Garbage Collection

Filip Pizlo

CS 565 Fall 2008 | W. Lafayette, IN

- Programs may choose different strategies for memory management:
  - Static
  - Dynamic but unsafe
  - Dynamic but safe (Garbage Collection)

Static Memory Management

- Statically preallocate everything.
- Use “pools” for “static allocation”.
- Conceptually easy to implement.
- Works in every language.
- *Hinders component reuse and modularity.*
- *Often hard to get right.*

Unsafe Dynamic Memory Management

- Allocate when you need it, free when you don't.
- Conceptually hard to use.
- Makes the language inherently unsafe.
- *Impact on predictability.*
- *Often hard to get right.*

Garbage Collection

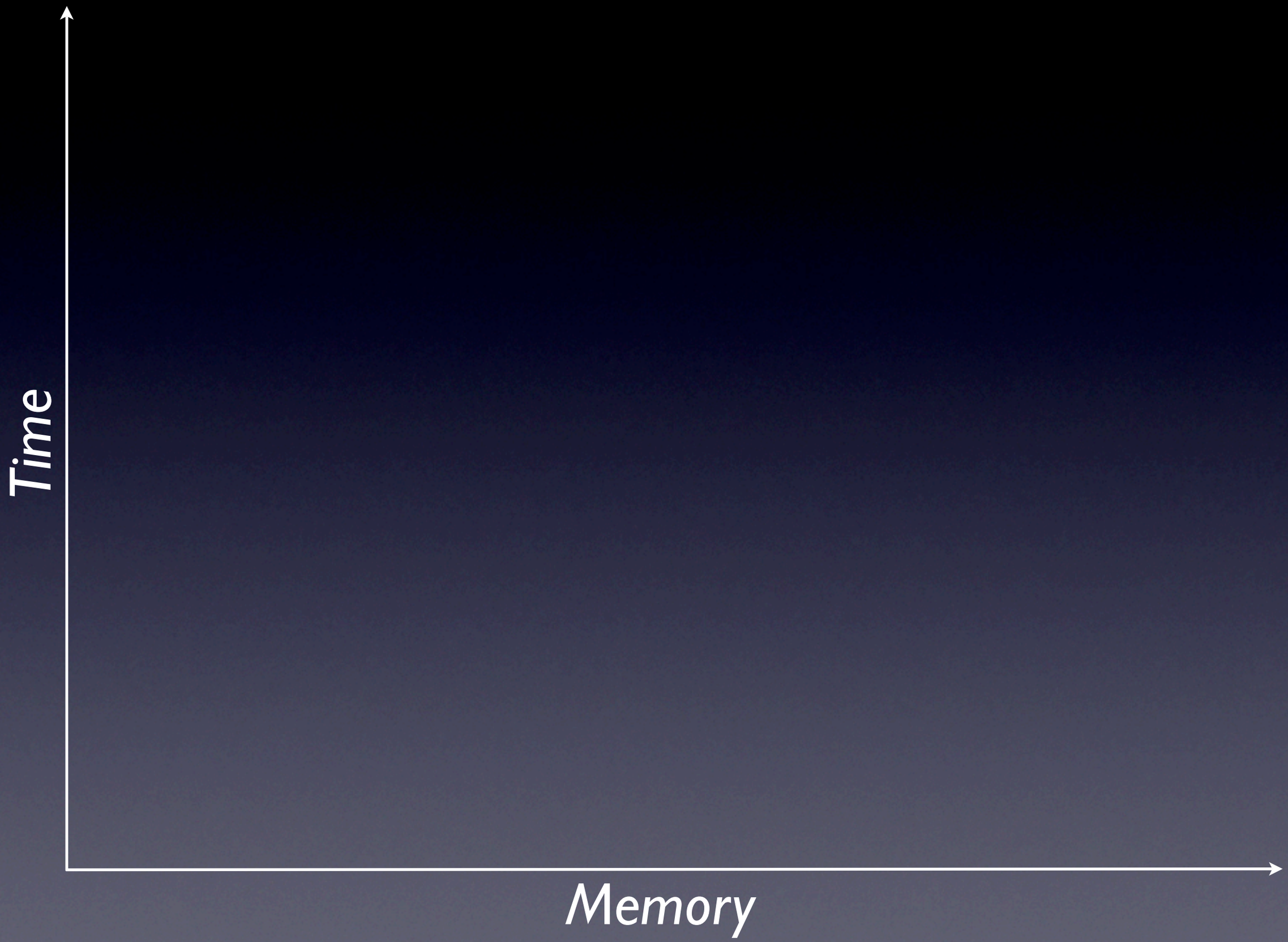
- Allocate when you need it, *let the system take care of it when you don't.*
- Really easy to use.
- Makes the language memory-safe.
- Offers excellent component reuse and modularity.
- *We (think) we know how to make it predictable.*
- *May hurt performance.*

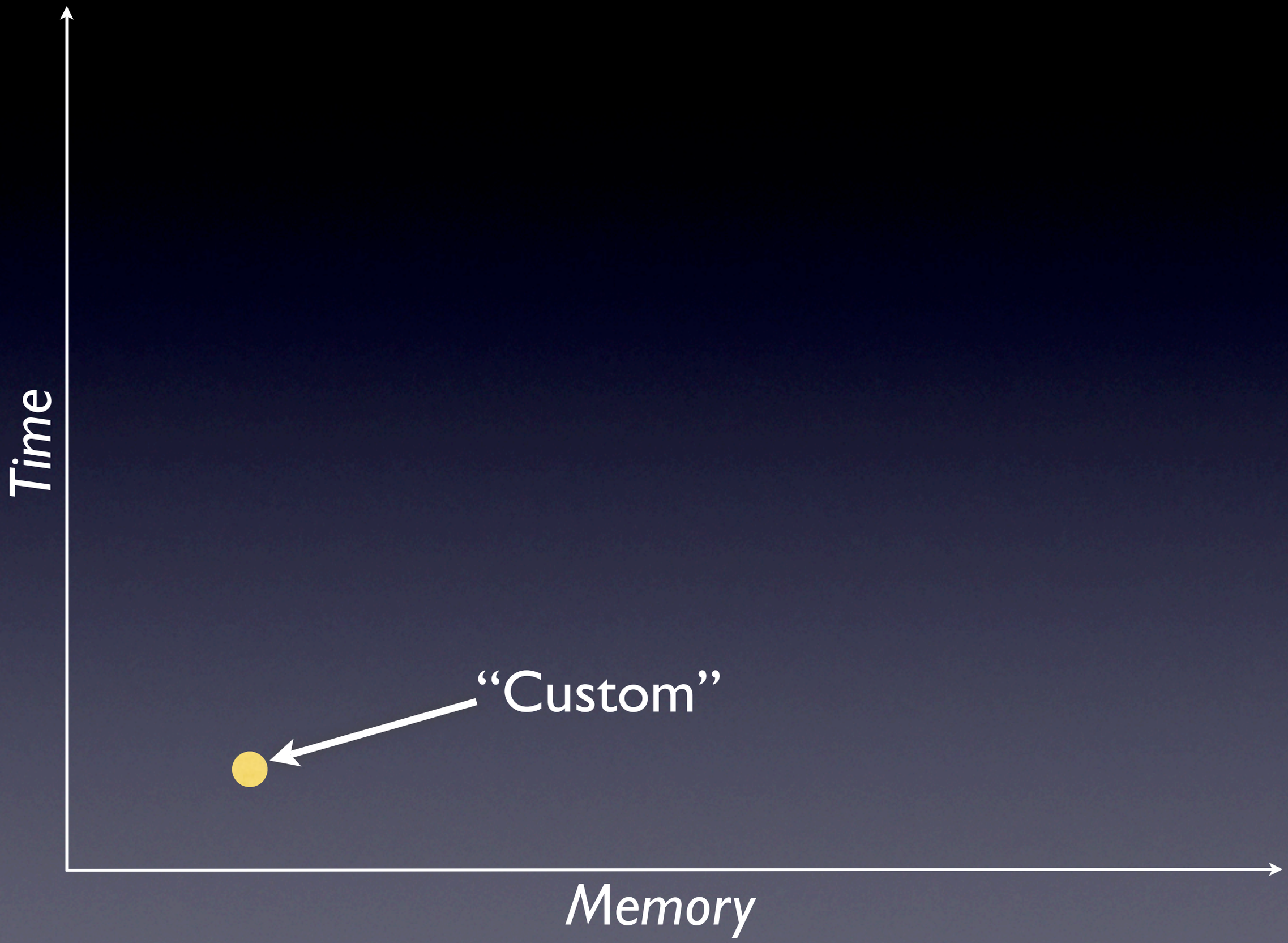
- But how do these approaches really compare when it comes to performance?

