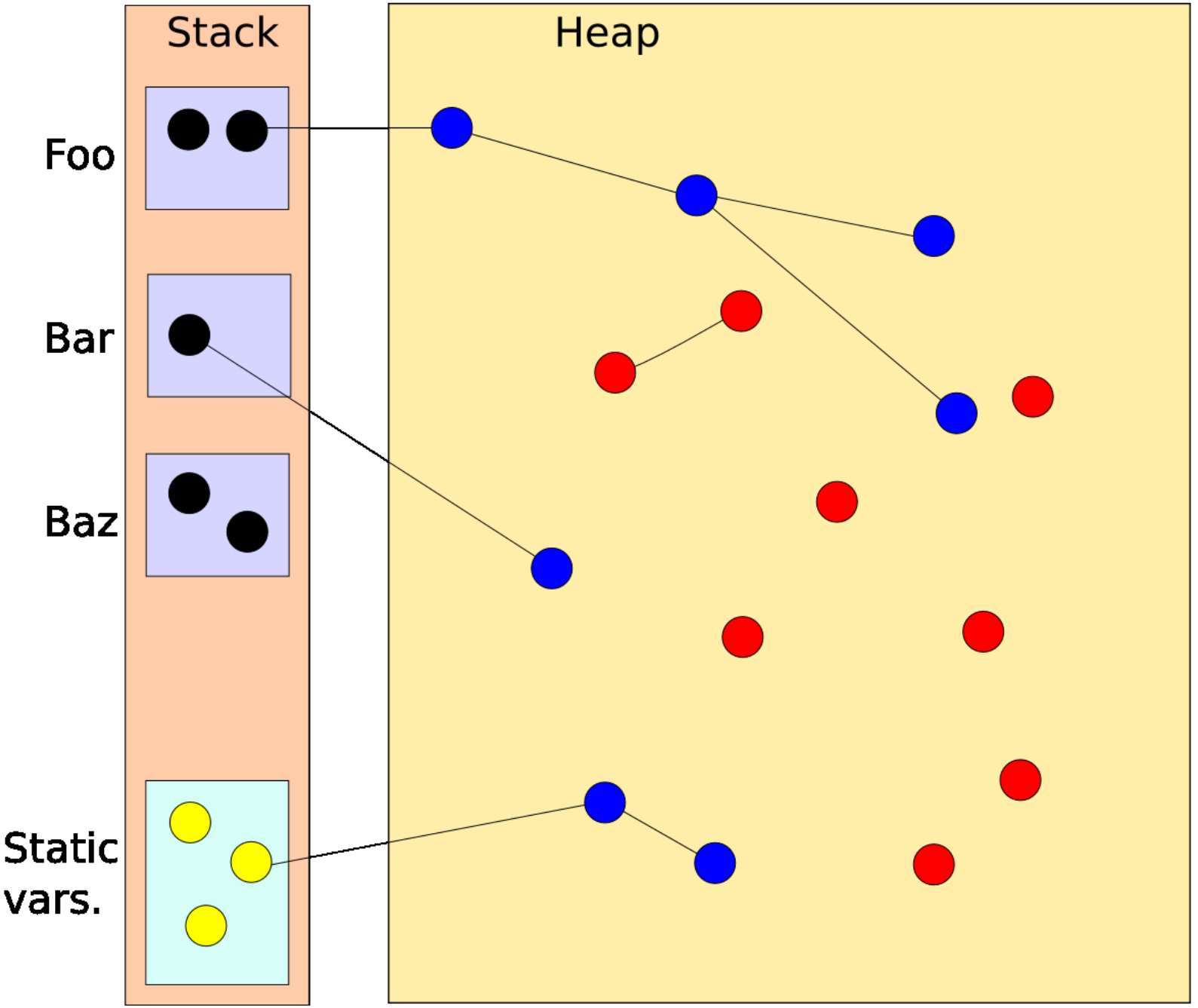
An ultra-low pause time Garbage Collector for OpenJDK

Christine H. Flood
Roman Kennke
Pavel Tišnovský

What does ultra-low pause time mean?

- It means that the pause time is proportional to the size of the root set, not the size of the heap.
- Our goal is to have < 10ms GC pause times for 100 GB+ heaps.

Size of the root set?



Why not pause-less?

- Our long term goal is to have an entirely pause-less collector.
- Shenandoah is a giant step in the right direction.

Shenandoah Features

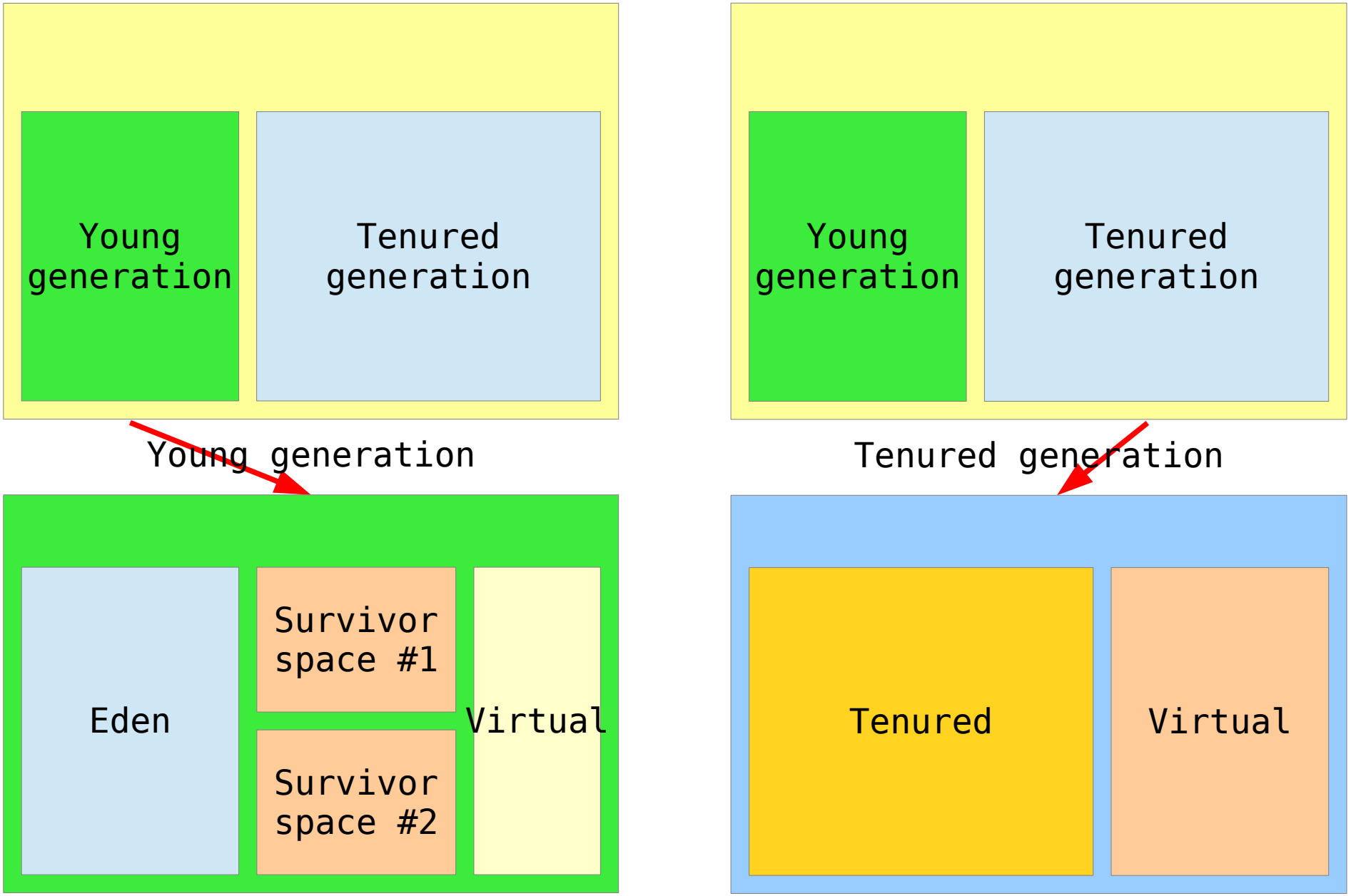
- Pauses only long enough to scan root set.
- Concurrent and parallel marking.
- Concurrent and parallel evacuation.
- No card tables or remembered sets.

