17-363/17-663: Garbage Collection

Joannah Nanjekye, PhD
jnanjeky@cs.cmu.edu
Python Core Developer and Programming Languages Researcher
October 21, 2024

Garbage Collection

- ➔ A 70-year old field now
- ➔ Automation of dynamic memory allocation
- ➔ Automation is vital for **managed languages**
- ➔ Where the VM automatically allocates and deallocates memory on the user's behalf because:
  - It is necessary for working programs
  - It is a complex manual task
  - And error-prone

A garbage Collector is the component that reclaims memory that the program no longer needs

Dijkstra Terminology

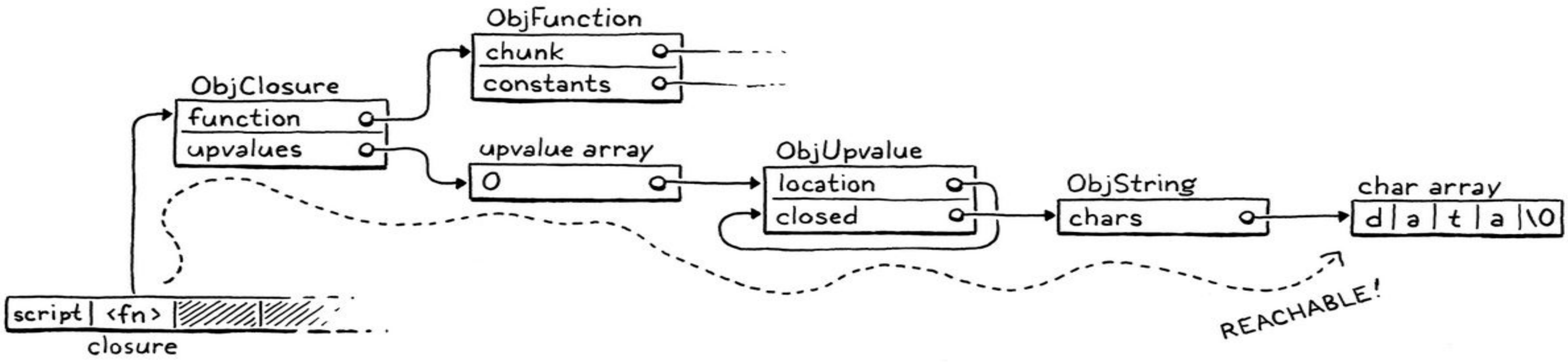
The collector refers to the aspect of the application that reclaims memory for objects that are considered garbage

The mutator refers to the application program

The allocator is responsible for the allotment of memory for objects

Categorizing Garbage Collectors

- Garbage collectors are categorized by how they:
  - Allocate objects
  - Identify unused objects
  - Free unused memory



Allocation

Objects are allocated in memory in one of two ways:

1. ***Contiguous Allocation:*** Places objects in memory in the order in which they are allocated
    - Achieves this by incrementing the allocation pointer based on the size of the object to be allocated
    - Algorithms based on this technique have good locality because objects are allocated and used together
2. ***Free-list allocation:*** Free lists are lists of variable-size cells of memory. Free-list allocation places objects in these cells
    - Objects are allocated in memory based on their size relative to the cell
    - Allocation is in a first-fit fashion rather than allocation order
    - It permits non-contiguous allocation, which is prone to fragmentation and poor locality

Identifying Garbage

Achieved in the following one of two ways:

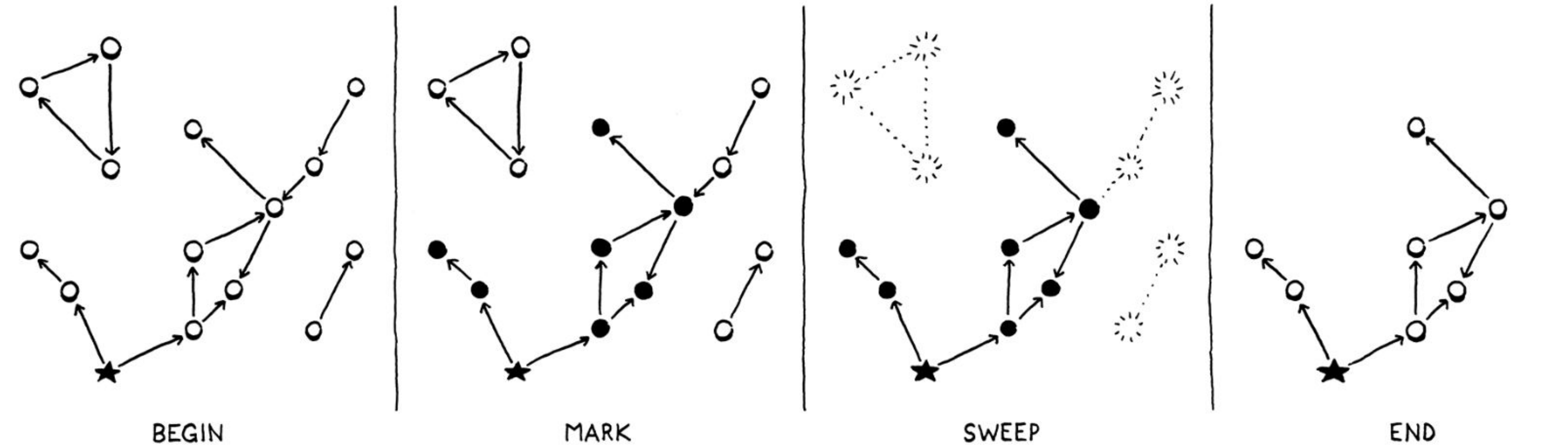
1. ***Reference Counting:*** Involves tracking the number of times an object is referenced by other objects
    - An object is considered garbage if its reference count drops to zero
2. ***Tracing:*** Scans the object graph for objects that are not directly or indirectly referenced from a root object
    - An object that is not reachable by reference from the root is considered garbage and thereby collected

Memory Reclamation

Memory reclamation is achieved by one of these strategies::

1. ***Back to a free-list:*** Memory is returned to the free-list at the time of deallocation
2. ***Sweeping:*** Involves traversing the object heap, marking unused blocks of memory as free
3. ***Compaction:*** re-arranges the remaining objects after a collection to avoid fragmentation
4. ***Copying:*** Moves objects from one region in memory to another, freeing up the former

Overview of Tracing Garbage Collection



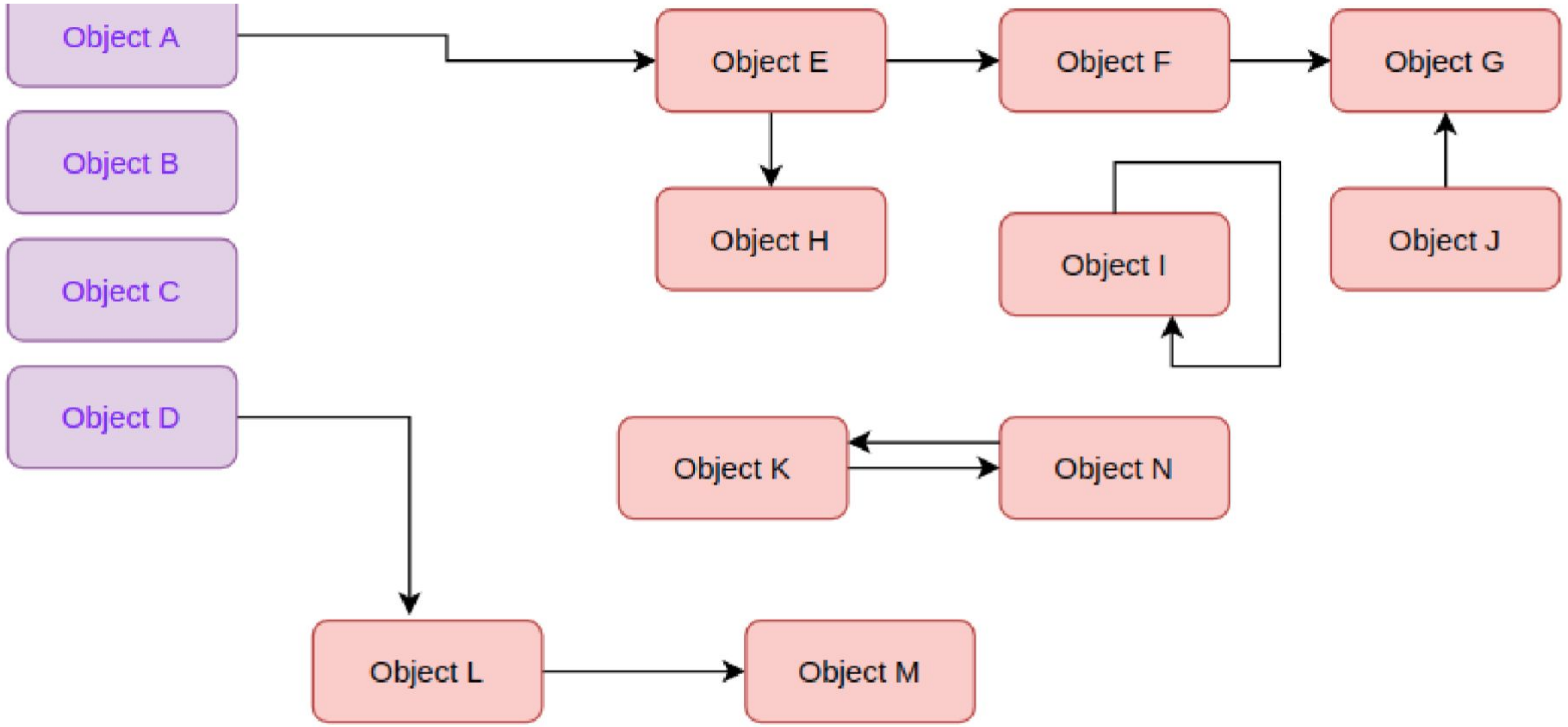
A **tracing garbage collector** is any algorithm that traces through the graph of object references. This is in contrast with reference counting, which has a different strategy for tracking the reachable objects.