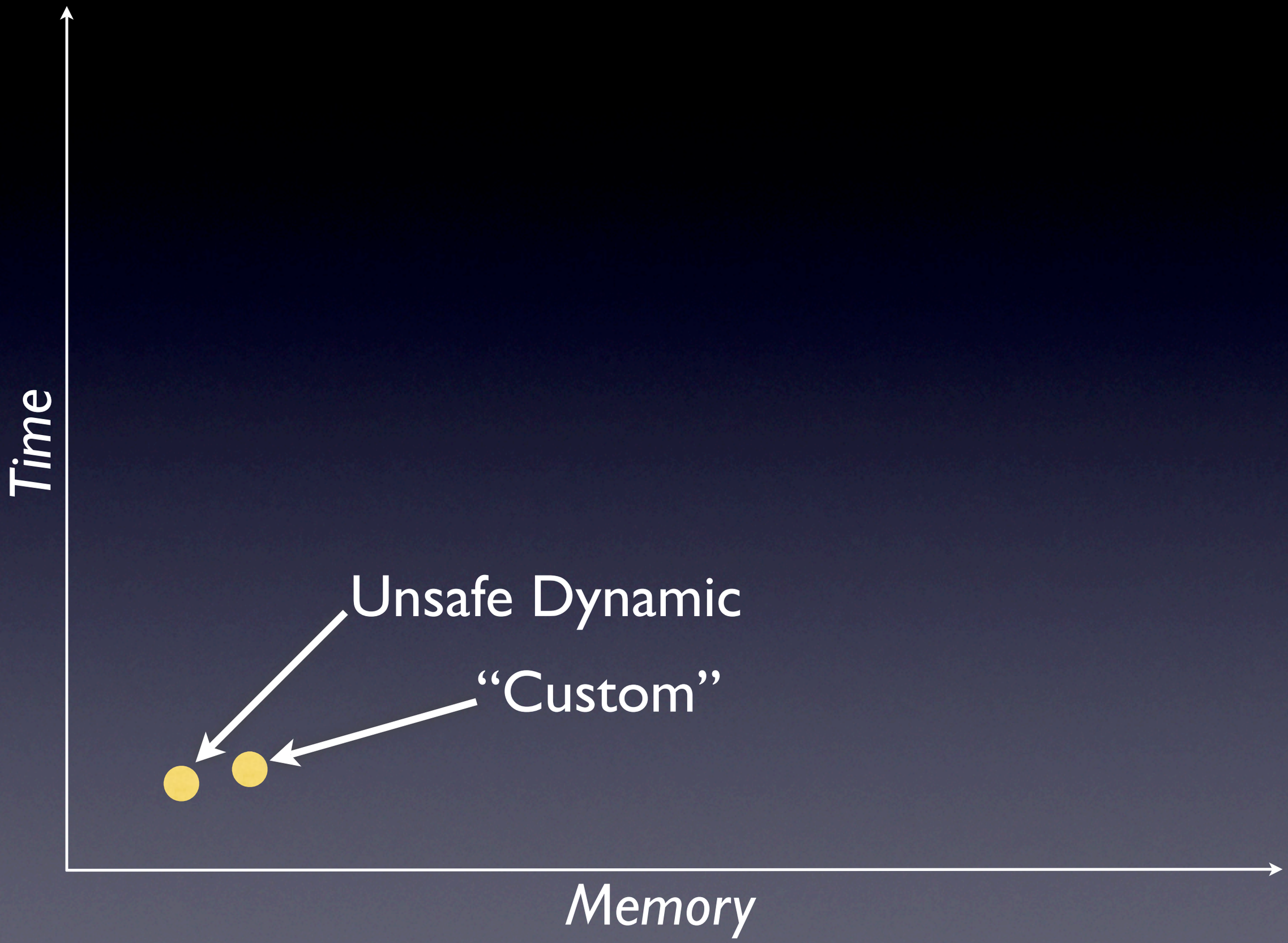
- Berger et al (OOPSLA 2002) asked the question: how does **"custom"** memory management compare to dynamic memory management?

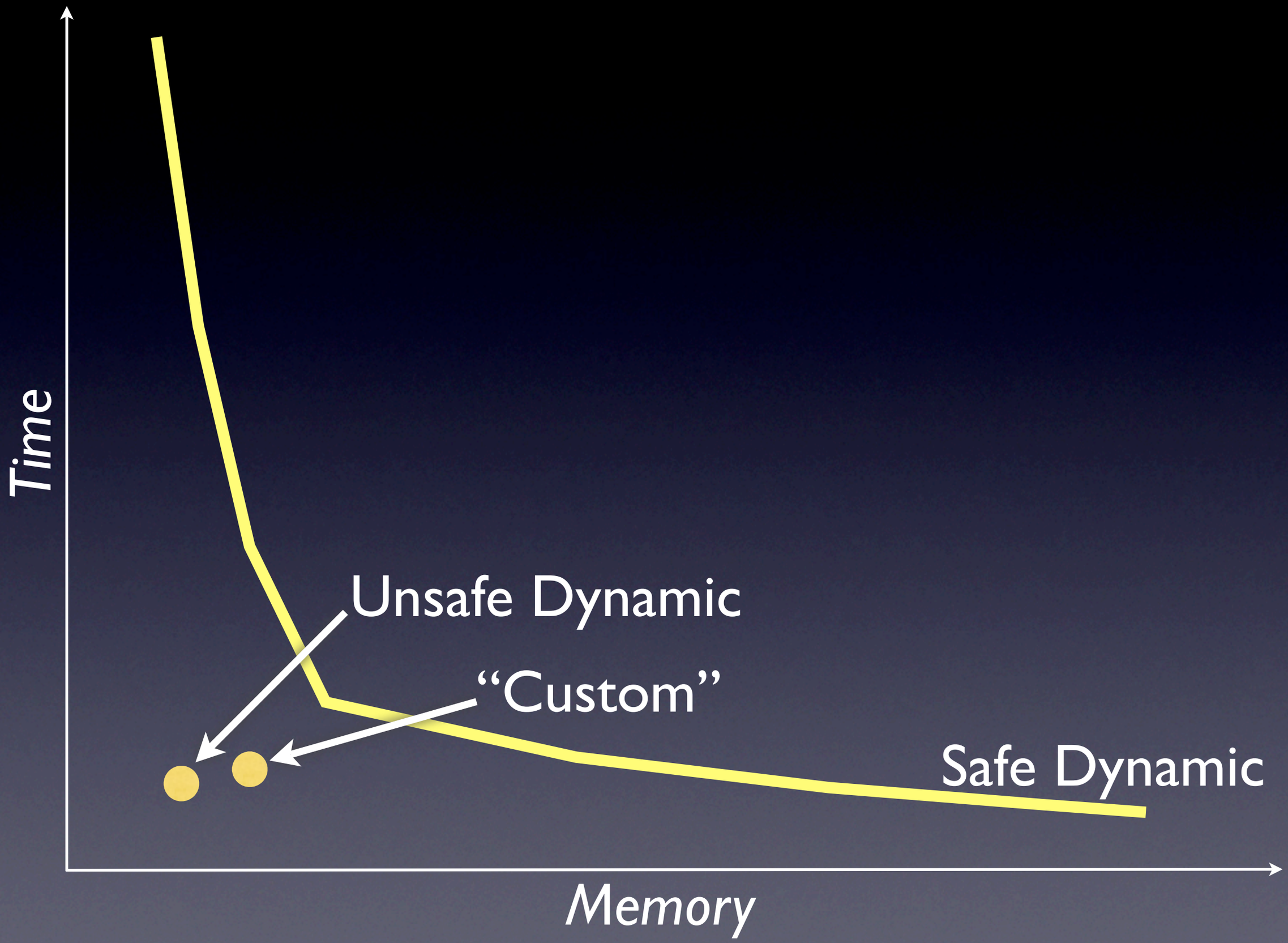
- Hertz et al (OOPSLA 2005) compared unsafe dynamic memory management to garbage collection.

- In the Berger et al study, "Custom" Memory Management included:
  - Regions
  - Object pools - *"static allocation"*