Shenandoah
vs.
Generational GC

Generational GC

- Based on generational hypothesis
  - “most objects die young”
- Two generations (and two heap areas)
  - young (nursery)
  - old (tenured)
  - objects are allocated in young generation
- TLAB
  - Thread-local allocation buffer
  - allocated for each thread (a.k.a. mutator)
  - allocation = pointer bumping

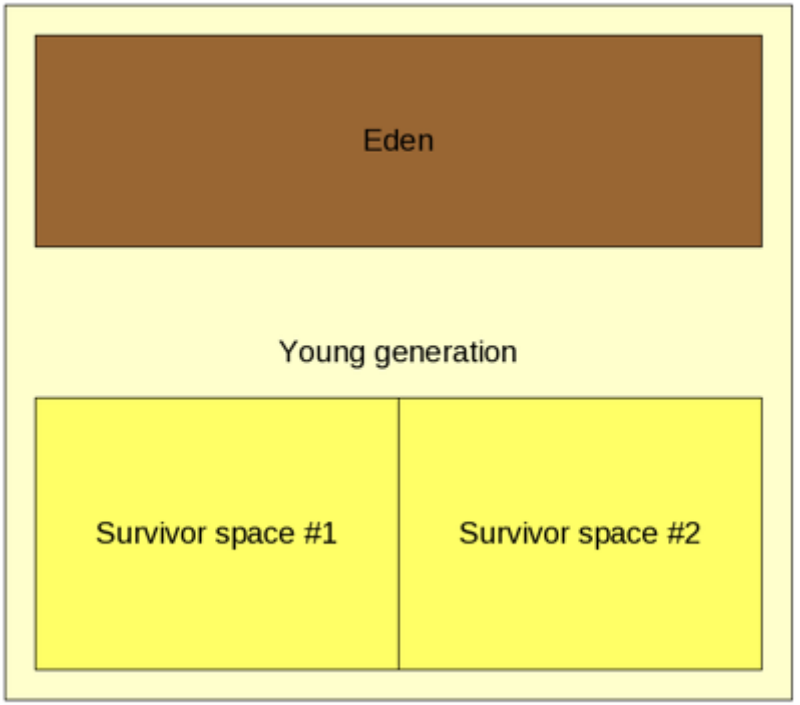
Generational GC



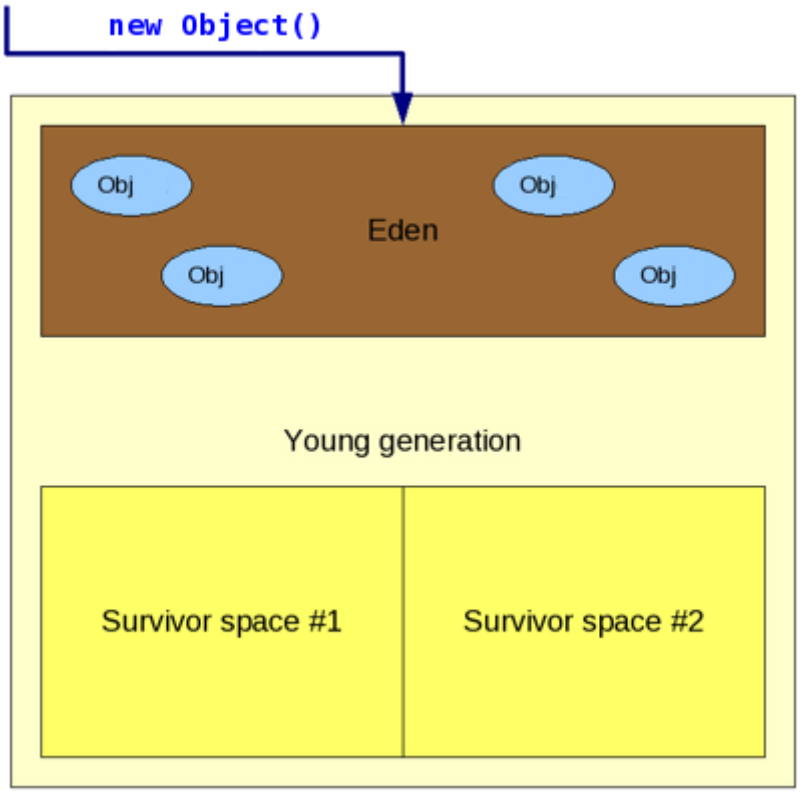
Three types of generational collectors

- Serial collector
  - good for smaller heaps (<1GB)
  - young collection + full collection (both stop-the-world)
  - short + long pause times
- Parallel collector
  - good for multicore machines and larger heaps (>2GB)
  - similar to serial GC, but uses more GC threads
- Concurrent collector
  - parallel scavenger (young generation, work stealing)
  - medium heaps (~4GB), CMS for old generation