Terminology

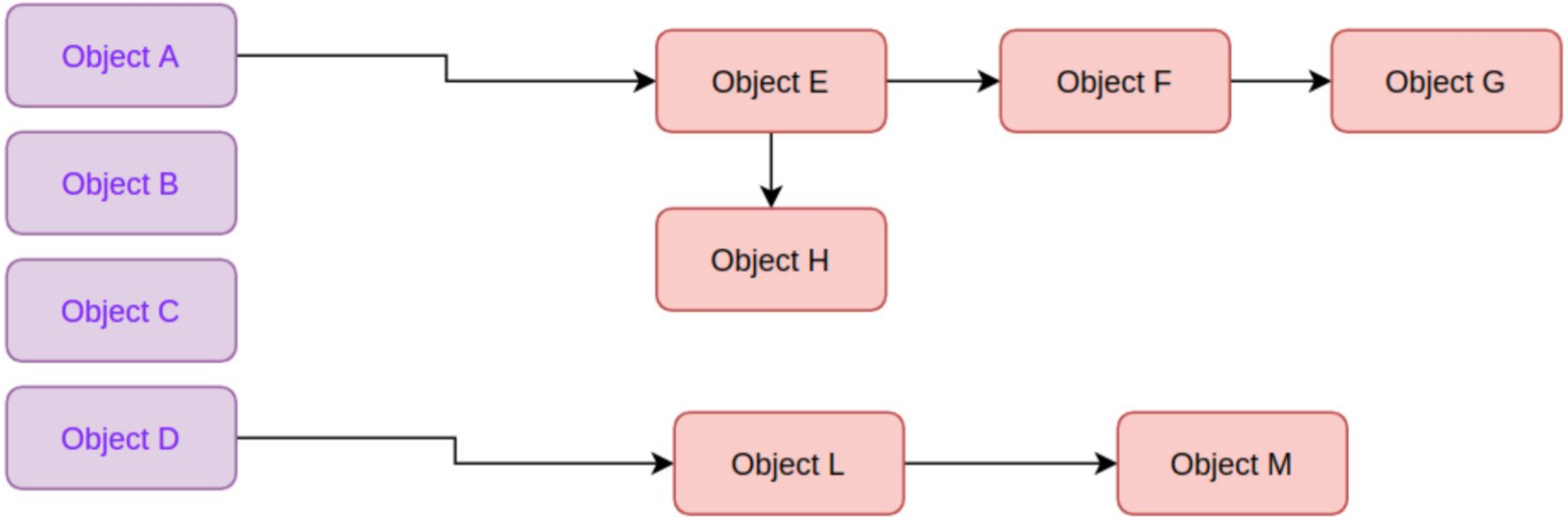
1. ***Object:*** This is simply an instance of data stored on the heap
2. ***Object graph:*** This is the layout of objects in memory; the objects make up the nodes of this graph
3. ***The Root Set:*** This is a set of objects in the object graph from which references originate and are directly accessible by the mutator.
4. ***Reachable or Live Objects:*** These are objects that have an incoming edge referencing them from one of the root sets or edges from other reachable objects
5. ***Unreachable or Dead Objects:*** Objects that do not have any incoming edge referencing them from the root set or edges from other reachable object
6. ***Collection:*** The process of reclaiming memory that is occupied with unreachable objects
7. ***Barrier:*** An operation that is invoked before reading or writing to a pointer
8. ***A Conservative Collector:*** A garbage collector that works with minimal information about pointers