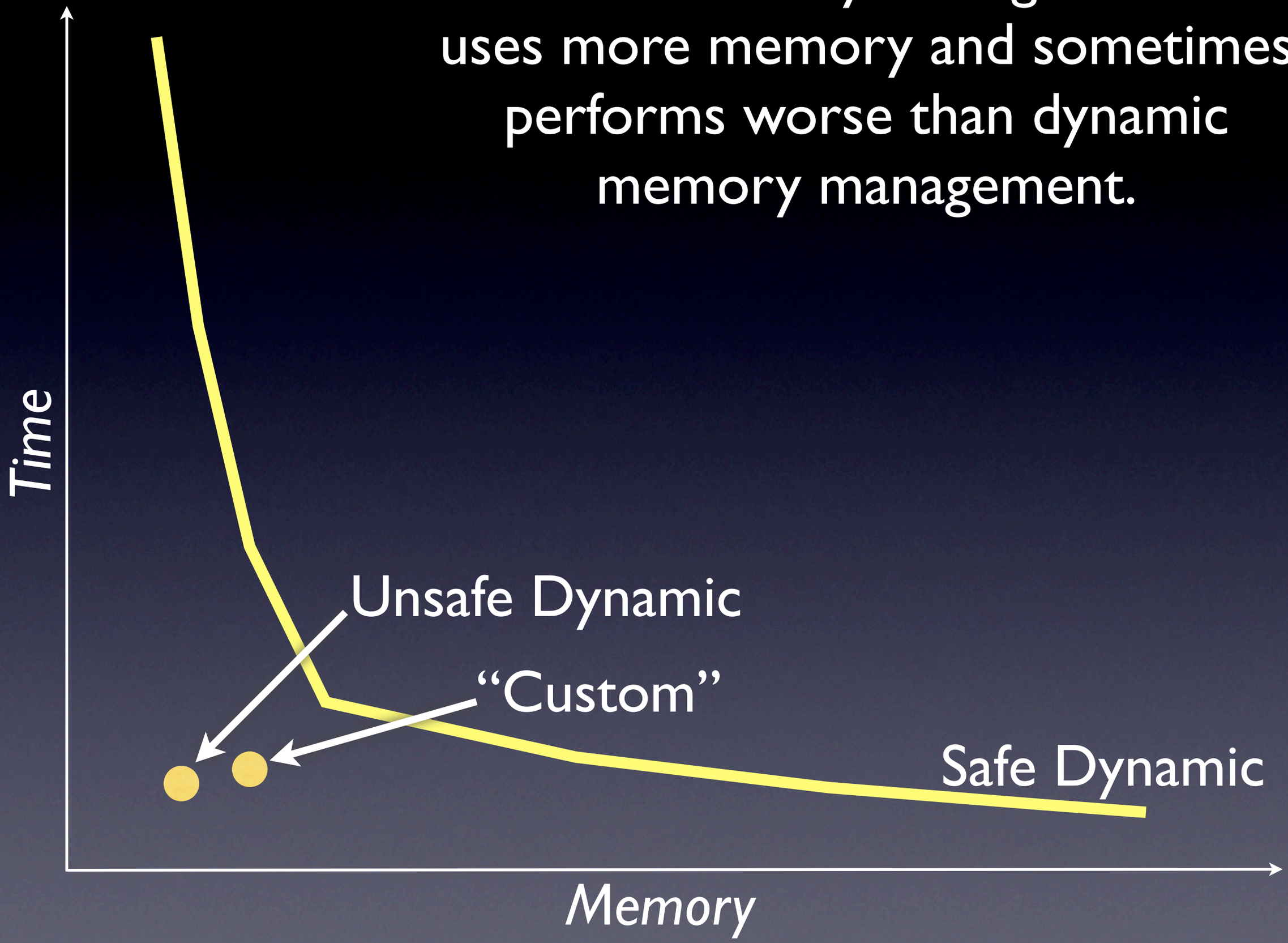

Time
Memory


Time
"Custom"
Memory


Time
Unsafe Dynamic
“Custom”
Memory

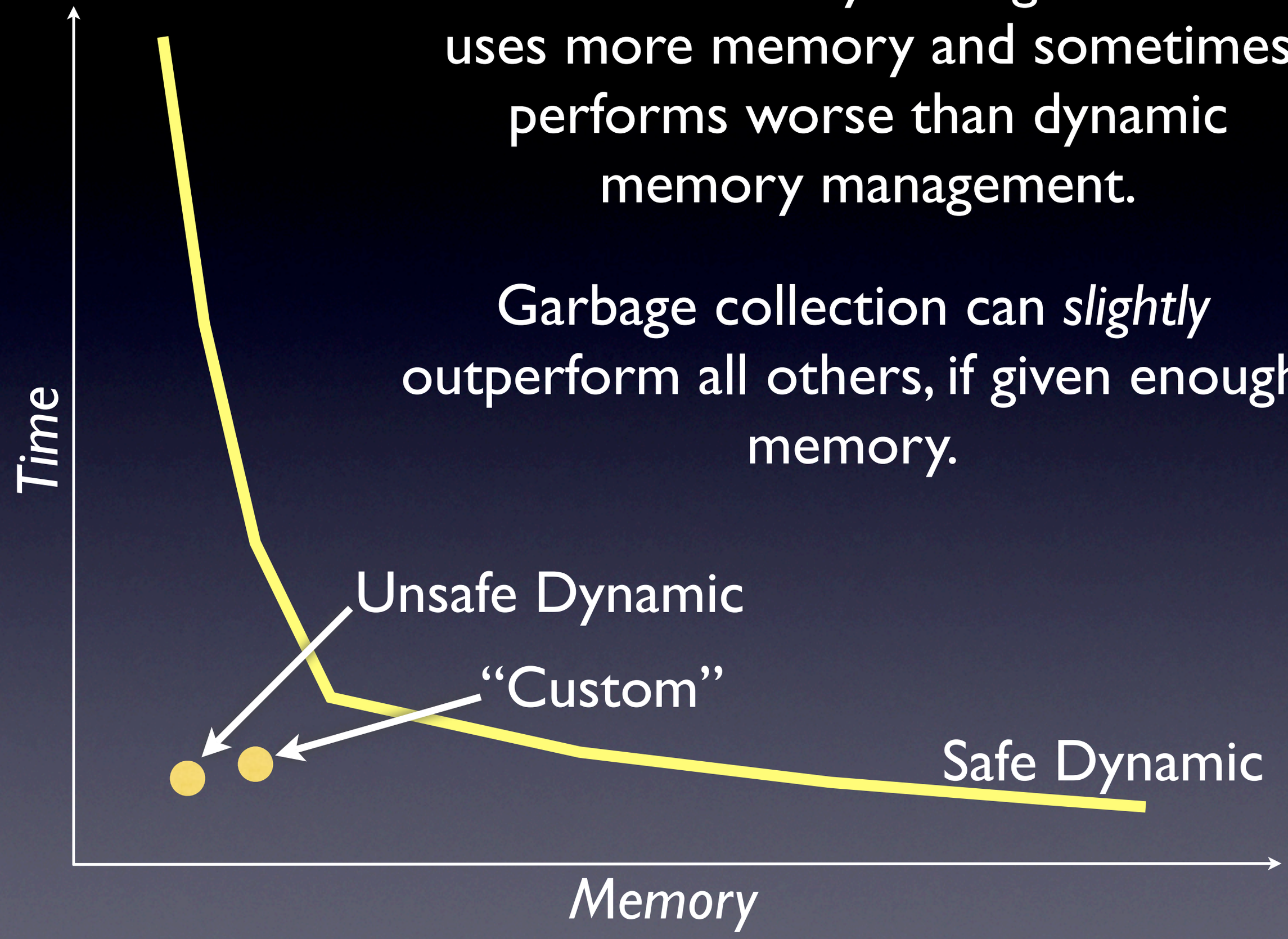

Time
Unsafe Dynamic
"Custom"
Safe Dynamic
Memory

"Custom" memory management often uses more memory and sometimes performs worse than dynamic memory management.



"Custom" memory management often uses more memory and sometimes performs worse than dynamic memory management.

Garbage collection can *slightly* outperform all others, if given enough memory.



- For details:
  - Berger, Zorn, McKinley (OOPSLA 2002)
  - Hertz, Berger (OOPSLA 2005)

GC: what is it?

- Modern programs allocate memory in variable-size chunks ("objects") dynamically.
- Unused objects must be returned to the system to allow for new allocations.
- Garbage collectors, or "GCs", find unused objects and return prepare them for reuse for new allocations automatically.

- GCs make programming easier, because they allow the programmer to mostly ignore object lifetimes.

- But GCs hinder performance, since the collector must at some point assess the state of the heap and reclaim objects.

- State-of-the-art collectors may in the worst case perform an O(heap size) operation to free memory on an allocation request that would exhaust the heap.

- Such behavior is undesirable for systems in which execution time is part of the specification (so-called "real-time" systems).

The GC Problem

- GCs must find those objects in the heap that are no longer in-use.
- What does it mean to be in-use?
  - Modern GCs view an object to be in use if it is reachable to the application.
- But how do we determine reachability?

GC: two approaches

- Reference Counting:
  - Every time a new reference to an object is made, increment a counter for that object. Decrement it when the reference is broken.
- Tracing:
  - Perform a graph search starting at the "roots" - registers, global variables, thread stacks, and other memory locations not subject to GC reclamation.

GC: two approaches

- Reference Counting:
  - Generally quite slow (because of increments/ decrements), doesn't reclaim object cycles, .
- Tracing:
  - Requires a graph search. Naively, graph searching requires the graph to be constant. Meaning: the application must not modify the heap while the tracing is on-going.

GC: two approaches

- Reference Counting:
  - Often gives good responsiveness but at a huge cost to throughput.
- Tracing:
  - Often involves one big pause.

GC: two approaches

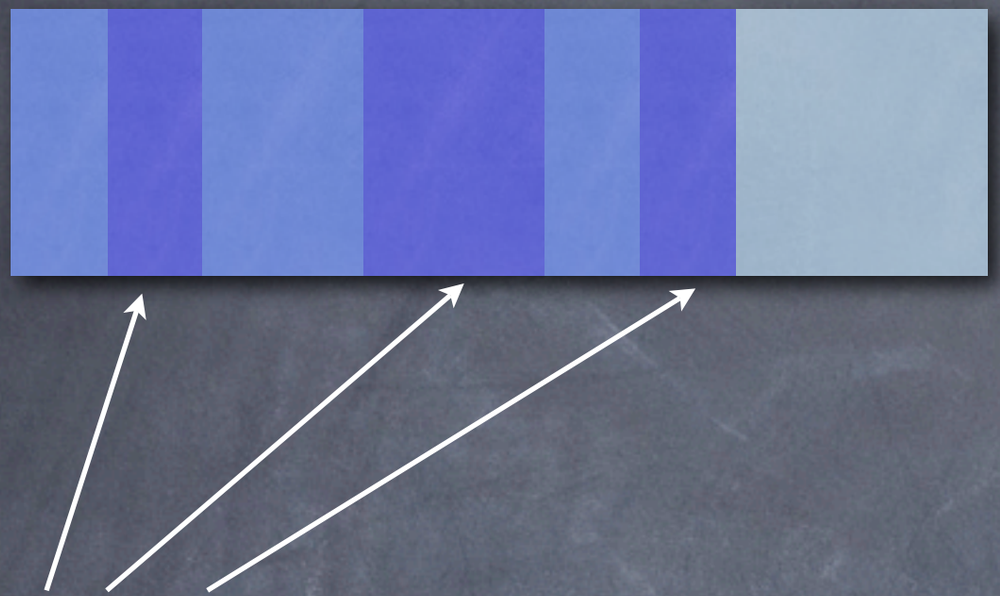
- Reference Counting:
  - By itself, does not defragment the heap.
- Tracing:
  - Heap defragmentation is possible, since tracing can be retrofitted to also move objects.

- Modern GCs use the tracing technique because:
  - Ref counting (RC) is slow unless "deferred RC" is used (which defers book-keeping and deallocation work to one big pause triggered when memory runs out). Hence - responsiveness isn't any better.
  - RC doesn't reclaim all objects; thus, modern RC collectors fall back on tracing anyway when memory runs out.
  - Tracing allows defragmentation.
- In short: you need tracing anyway in the worst case, so RC just leads to an unnecessary overhead.