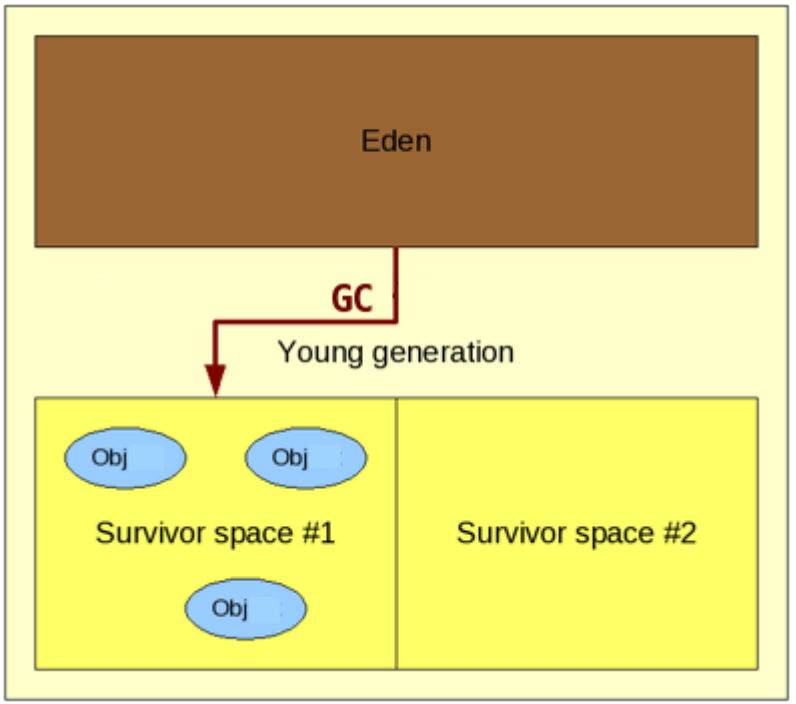
Compaction in generational GCs



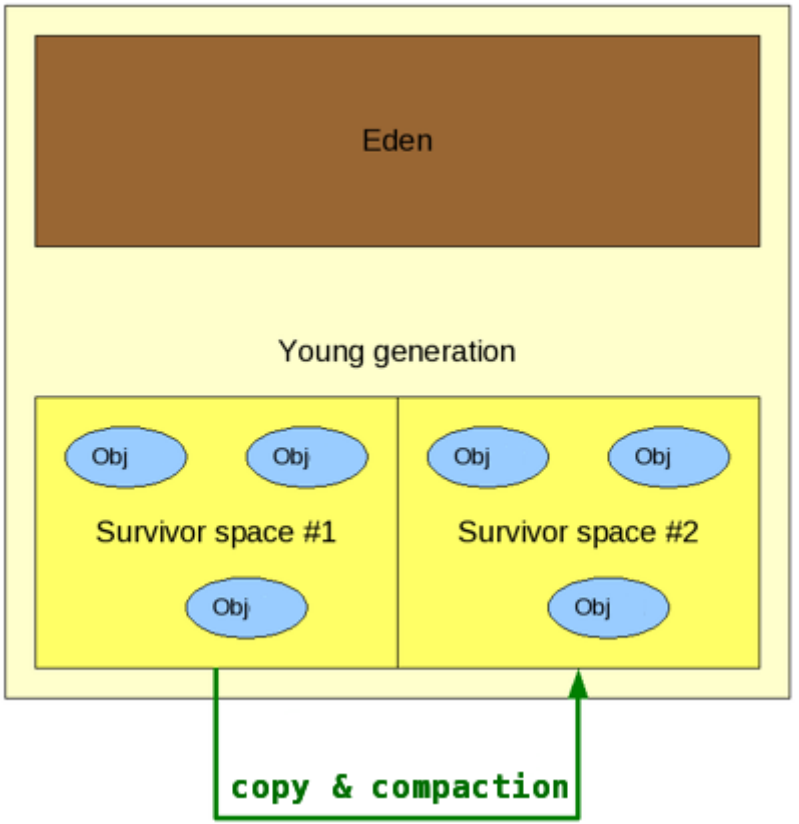
Compaction in generational GCs



Compaction in generational GCs

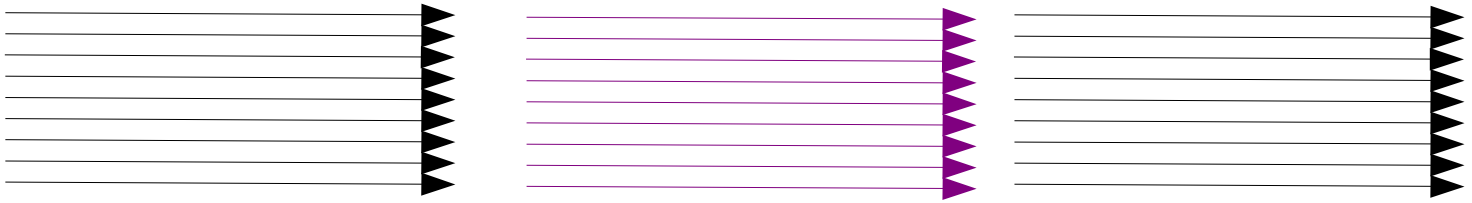


Compaction in generational GCs

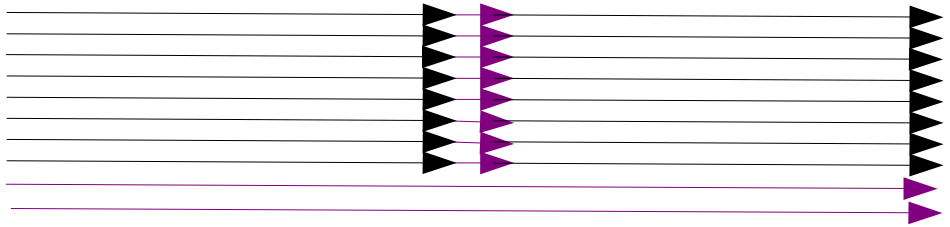


Sequential GC

Parallel GC

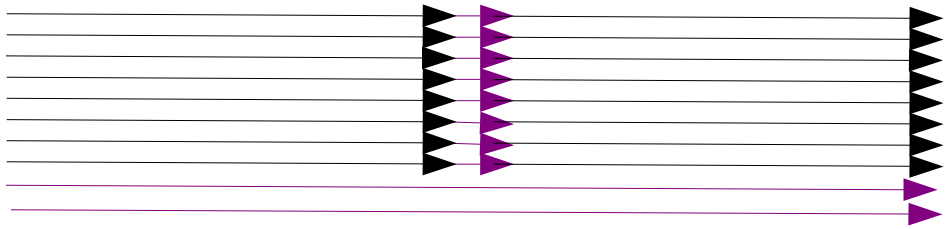


Concurrent Mark and Sweep



No compaction

Shenandoah



Compaction

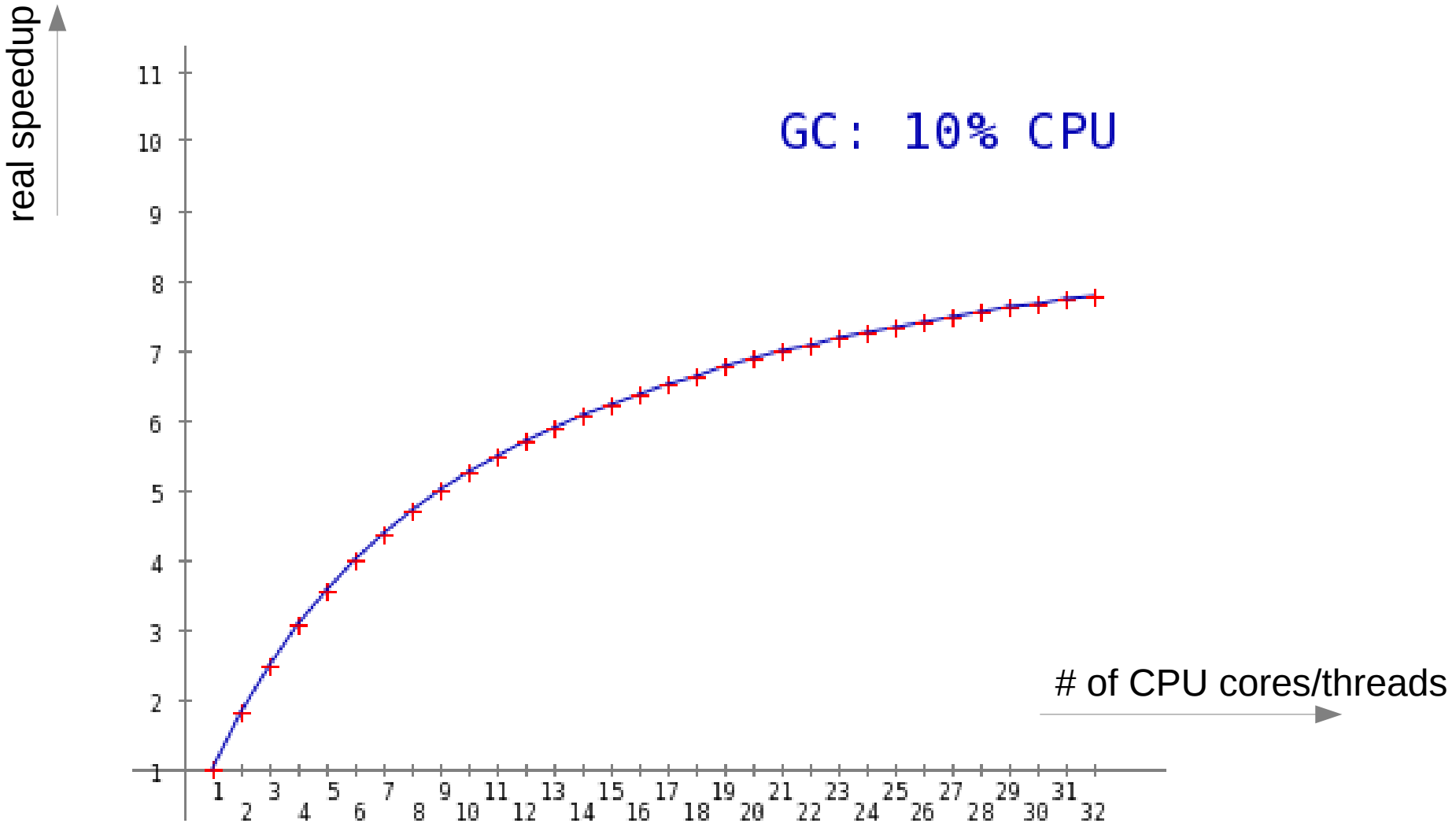
Why is Compaction Important?

- Allocation by pointer bumping is much faster than scanning a free list.
- Mark and Sweep eventually leads to fragmentation which results in an inefficient VM.

How do we get there?

- We need to have a compacting GC do the evacuation work as well as marking work while the Java threads are running.
- All the existing OpenJDK GC algorithms stop the Java threads during compaction.

Stop-the-world GC and Amdahl's law



Why is it hard?

- What if the GC moves an object that the Java threads are modifying?
- We can't have two active copies of an object.
- There may be multiple heap locations that refer to a single object. When that object moves they all have to start using the new copy of the object.

How do we do that?

- There are options:
  - We could have memory protection traps on objects which are scheduled to be moved.
  - We could add a level of indirection
    - Accesses to objects go through a forwarding pointer which allows us to update all references to an object with a single atomic instruction.

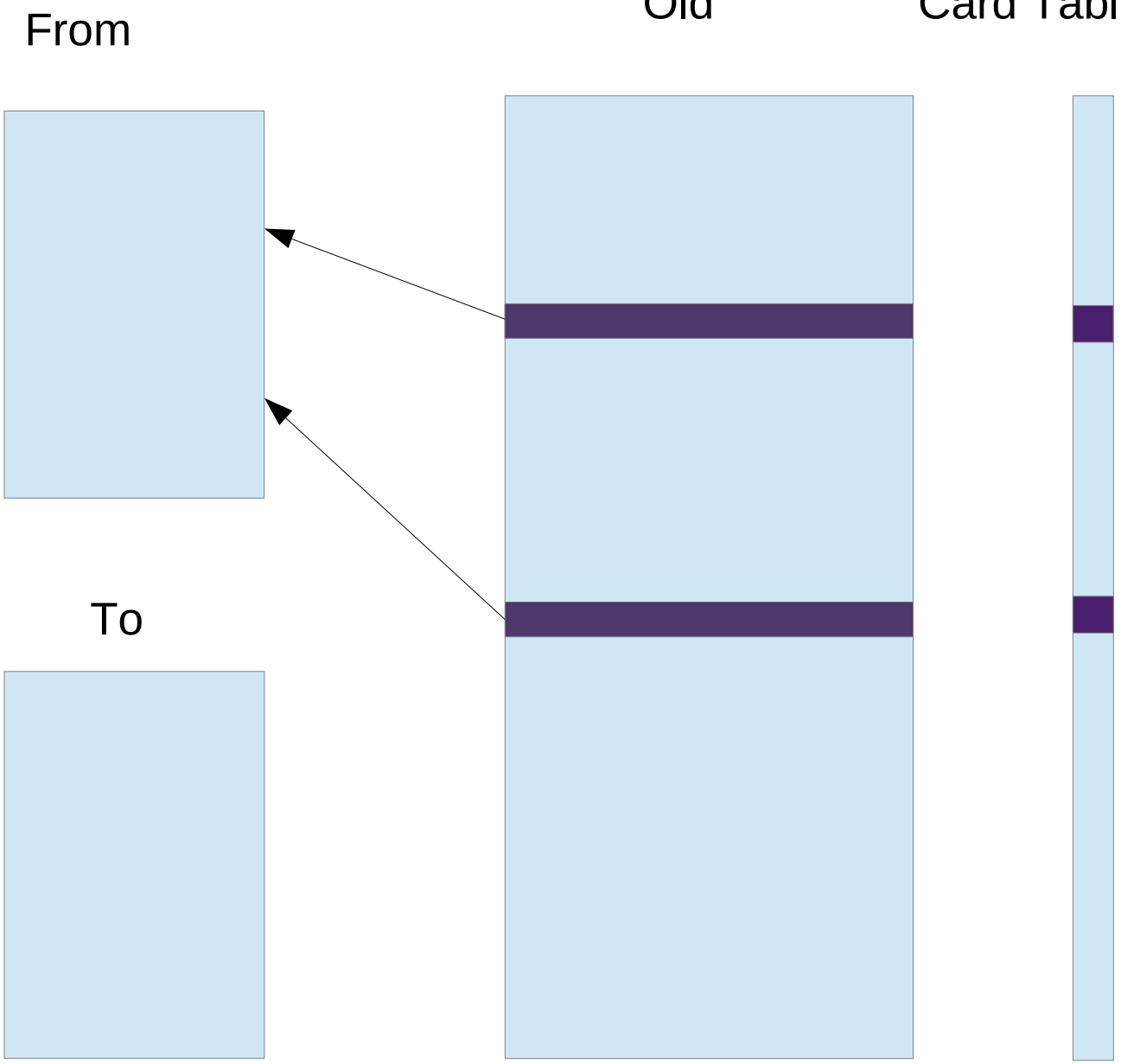
We chose forwarding pointers.