Garbage Collection Example

Before Collection



After Collection

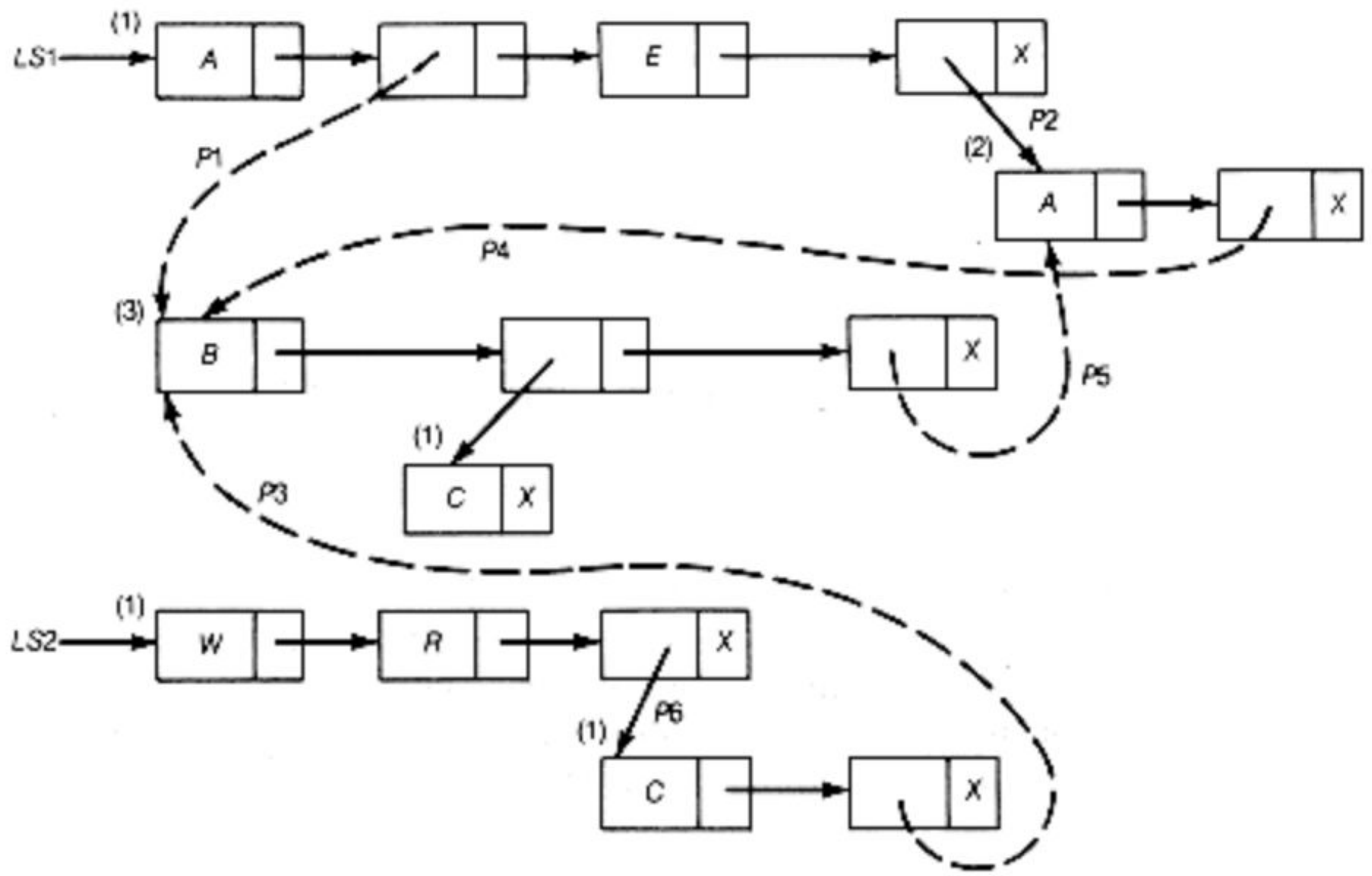


Garbage Collection Algorithms

Reference Counting: works by tracking a count of all incoming references to an object, collecting objects whose reference count decreases to zero