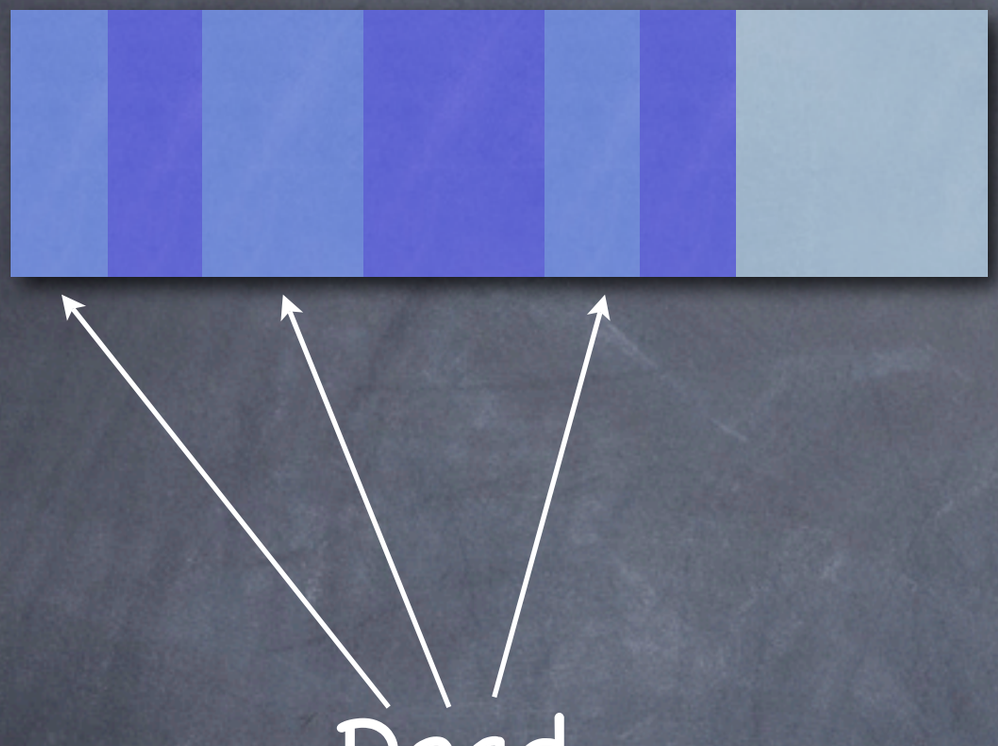
Tracing GC review

- Three activities typically make up tracing collectors:
  - Marking - while tracing the heap we "mark" objects that are in use.
  - Copying - we "copy" objects to new locations to reduce fragmentation.
  - Sweeping - we locate the parts of the heap that are not used and organize them for reuse in the "sweep" phase.

- Main collector types:
  - Mark-Sweep: one phase to mark used objects, another to sweep unused space. No copying is performed.
  - Mark-Compact: one phase to mark used objects, another to copy all used objects to one side of the heap.
  - Copying: one phase to simultaneously mark and evacuate all used objects; the space that previously held all objects is then completely free.