- No more remembered sets.
  - Can be concurrency bottleneck.
  - Can grow large.
  - In fact some claim that the benefit of generational collectors is smaller remembered sets, not better modeling of object lifetimes.
- No assumptions about object lifetimes.
  - Not all applications obey the generational hypothesis that most objects die young.
- No dependence on user or kernel level traps.
  - Software only solution.
- No read storms to update references.

Digression on remembered Sets

- Remembered sets allow you to collect part of your heap without collecting all of your heap.
- Generational garbage collectors usually use card tables.
- G1 uses into remembered sets.

Generational GC



Card table keeps track of old to young references so you can GC the young generation independently.

The old generation is broken into cards, and each card is represented by a single bit in the card table.

If that bit is set then the whole card must be scanned at young generation GC time.

Why is this a problem?

- Large multi-threaded applications which are carefully crafted to scale with padded data structures sometimes end up thrashing over the cache lines making up the card table.

Original G1

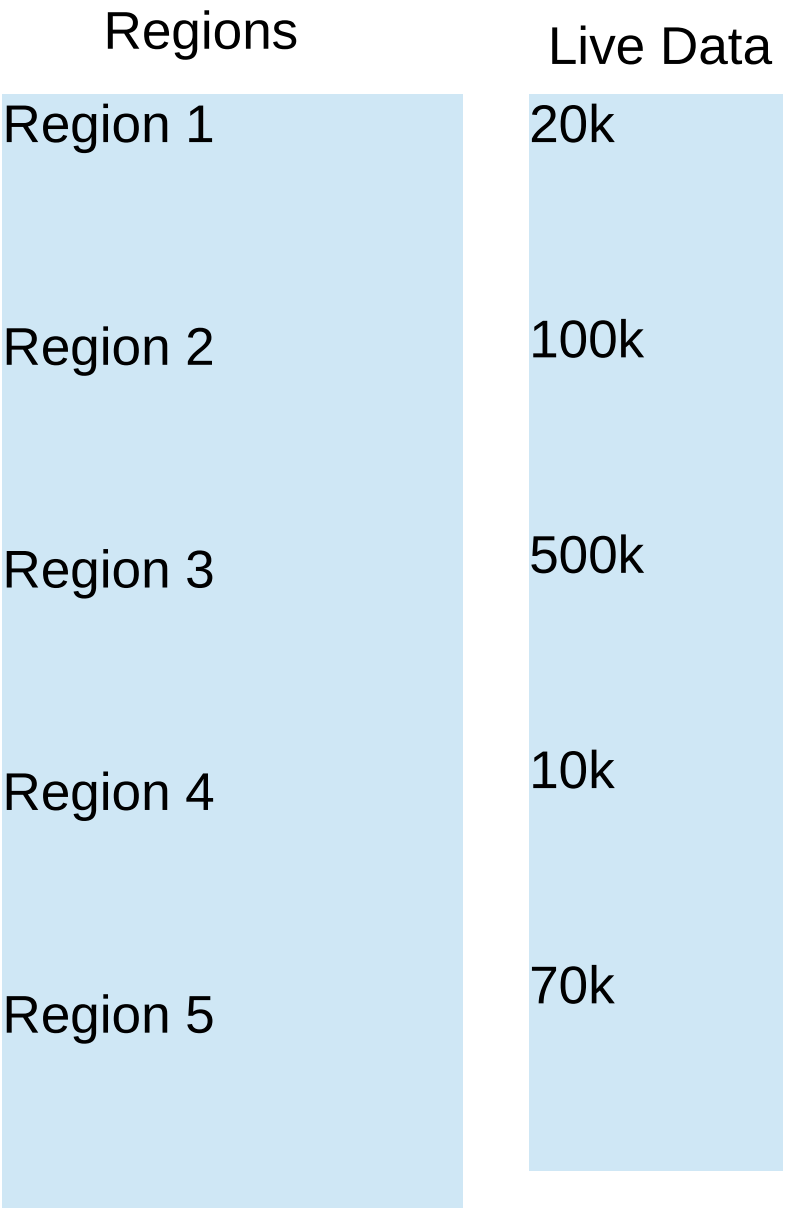
Remembered Sets	Regions	Live Data
Pointers into R1	Region 1	20k
Pointers into R2	Region 2	100k
Pointers into R3	Region 3	500k
Pointers into R4	Region 4	10k
Pointers into R5	Region 5	70k

G1 can independently collect whichever regions have the least live data.

Unfortunately into remembered sets can grow large.

This was made better by generational G1. They no longer needed to keep track of into pointers from young regions since they were guaranteed to be a part of the next collection. Unfortunately that forced G1 into a generational paradigm.

Shenandoah



Forwarding pointers enable Shenandoah to collect each region independently without remembered sets.

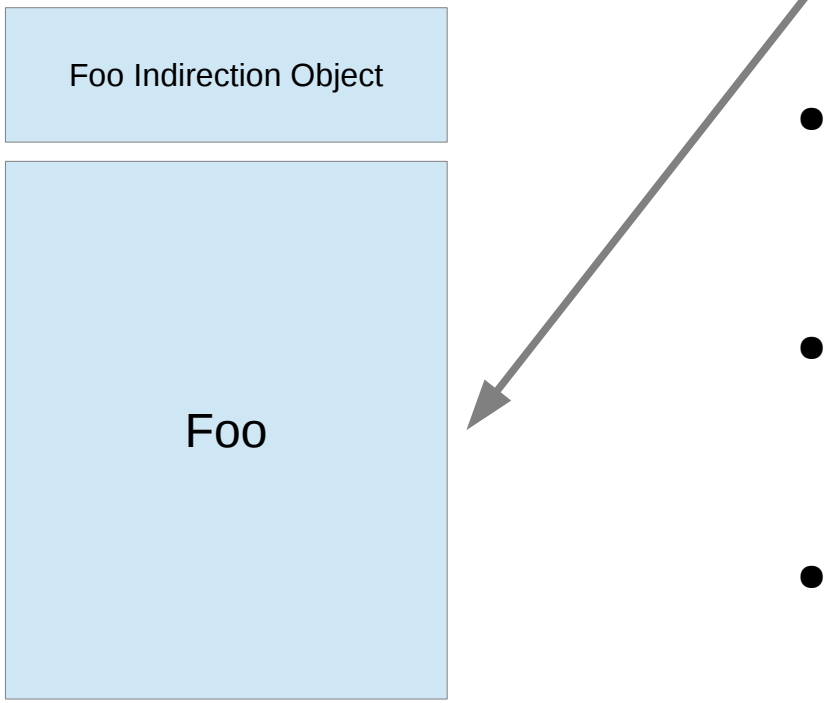
We truly are “garbage first”

Forwarding pointers based on Brooks Pointers

- Rodney A. Brooks “Trading Data Space for Reduced Time and Code Space in Real-Time Garbage Collection on Stock Hardware”

  1984 Symposium on Lisp and Functional Programing

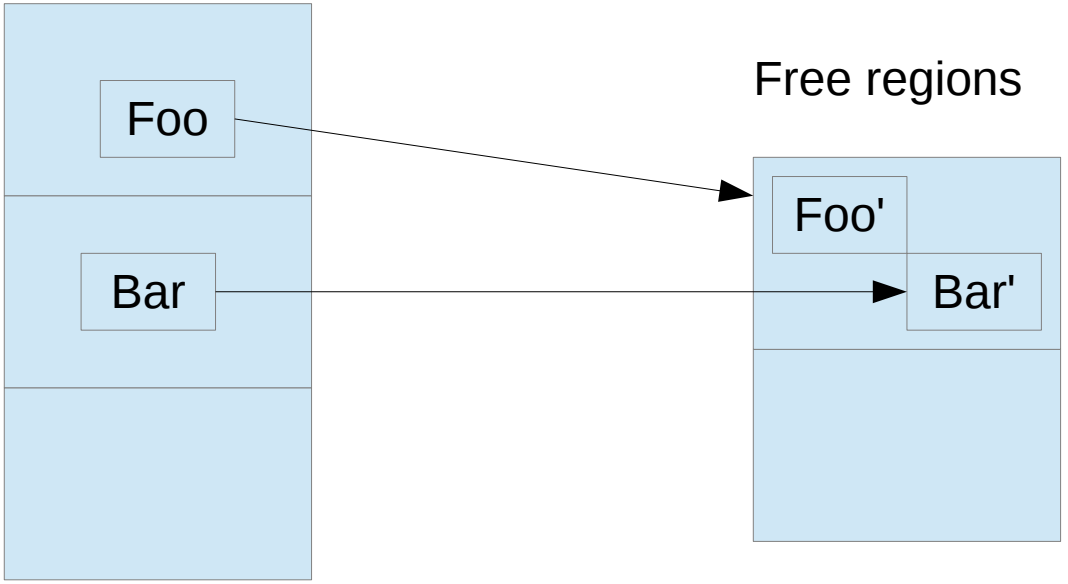
Forwarding Pointer in an Indirection Object



- Object Format inside the JVM remains the same.
- Third party tools can still walk the heap.
- Can choose GC algorithm at run time.
- We hope to one day be able to take advantage of unused space in double word aligned objects when possible.