- A write barrier is used to synchronize changes to every pointer,
- Hailed for low memory overhead and ease of implementation
- Has two main limitations:
  - Cannot collect garbage for objects in a cycle
  - Incurs performance overhead as a result of tracking pointer mutations

A lot of research exists to address these known challenges



Garbage Collection Algorithms

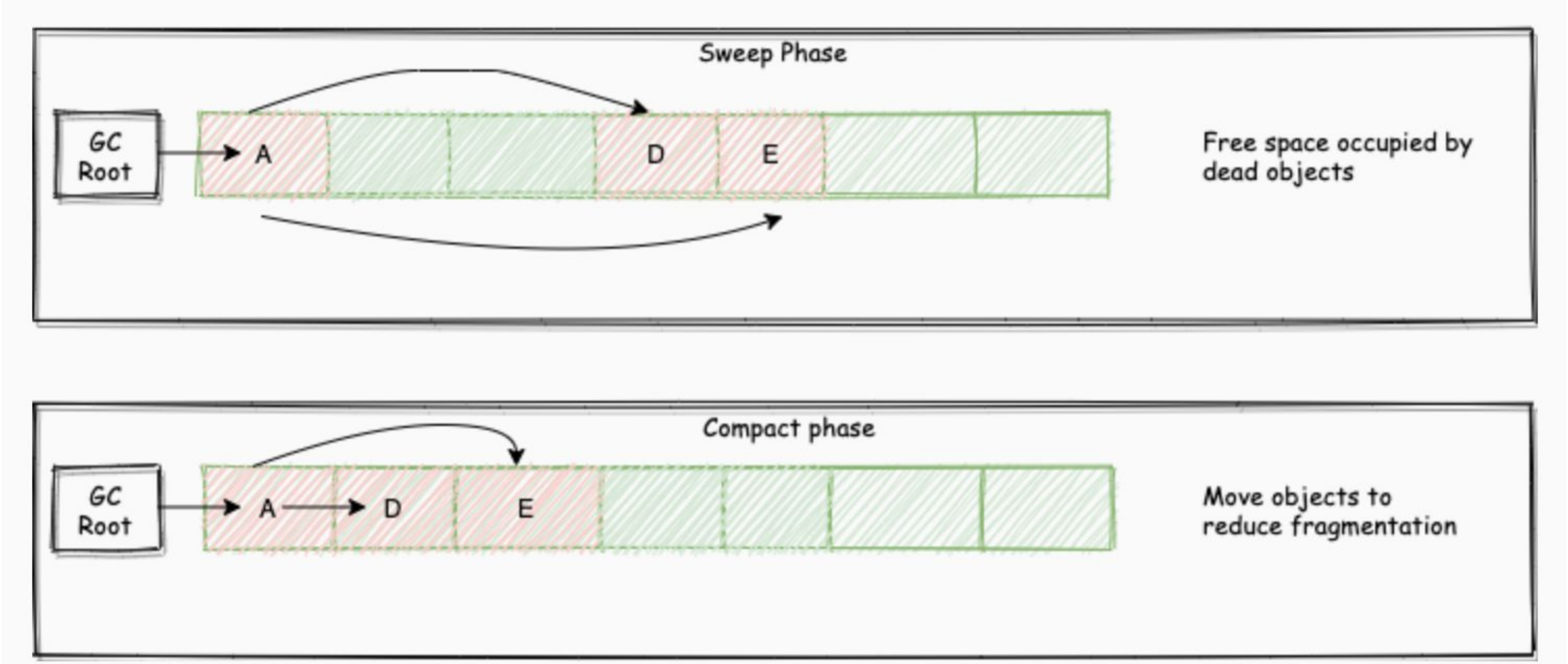
Mark-Sweep: Performs memory management activities in two phases, i.e., the mark and sweep phase