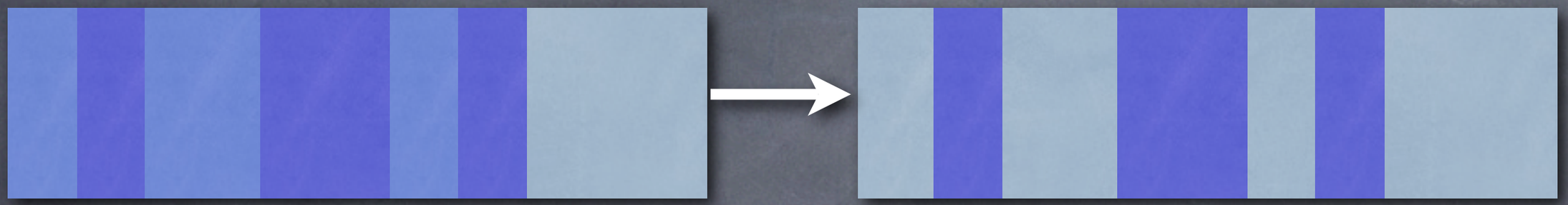
In-use



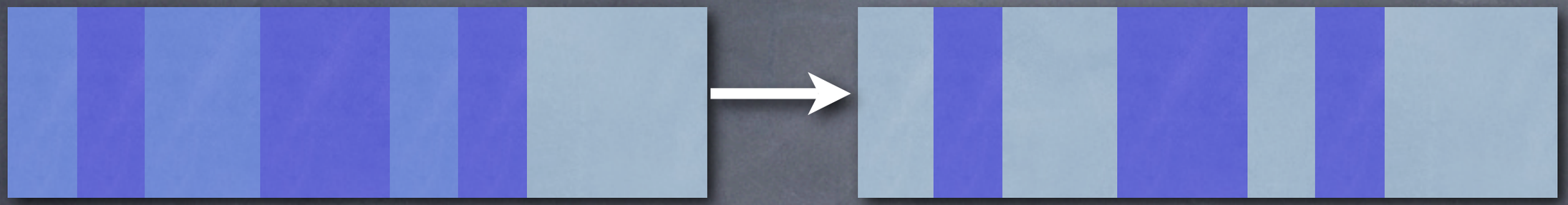
Dead

Free

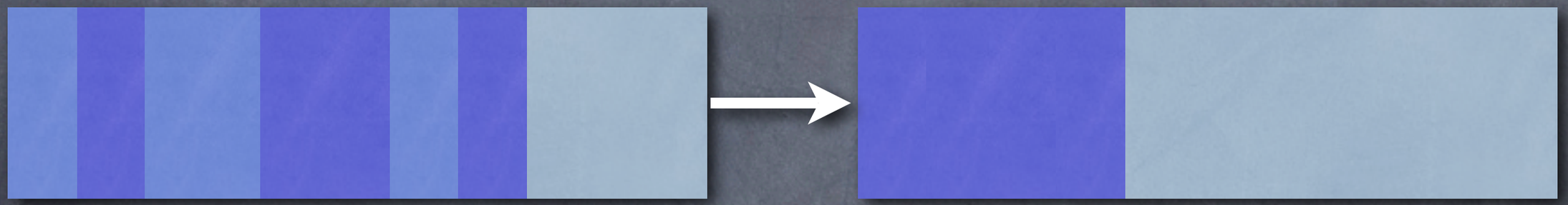
Mark-Sweep



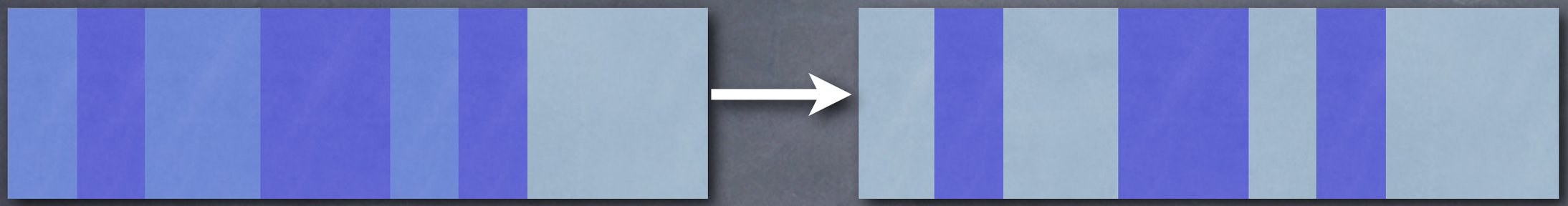
Mark-Sweep



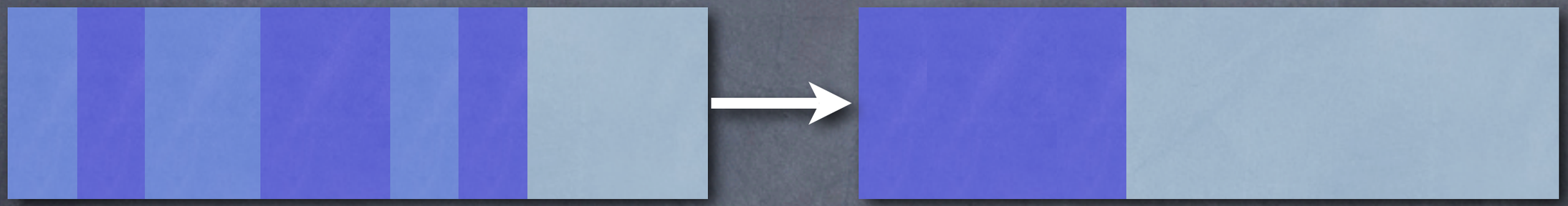
Mark-Compact



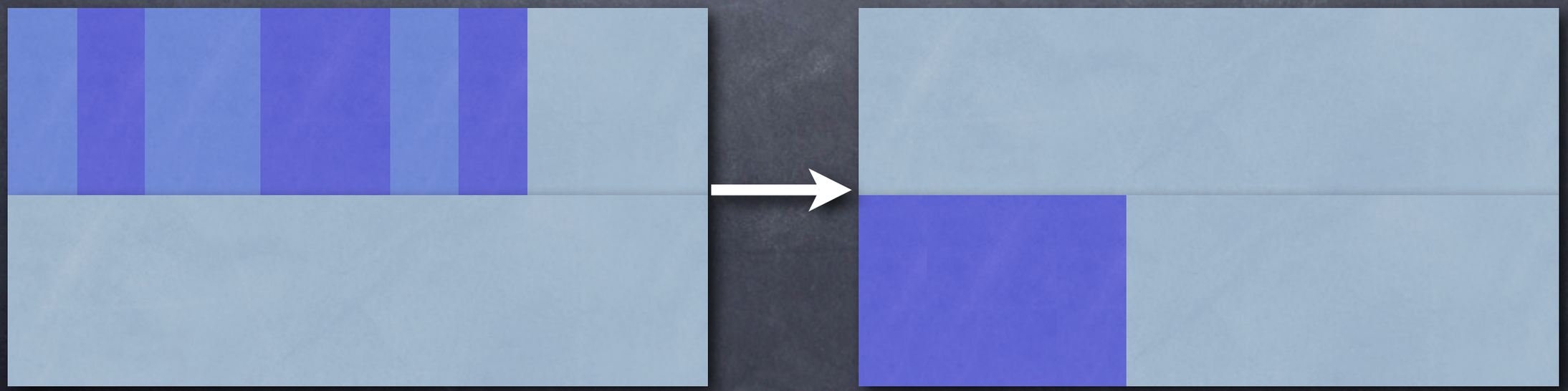
Mark-Sweep



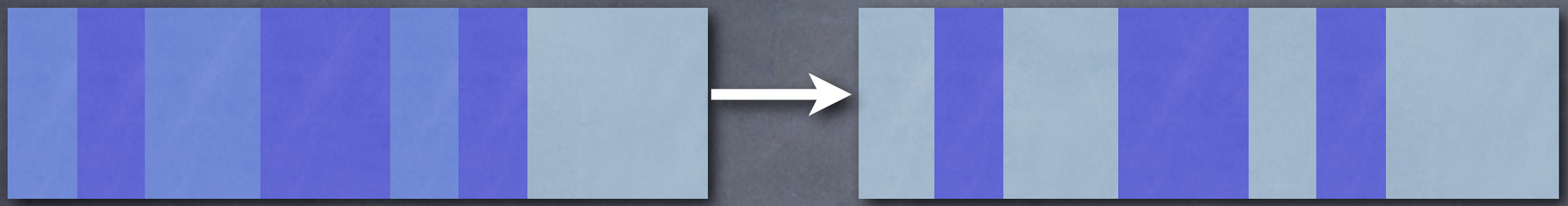
Mark-Compact



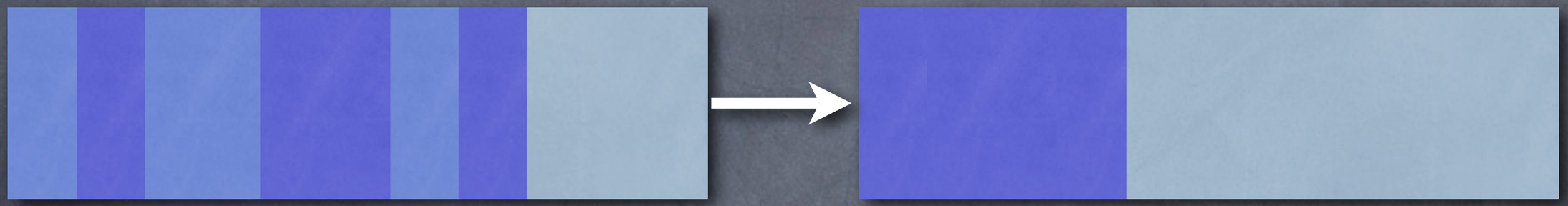
Two-space Copying



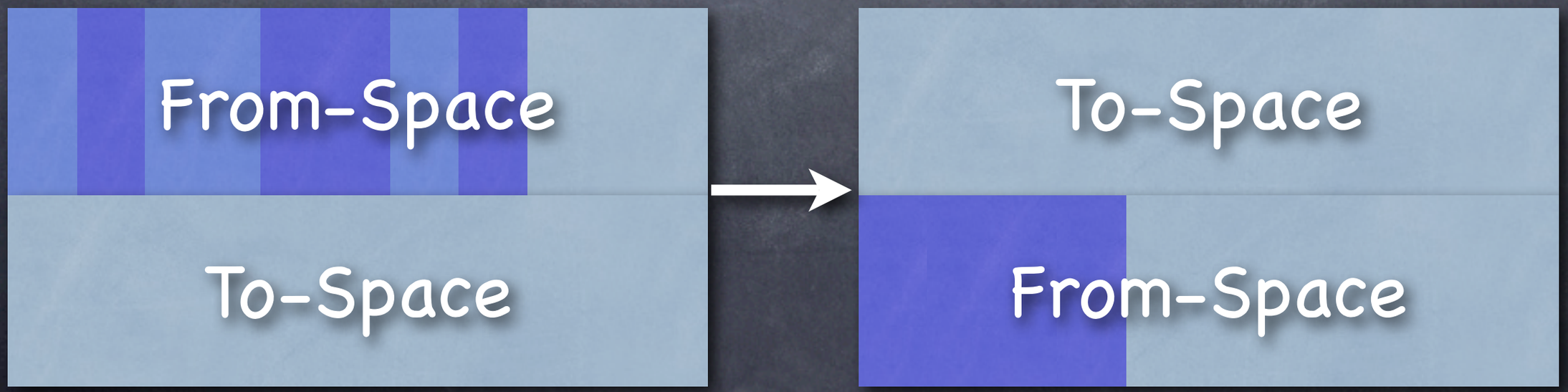
Mark-Sweep



Mark-Compact



Two-space Copying



- Main collector types:
  - Mark-Sweep: can be really fast, but no defragmentation means no strong space usage bounds.
  - Mark-Compact: tight space usage bounds, but typically really slow, because the "compact" phase first requires identifying the layout of used objects and then sliding them into a new layout. In simple implementations, three heap passes are required.
  - Copying: typically fast, but evacuation is always to a free part of the heap - thus, 2x memory overhead in simple case.

Real-time Garbage Collection

Filip Pizlo

POPL'08 | W. Lafayette, IN

RTGC: what is it?

- Real-time Garbage Collection is supposed to:
  - Offer time- and space-predictability.
    - Time-predictability means that for any "meaningful" unit of work, the max time required is not too big.
    - Space-predictability means: we don't want to run out of memory.

RTGC: what is it <u>really</u>?

- High average-case throughput
- Worst-case running time of any task is U×M+P, where M is the running time without GC, U≤2 (tyically), and P is small (under a millisecond).
- U represents the "utilization" that the collector gives to the application ("mutator")
- P is the longest pause due to collection activity.

RTGC: state of the art

- Sort-of Real-time GCs:
  - Too many to list - Baker'78 was first
- Really Real-time GCs:
  - Henriksson (1998)
  - Metronome (2003)
  - JamaicaVM (1999)

Talk Overview

- Why is RTGC hard?
- Early, pseudo-real-time approaches
- Modern approaches

Why is RTGC hard?

Or:

Why is non-RT GC easy?

- If responsiveness is not an issue, the GC can complete in one long pause under the assumption that there is no interleaved application (mutator) activity.

- This is critical to allowing many of the major optimizations used to make GCs fast.

- Why non-RT GC is easy:
  - Marking is easy.
  - Copying is easy.
  - Reducing space overheads is easy.

Marking is Easy

- Marking requires figuring out which objects are in use.
- We do this with a graph search.
- Graph searching is easy - IF the graph isn't changing while you're searching it.
- Thus: marking is easy if you pause the application for the entire time it takes to mark.

Copying is Easy

- Copying is considered essential for guaranteeing good space usage bounds.
- Copying an object from one location to another and then "fixing up" the heap so that the new copy is used is easy, provided that you can pause the application while you do it.
- Typically, we pause the app, perform all copying, perform all fixup, and unpause the app.

Space Overheads: Easy

- Marking requires searching; searching requires a worklist. Worklists require space.
  - But if the application is paused, it's easy to reuse object fields for the worklist.
- Copying requires keeping track of where objects are copied to; typically a "forwarding pointer" is added to the object header.
  - But if the application is paused, it's easy to reuse one of the other header fields for the worklist.

Non-RT GC is Easy

- Marking and copying are easy because the GC is the only entity operating on the heap.
- Space overheads are easy because you can cleverly reuse fields for bookkeeping purposes.

RTGC is Hard

- You don't want to do all collection in one long step, which would lead to a long pause.
- Marking is hard if you cannot do it in one big step.
- Copying is hard if the application is concurrently using the object being copied.
- Space overheads are hard because you have to store bookkeeping information on the side.

RTGC is Hard

- The challenge of RTGCs is:
  - Implementing a sound marking phase that allows mutator interleaving.
  - Implementing a sound copying phase that allows mutator interleaving.
  - Controlling space overheads.
  - Scheduling the collector work so as to give the programmer a reasonable performance model.
  - Making it fast.

Early Appraoches

- The first attempt at RTGC was by Baker in '78, with his real-time list processing.

- This was a LISP collector, with all objects being the same size and small.

- Baker'78 is a copying collector that instruments the mutator to maintain:

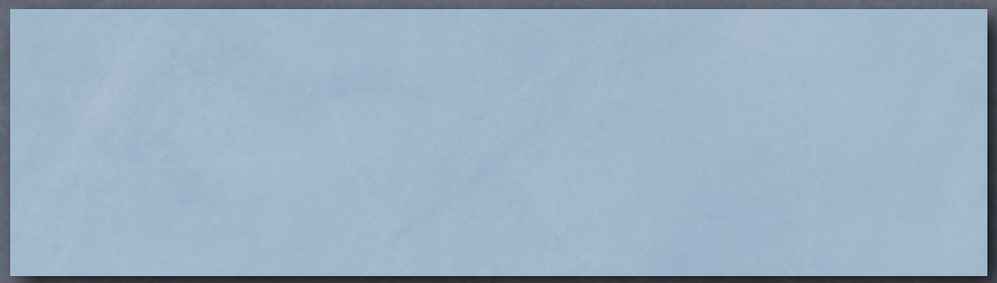
  - To-space invariant (mutator always uses to-space, never sees from-space)

  - Steady state (collector keeps up with mutator allocations)