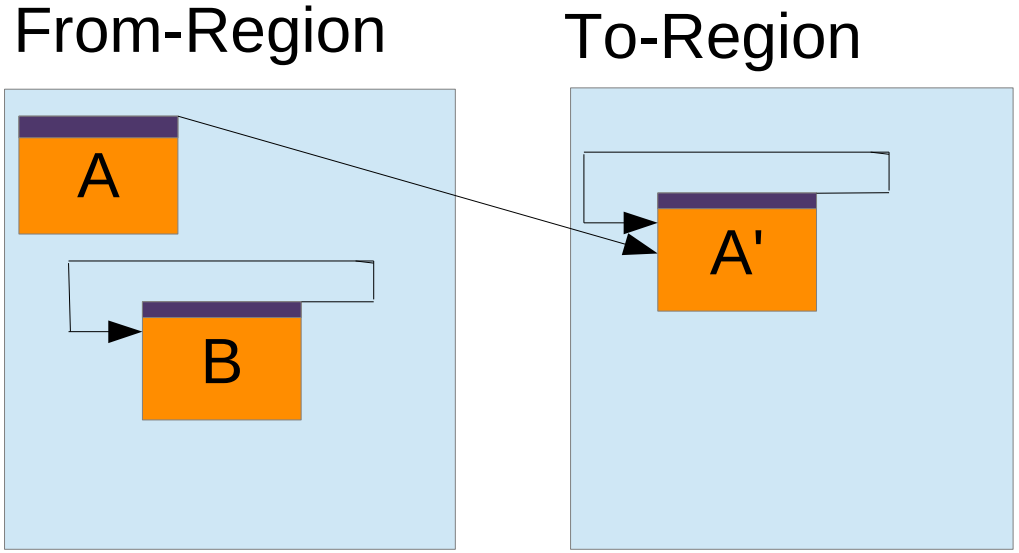
Adapted Brooks Pointers for regions.

Regions targeted for evacuation



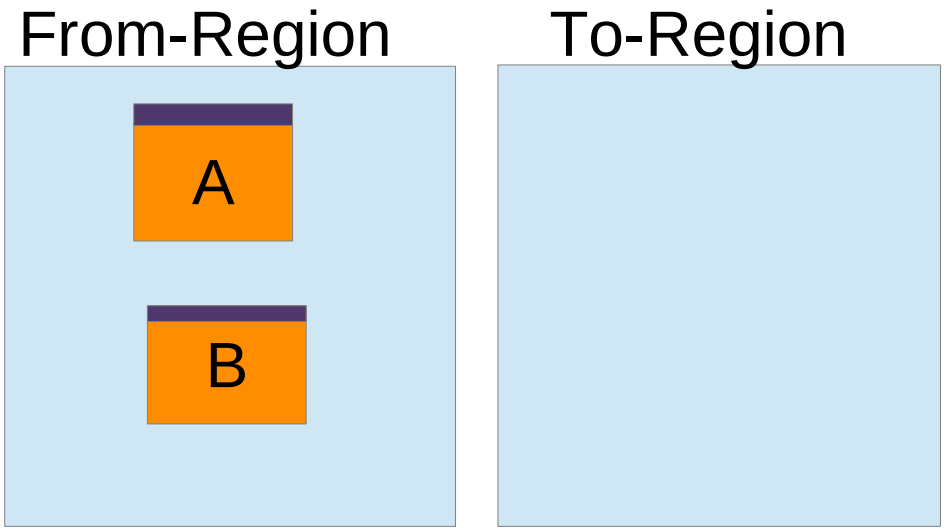
GC work is not tied to allocation work, instead dedicated GC threads evacuate regions.

Forwarding Pointers



Any reads or writes of A will now be redirected to A'

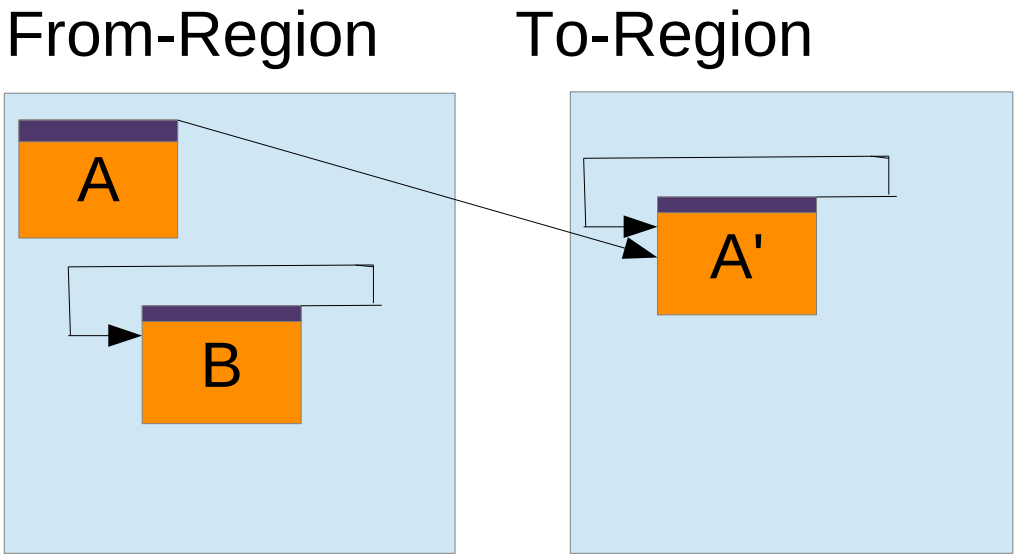
Forwarding Pointers



Reading an object in a From-region doesn't trigger an evacuation.

Note: If reads were to cause copying we might have a “read storm” where every operation required copying an object. Our intention is that since we are only copying on writes we will have less bursty behavior.