- The *mark phase* traverses the object graph from the root set, marking all objects or nodes that have an incoming reference from the root set or other reachable objects
- The *sweep phase* then traverses the whole heap, freeing any memory with unmarked objects

Garbage Collection Algorithms

Mark-Compact: Rearrange objects in memory after a collection cycle.
Compaction is commonly used in the mark-and-sweep collection

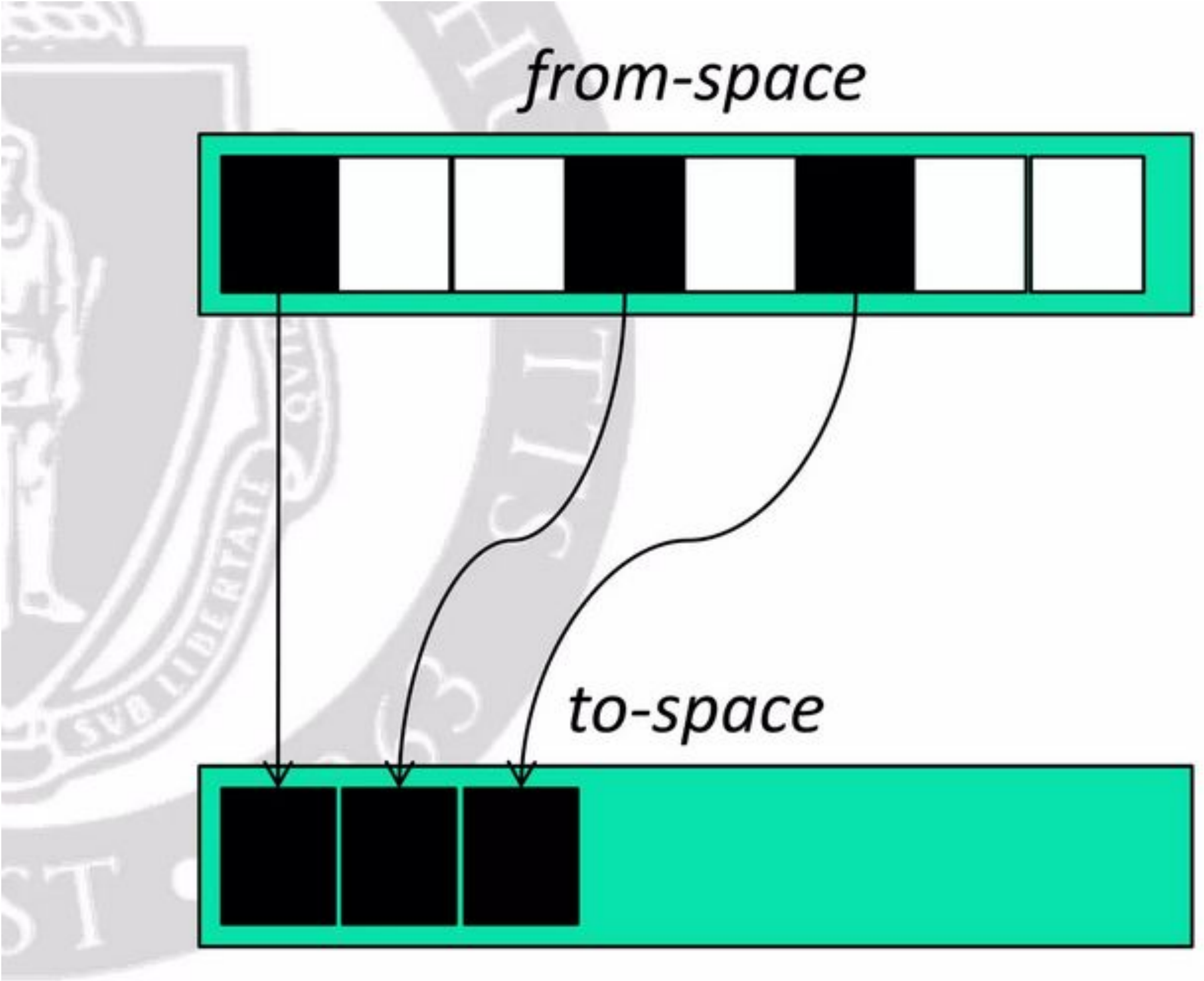
- After the marking phase, compaction can be used in addition to a normal sweep phase
- Sweeping without rearranging objects creates fragmentation and compaction solves this problem
- A simple compaction algorithm uses sliding to compress reachable objects into a contiguous memory space while maintaining their order in the heap