From-Space

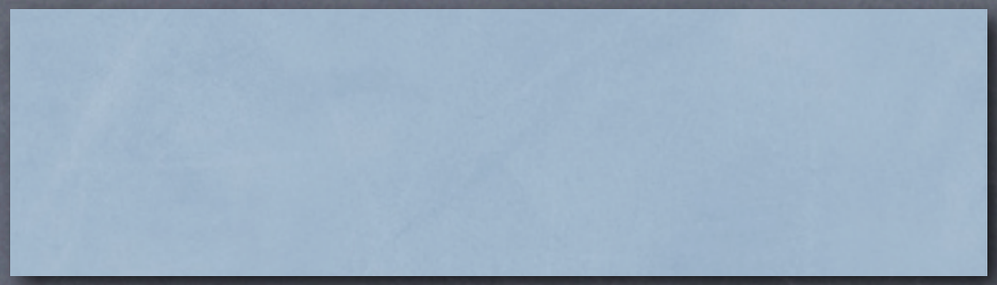
From-Space

To-Space



From-Space

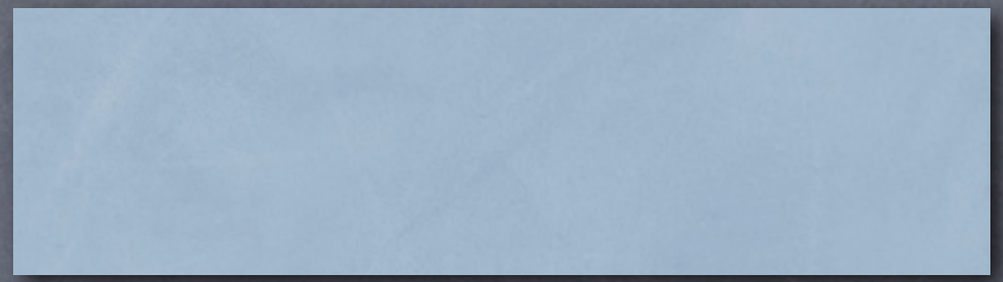
To-Space

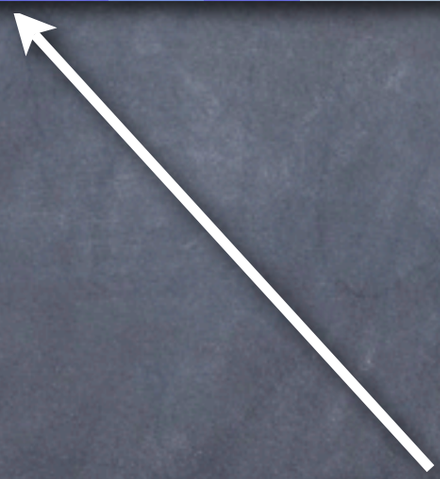


Mutator

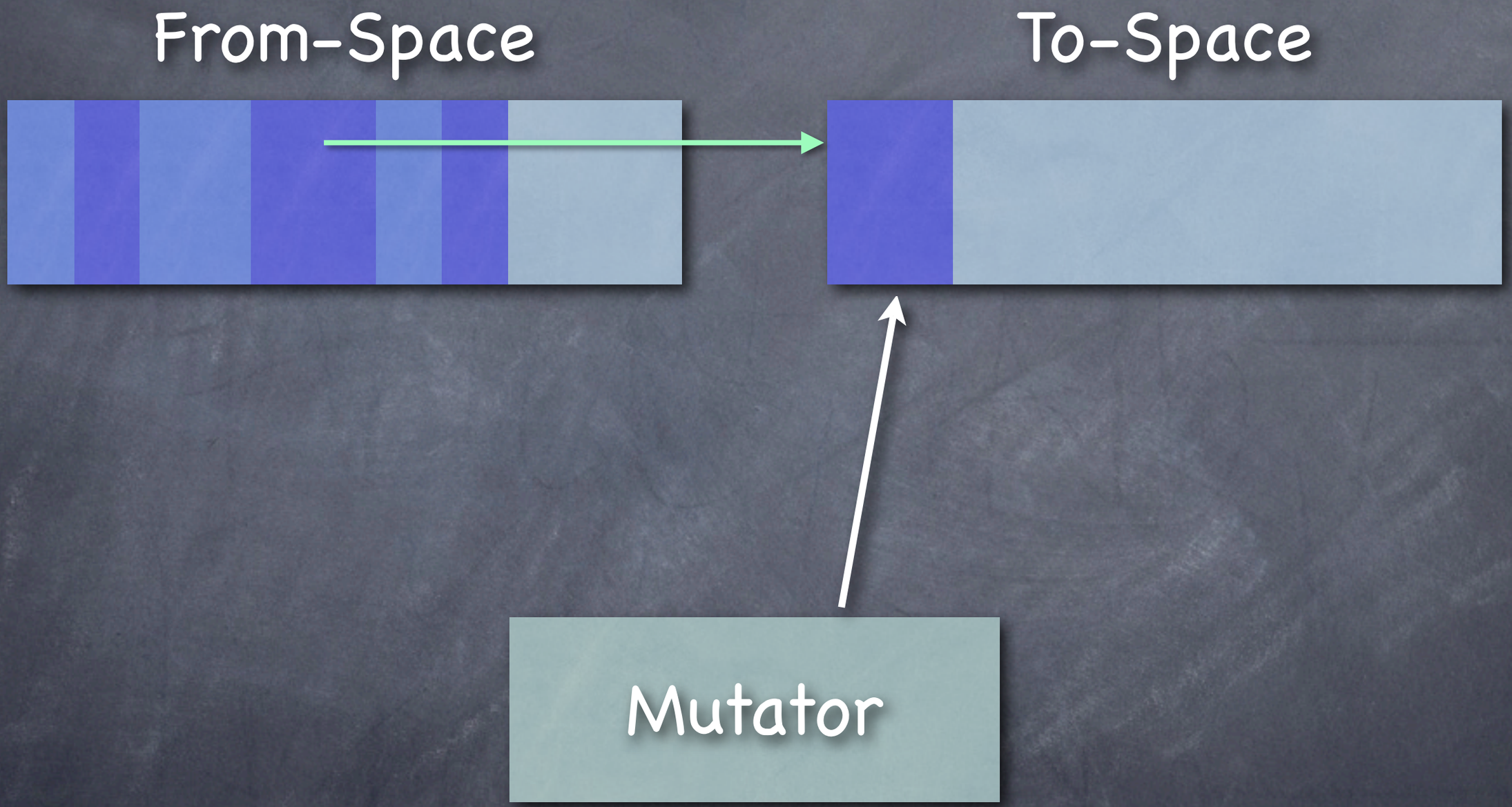
From-Space

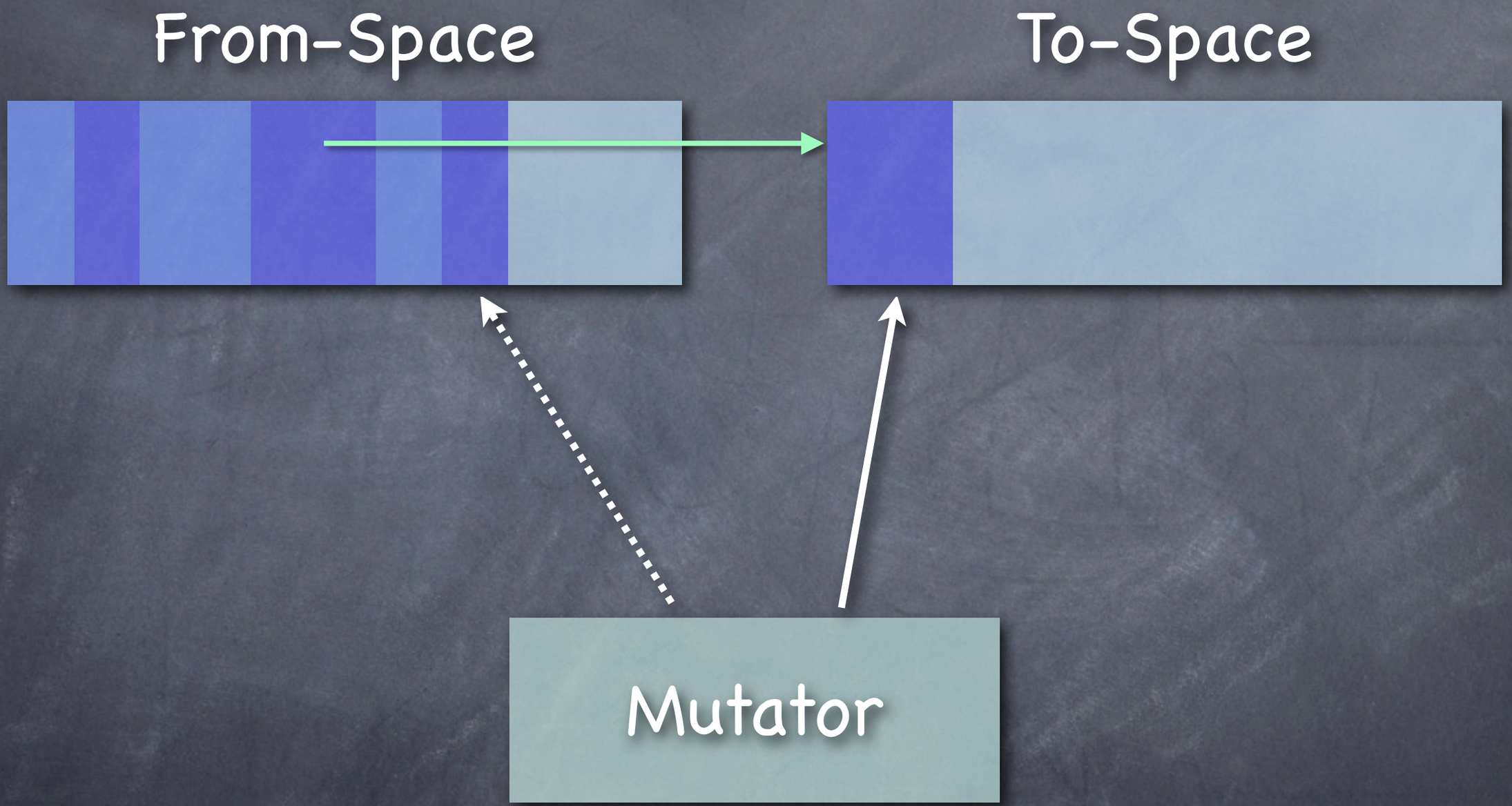
To-Space

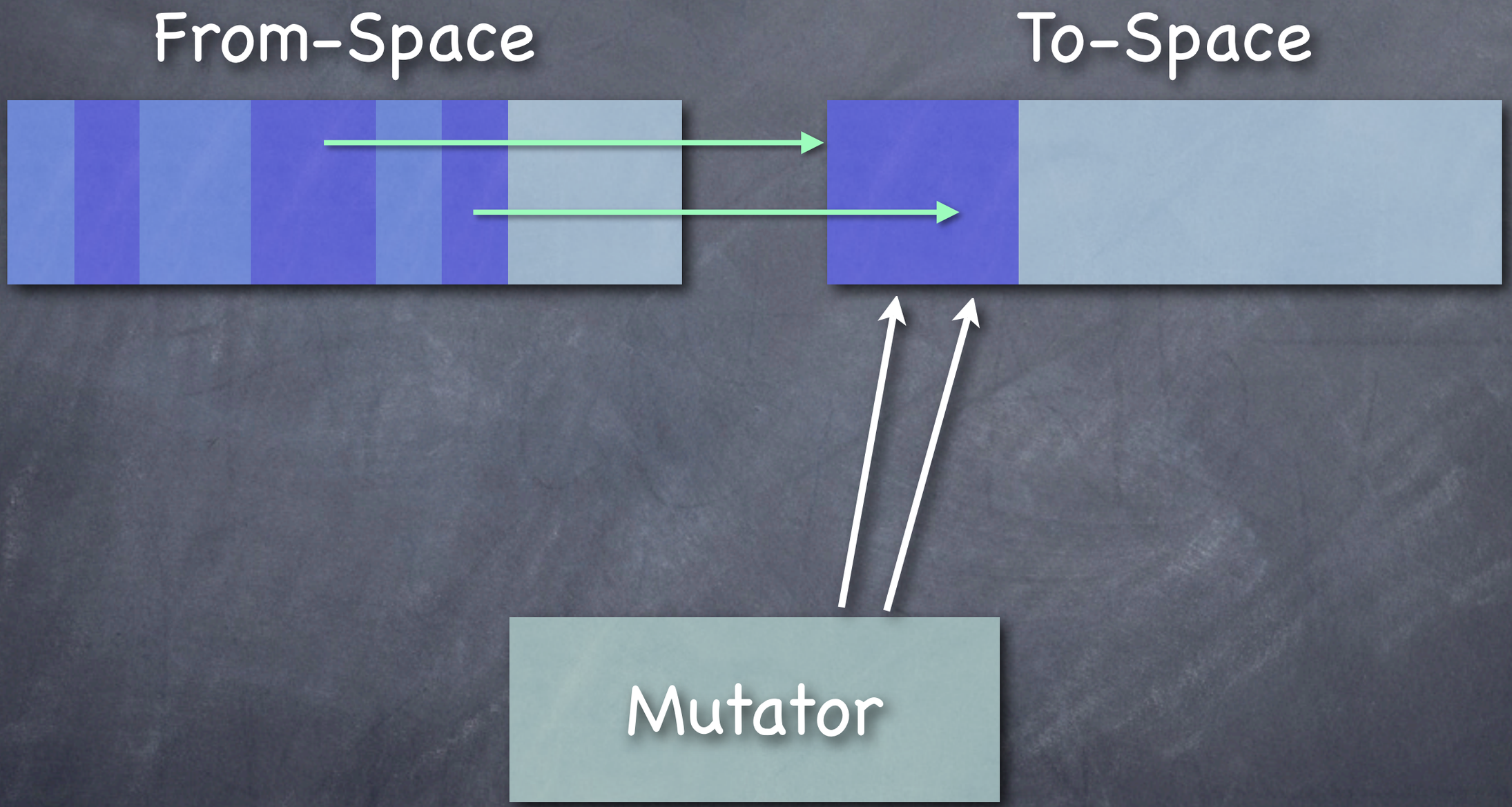


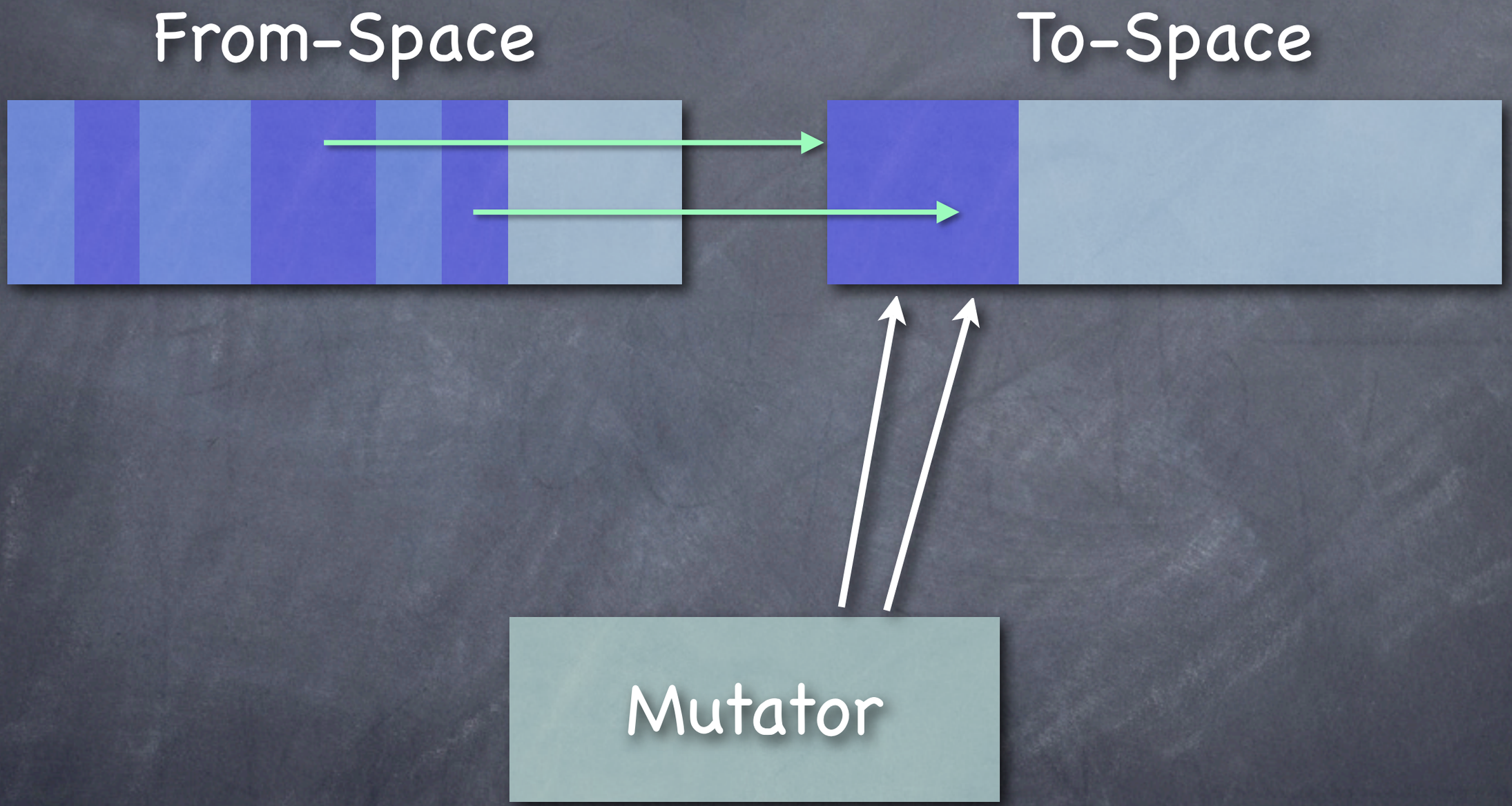


Mutator


From-Space
To-Space
Mutator


From-Space
To-Space
Mutator


From-Space
To-Space
Mutator



Reading a reference to an object triggers copying.

- What remains is to ensure steady state.
- In addition to reading (and writing) objects, the mutator will allocate.
- Allocation can exhaust the heap.
- Thus: we want the collector to finish before the heap is exhausted.
- Baker'78 does this by performing collector work in allocations.

- We say that Baker'78 is a work-based collector: mutator work (that is, allocations and reads) trigger collector work.

- "Work-based" is thus short for collector work being "based" on mutator "work".

- Prior to Henriksson '98, this was the state of the art of RTGC.

Dissecting Baker'78 Features

- Baker'78 copies objects and performs marking with interleaved collector activity by using a "read barrier":
  - The read barrier ensures that the mutator only sees to-space.
  - The read barrier ensures that the graph search doesn't lose track of heap changes.

Problems with Baker'78

- The chief problem with Baker'78 is the to-space invariant.
- Maintaining the to-space invariant requires a read barrier.
- Reads are frequent - thus, instrumenting reads leads to poor performance.
- Further, if objects have variable size (as in the example), the cost of a read will be variable and in the presence of polymorphism, unpredictable.
- What RT programmers fear: a burst of reads may come all at once, causing substantial worst-case slow-downs that are not easy to account for.

- Subsequent work on RTGC lead to the following:
  - Whereas Baker uses the to-space invariant, modern incremental collectors use tri-color marking, which allows the use of cheaper barriers (see Pirinen'98 for a review).
    - Though, fast tri-color collectors like Doligez-Leroy-Gonthier do not perform defragmentation!
  - Collectors such as Nettles-O'Toole perform defragmentation without a read barrier.

Modern Approaches

Overview of Modern RTGCs

- JamaicaVM
- Henriksson
- Metronome

First I will talk about JamaicaVM, for two reasons

First: in many ways it's the most traditional design.

Second: as I will show you, it's INSANE.