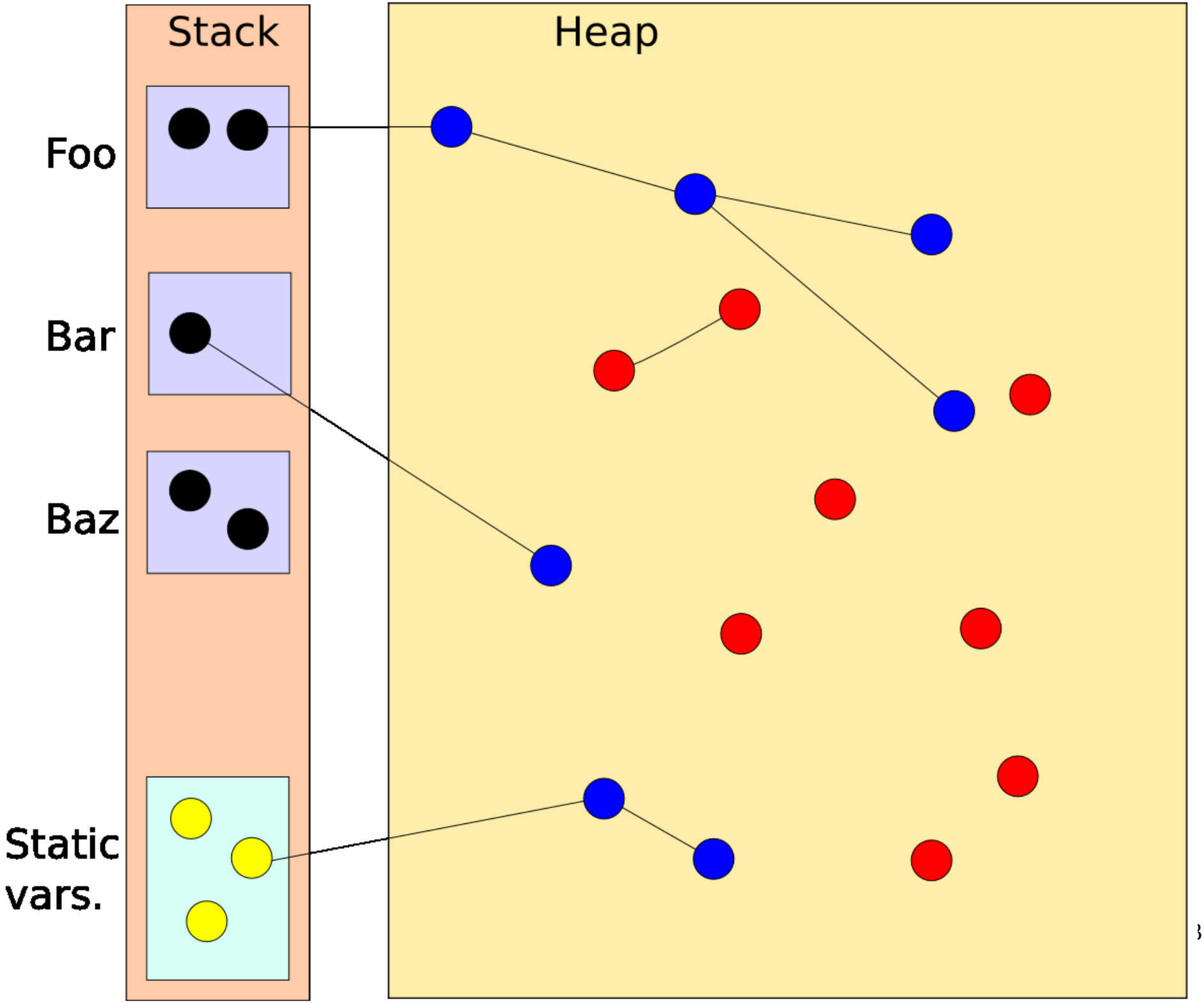
Forwarding Pointers

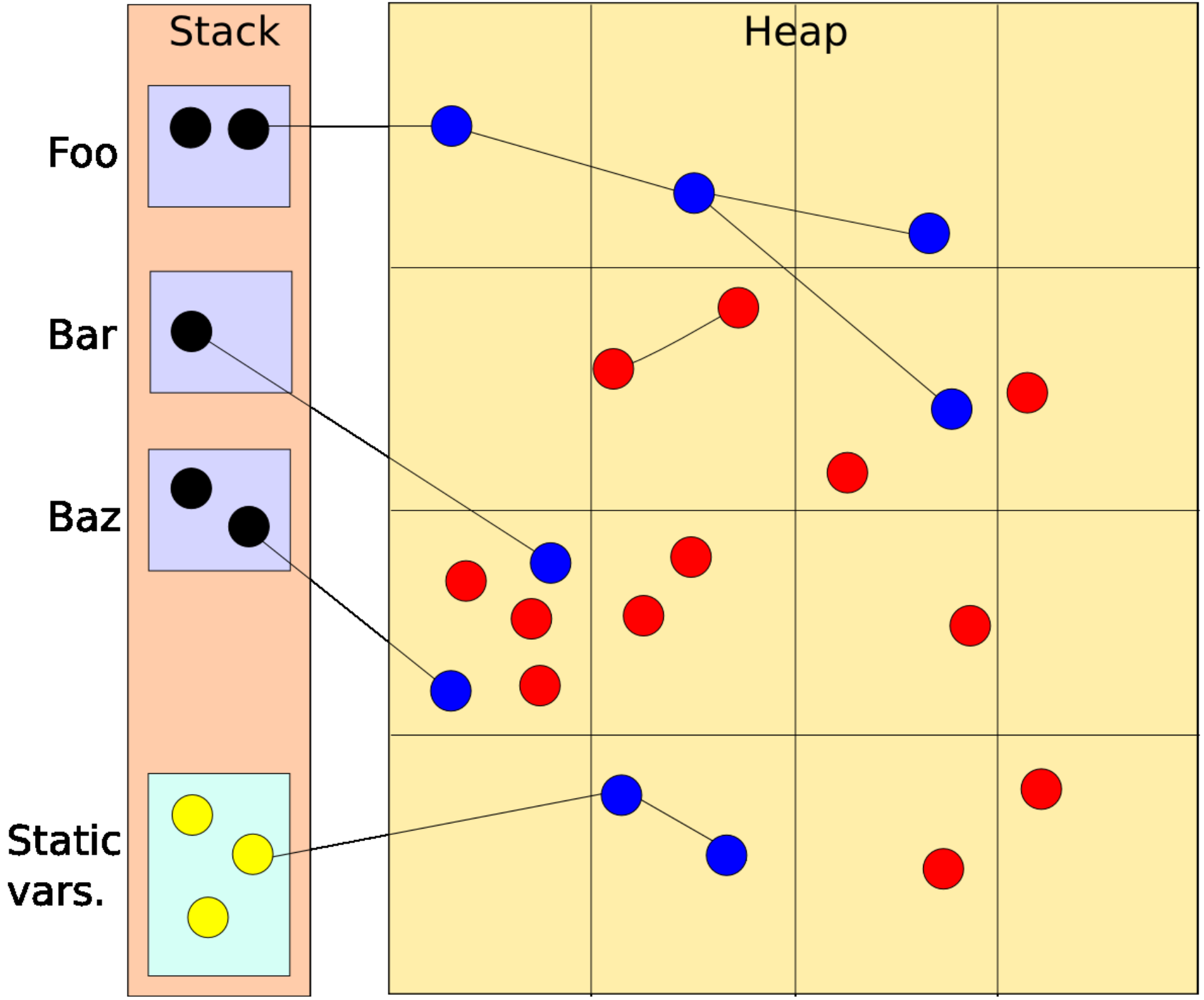


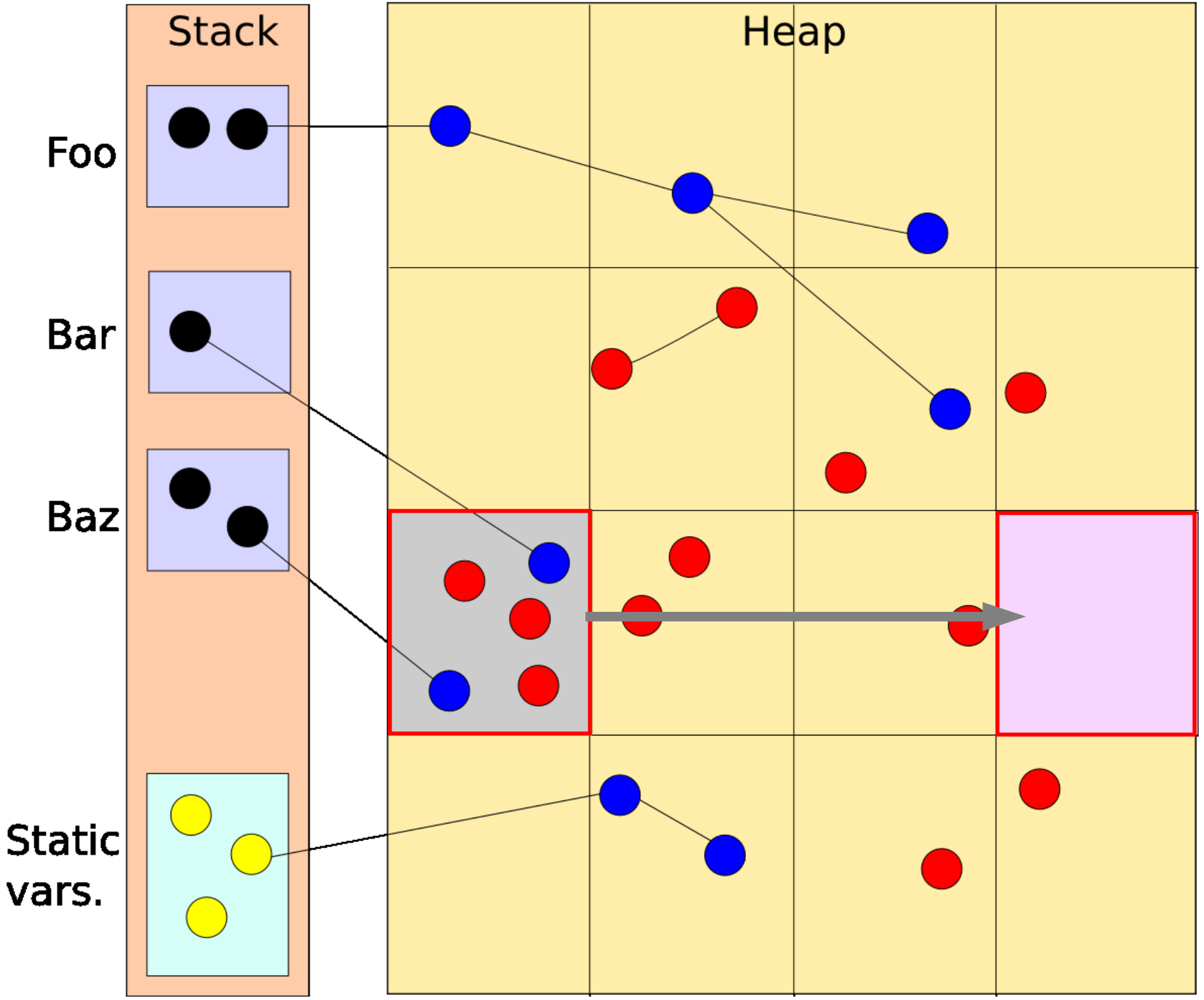
Writing an object in a From-Region will trigger an evacuation of that object to a To-Region and the write will occur in there.
This is necessary to keep one consistent copy of the object.

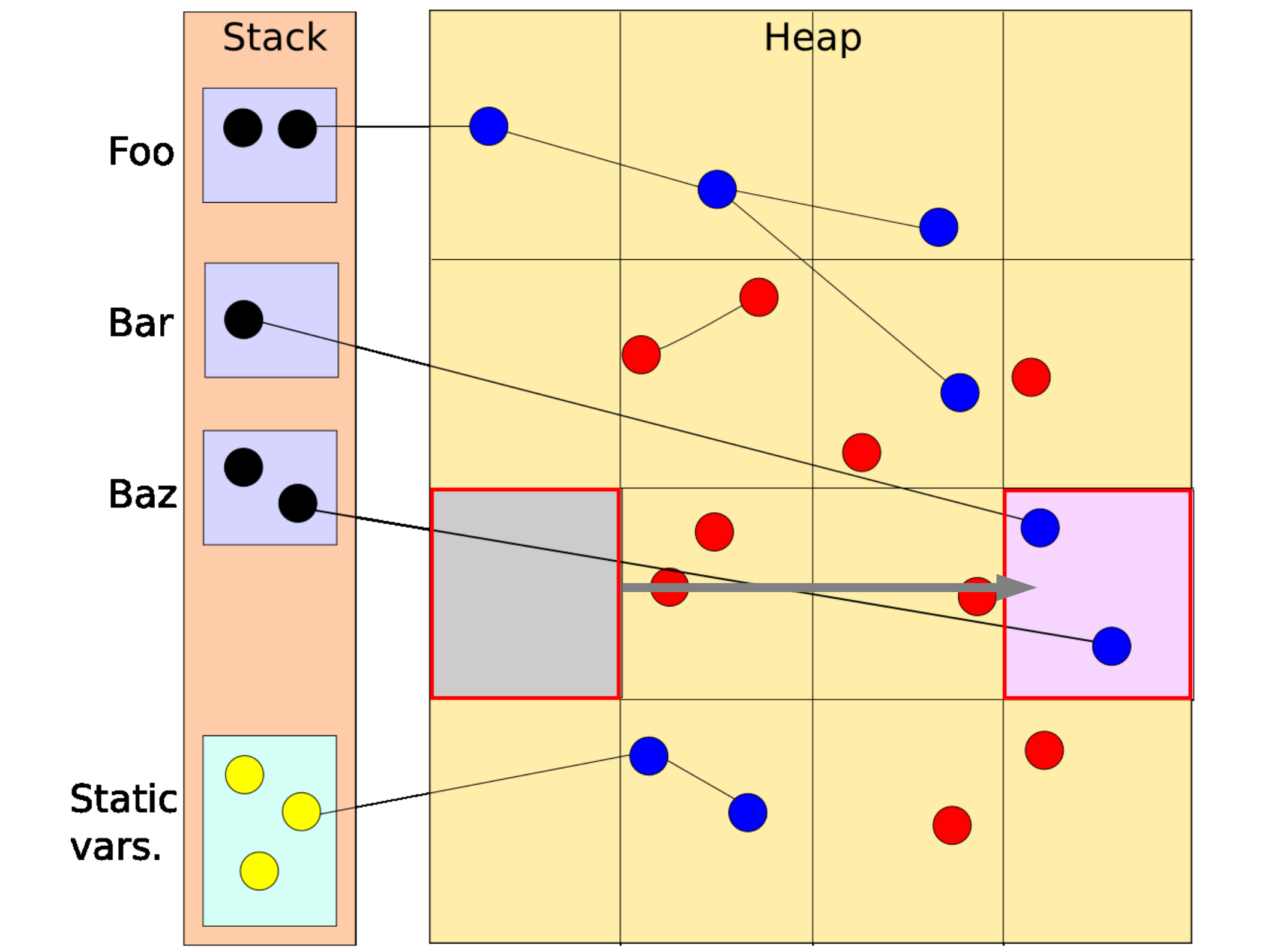
Shenandoah Algorithm

- Heap divided into regions.
- Concurrent marking keeps track of live data in each region.
- GC threads pick the regions with the most garbage to join the collection set.
- GC threads evacuate live objects in those regions.
- Subsequent concurrent marking updates all references to evacuated regions.
- Evacuated regions reclaimed.

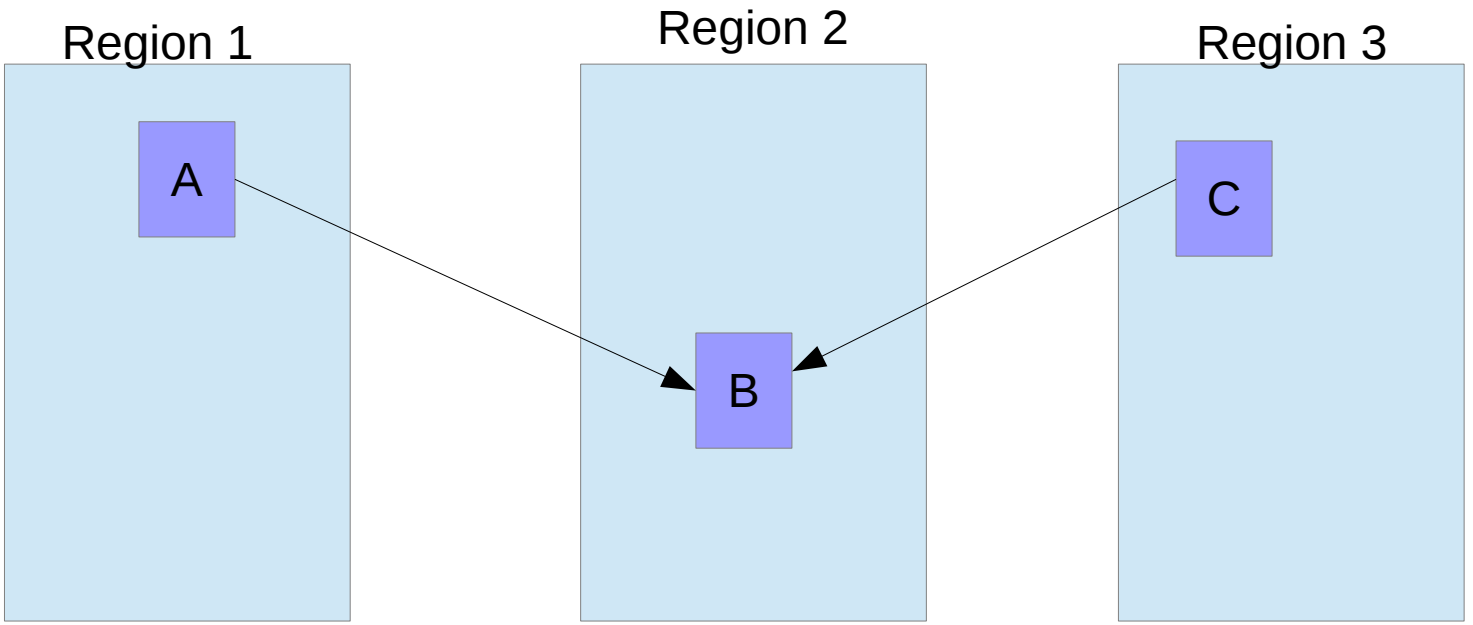

Stack
Heap
Foo
Bar
Baz
Static
vars.


Stack
Heap
Foo
Bar
Baz
Static
vars.


Stack
Heap
Foo
Bar
Baz
Static
vars.

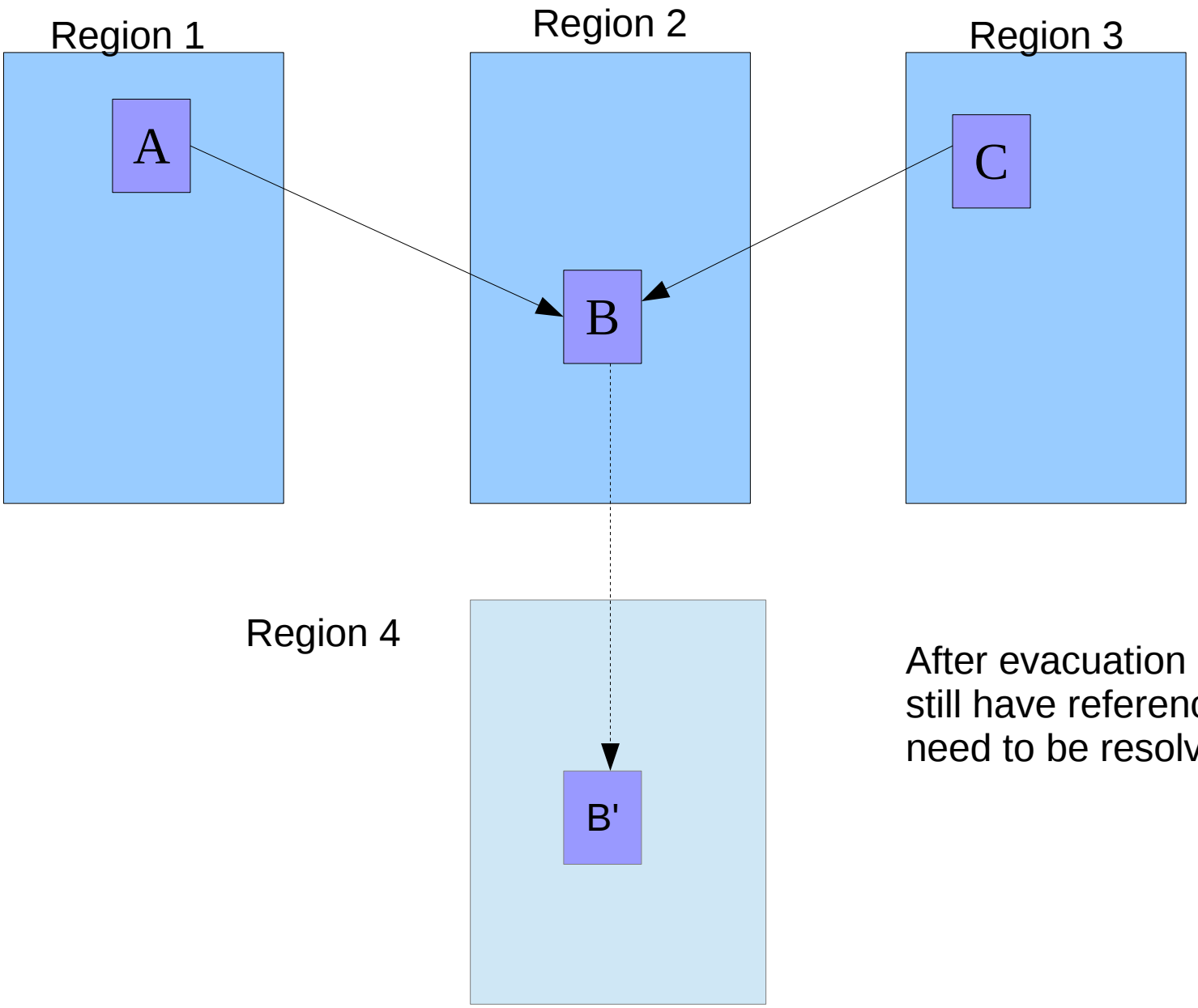

Stack
Heap
Foo
Bar
Baz
Static
vars.