JamaicaVM

- JamaicaVM is a modern example of a work-based collector.
- JamaicaVM has the following features:
  - Mark-Sweep, w/o copying.
  - **All objects are same size.**
  - **Mutator root scanning is fully incremental.**

- JamaicaVM is insane:
  - All Java objects are split into 32-byte chunks (large objects become linked lists, arrays become tries)
  - No copying
  - The stack is logically a heap object.
  - All collector pauses are aggressively bounded.

- JamaicaVM uses a write barrier to mark objects that are store into the heap.
  - This includes the stack, since the stack is a heap object!
- Allocation triggers a bounded amount of collector work, which may be one of:
  - Stack scanning
  - Marking
  - Sweeping

- JamaicaVM is not well-liked
- It has never been accepted by the PL community (no PL publications, totally overlooked in mainstream GC literature)
- Why is this?

- Modern RTGCs cannot just be responsive and have reasonable scheduling.
  - JamaicaVM is without a doubt the best at scheduling.
    - All mutator work leads to bounded and small increments of collector work.
- Modern RTGCs also have the be fast.

- Though no comprehensive study of JamaicaVM performance versus current best-of-breed Java systems has ever been performed...

- It stands to reason that marking objects on heap reads, turning array accesses into trie traversals, and accessing object fields via linked lists is absurdly slow.

- That said: JamaicaVM should not be overlooked, and there "should" be a burden on any RTGC designer to compare his results against the JamaicaVM strategy.

Henriksson

- The Henriksson collector takes a heroic approach to real-time guarantees:
  - Henriksson attempts to guarantee that a real-time task will never experience GC interference.
- How is this done?
  - Slack-based GC scheduling: an RT task can always preempt the GC.

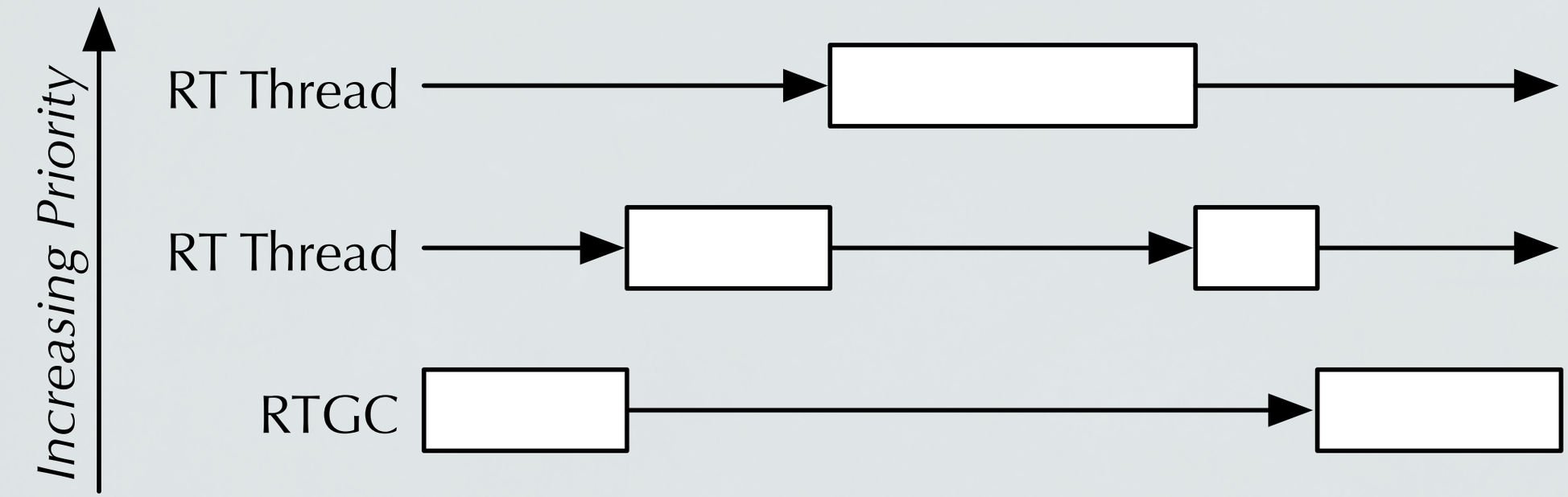


Figure 2: *Slack-based RTGC scheduling. The RTGC only runs when none of the real-time threads are running.*

- The Details:
  - Henriksson is a copying collector.
  - Objects have normal representation.
  - Read & write barriers are used to ensure marking/copying soundness.
  - If the mutator wishes to run, the collector rolls back the current operation (i.e. object copy) and yields.

- Why it's great:
  - Slack-based scheduling gives the application programmer full control over the GC schedule.
  - In many cases this Just Works for RT apps.
  - Sun is using a variant of the Henriksson collector in their Real Time Java product.

- Why it's bad:
  - Read barrier that marks objects.
  - Object copying is aborted when mutator runs (this makes guaranteeing collector progress hard).
  - Programmer must know his real-time tasks' allocation rate, and leave enough slack to ensure that the collector keeps up.

Metronome

- The Metronome Is Important.
  - It's the first RTGC to be "recognized" by the PL community.
  - It's arguably the most successful.
  - It's arguably the fastest.
  - It's arguably the simplest to reason about.

- Metronome uses time-based scheduling: the collector runs periodically for a bounded amount of time.
- Amount of collector interference is easy to understand.
- Progress is easier to guarantee than in Henriksson.

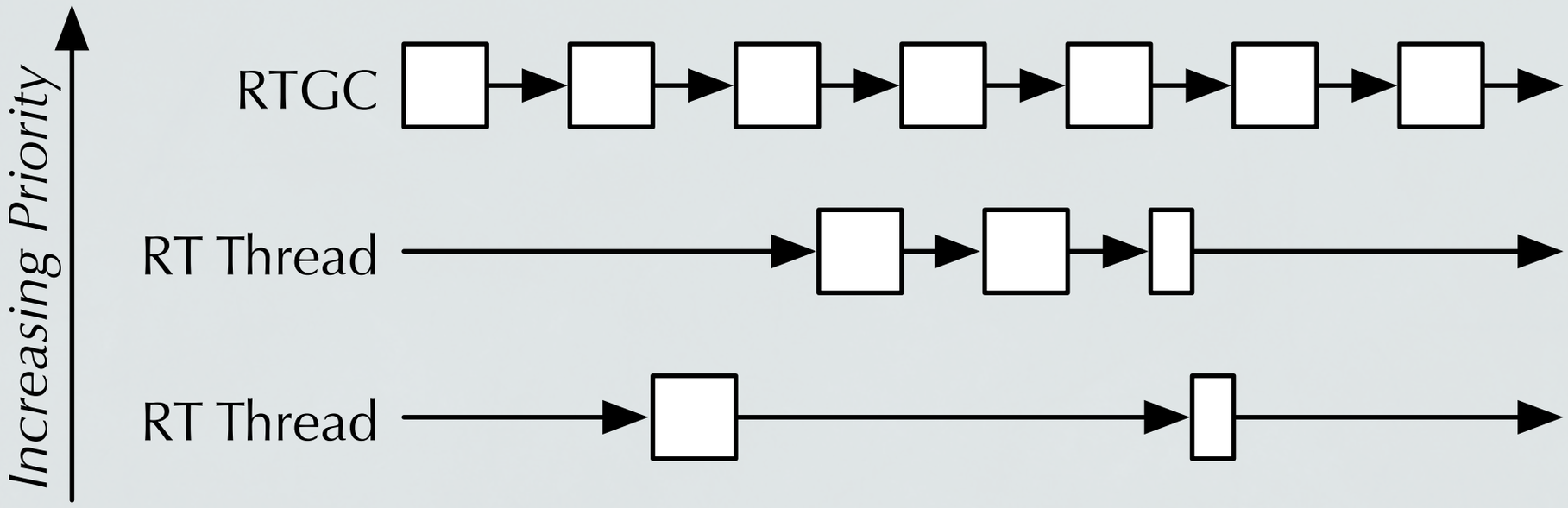


Figure 3: *Time-based RTGC scheduling. The RTGC has the highest priority but only interrupts the application at well-defined intervals.*

- Metronome is a Mark-Sweep-and-sometimes-Copy collector.
  - A write barrier is used to ensure marking soundness as in DLG.
- A very light read barrier is used to ensure copying soundness.
- Almost-empty pages are evacuated in response to fragmentation.

- Metronome uses a weird object model to ensure timeliness.
  - Arrays are split into page-size "arraylets", leading to array accesses that generalize to something like the JamaicaVM tries.
  - Thus collector increments are lower-bounded by the time it takes to copy a page-size object.
  - And, allocation of large objects is a non-issue.

- Why it's good:
  - Easy to understand.
  - Probably fast (no heavy read barrier).

- Why it's bad:
  - Need to know worst-case allocation burst rate.
  - Probably slow (because array accesses are trie accesses).

Review of Barrier Overheads

	Local Reference Assignment
No Barriers	a = b
Metronome	a = b
Henriksson	a = b
JamaicaVM	mark(b) a = b

	Primitive Heap Load
No Barriers	a = b->f
Metronome	a = b->forward->f
Henriksson	a = b->forward->f
JamaicaVM	a = b->f

	Reference Heap Load
No Barriers	a = b->f
Metronome	a = b->forward->f
Henriksson	a = b->forward->f mark(a)
JamaicaVM	a = b->f mark(a)

	Primitive Heap Store
No Barriers	a->f = b
Metronome	a->forward->f = b
Henriksson	a->forward->f = b
JamaicaVM	a->f = b

	Reference Heap Store
No Barriers	a->f = b
Metronome	mark(a->forward->f) a->forward->f = b->forward
Henriksson	mark(b) a->forward->f = b->forward
JamaicaVM	mark(b) a->f = b

- Notes:
  - Array accesses in Metronome require more work (trie access)
  - Accesses to any field past the first 32 bytes in JamaicaVM require more work (linked list access, trie access)