References are updated lazily



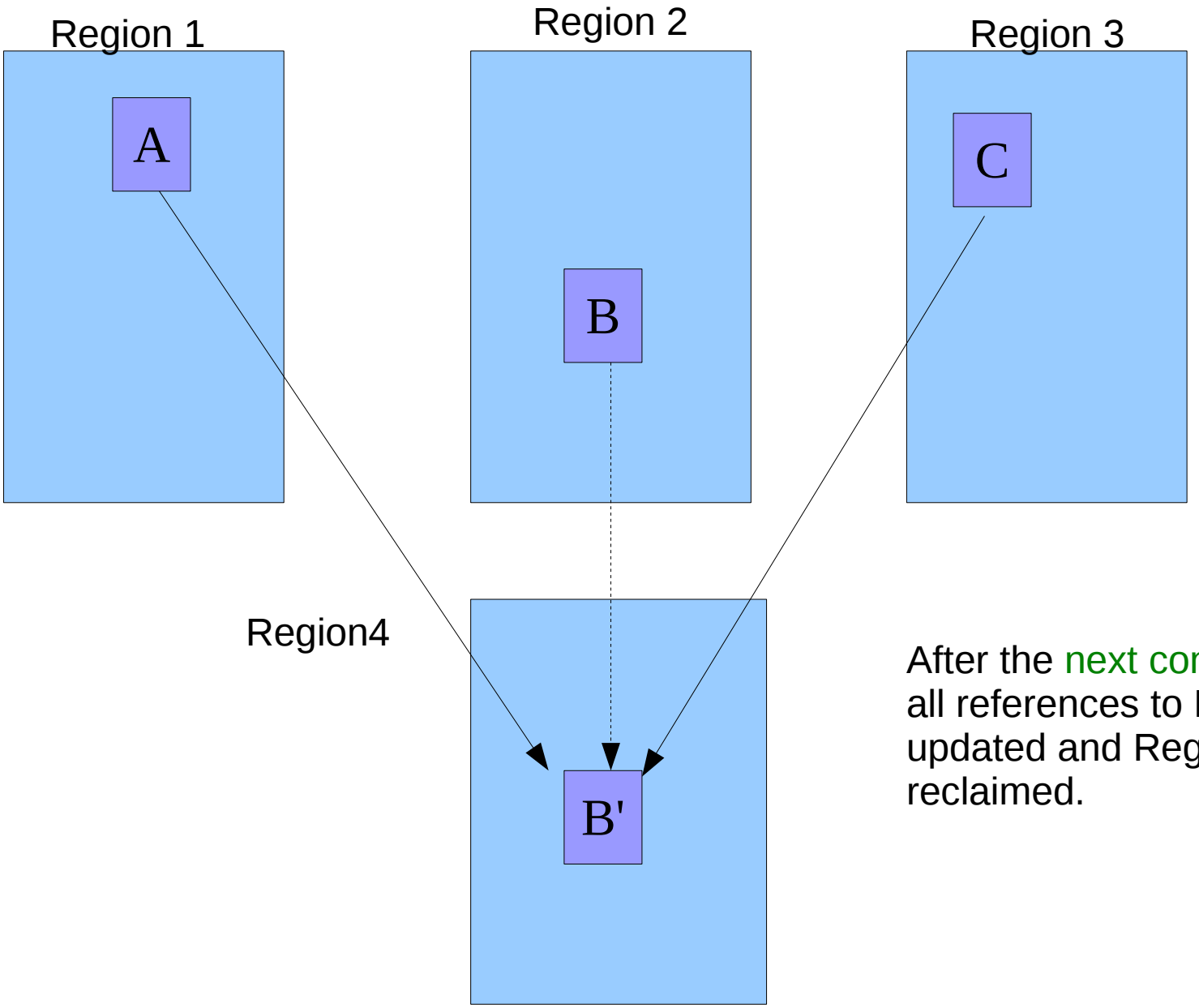
Region 2 is chosen to be part of the collection set.

After Evacuation



After evacuation is completed you still have references to B which need to be resolved when read.

After the next concurrent marking



After the next concurrent marking all references to B will have been updated and Region 2 may be reclaimed.

How does the GC interact with the Java Threads

- Reading is straightforward you simply indirect through the forwarding pointer. If the GC moved the object you see the new copy.
- Writing requires the Java thread to move the object. We need to ensure that writes only occur in to regions.

(Java thread!=GC thread)

Problem with writes in from-regions.

- GC thread copies Foo
- Java Thread updates Foo
- GC thread updates forwarding pointer to now obsolete copy of Foo.

All writes must occur in to-regions.

What if a Java Thread was writing an object when the GC moved it?

GC thread makes a copy

The GC thread and the Java thread both attempt to CAS the forwarding pointer to point to their copy of the object.
Only one can succeed.
The other thread must rollback their allocation.

Java thread makes a copy

What if a Java Thread was reading that object when the GC moved it?

- The Java thread either sees the old A or the new A' depending on whether the read barrier is executed before or after the move.

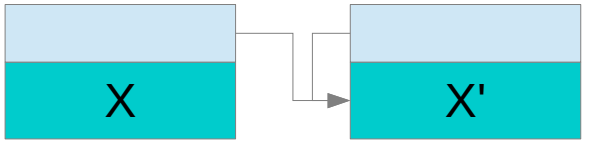
What could go wrong? Race windows get larger

- Before
  - Thread 1
    - read(a)
  - Thread 2
    - write(a).
- After
  - Thread 1
    - Resolve(a)
    - Read a
  - Thread 2
    - ResolveAndMaybeCopy(a)
    - Write(a)

Read Barriers

- Read (x)
  - Read X's forwarding pointer to get the current address for X.

- X has moved



- Y has not moved

Write Barriers

- Is X in the collection set?
  - Copy X to a new location.
  - CAS new address of X in X's Forwarding Pointer.
    - The CAS is to protect against other threads (Java or GC) attempting to move the object.
  - If the CAS fails, rollback the copy, and use the new value of the Forwarding Pointer as the address.

Why is Shenandoah worth it?

- Concurrent Evacuation
  - Greatly reduced pause times
- No more remembered sets
  - Card table marking can be a concurrency bottleneck.
  - Into remembered sets as in G1 can grow large and cumbersome
- Truly adaptive
  - If your application doesn't behave generationally you aren't saddled with a generational collector.

What we aren't telling you.

- There's a time overhead. Read and write barriers aren't free.
- There's a space overhead, especially when we are allocating an entire object (4 words) for our forwarding pointers.
- We are only just now at a point where we can start measuring those overheads.

What's left to do?

- Compiler support
- Very large (humongous) object support
  - If an object is larger than a heap region we need to be able to coalesce enough free space to allocate it.
- Performance heuristics
  - when do we start a concurrent mark?
- Round robin thread stopping instead of stop the world root scanning.
- Moving forwarding pointers into unused slots in the previous object if possible.
- Compressed oops?

More information

- Blogs
  - http://rkennke.wordpress.com/
  - http://christineflood.wordpress.com/
- Email
  - chf@redhat.com
  - rkennke@redhat.com

OK, Write barriers a little more complicated...

- We use Snapshot at the Beginning, so write barriers also need to keep track of previous values on writes to make sure everything that was live at the beginning of GC is still live.

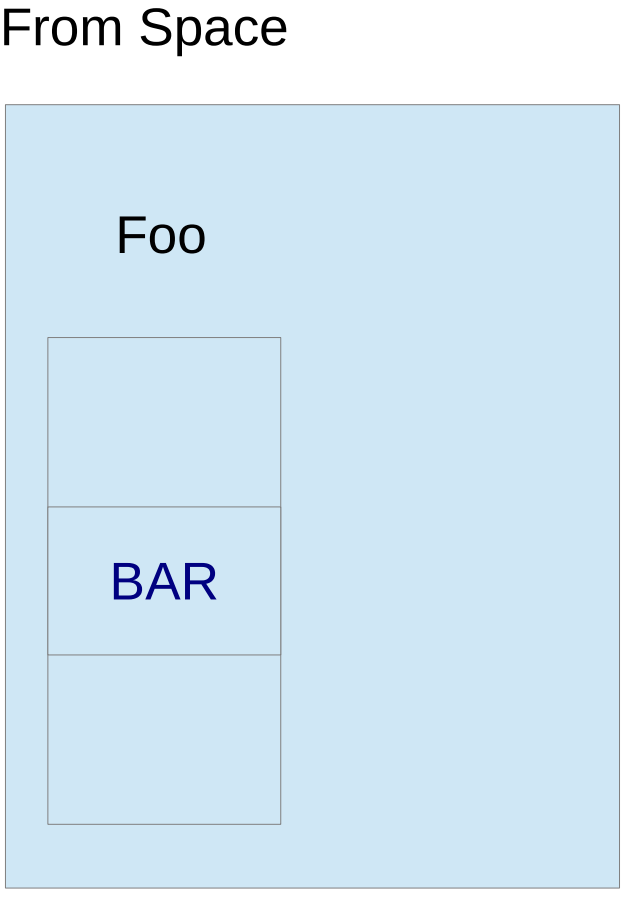
Issue with SATB

Start of Concurrent Marking

When a write occurs we keep track of the previous value to ensure that it gets marked.

Sometime during marking

Compare and Swap Object is complicated too.



- Compare and swap object is both a read and a write so it presents a special problem.
- If we want to CAS BAR to BAZ we first need to ensure that both FOO and BAR are in a to-region.
- If BAR is in a from-region than the CAS could fail because the GC updated BAR which isn't what we want.