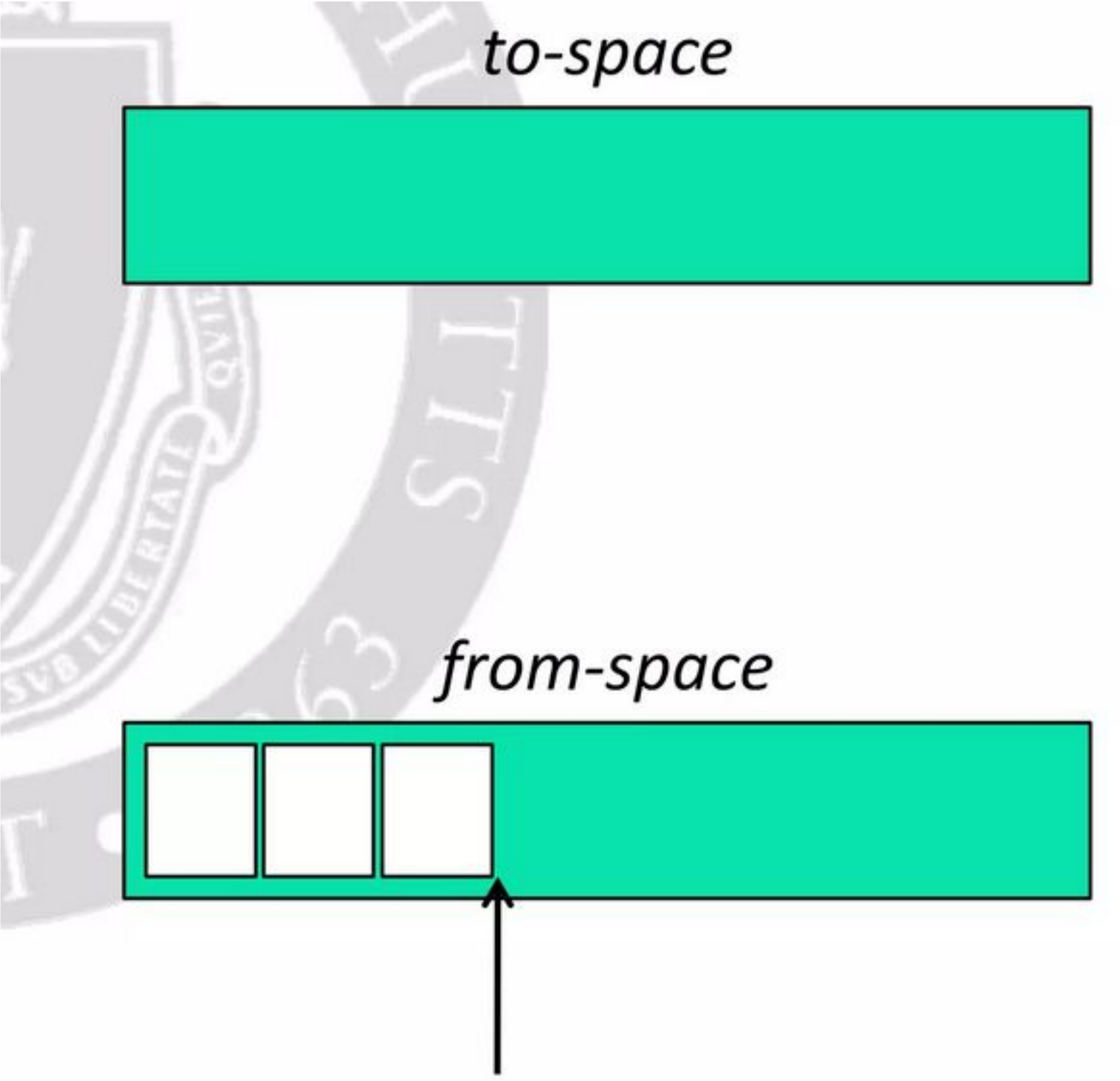


Garbage Collection Algorithms

Semi-space Copying: Copying collectors divide the heap into two regions

- Allocation of objects is done in the first region called the *from-space*
- When it runs out of space, collection takes place copying any live objects to the second region called the *to-space*
- The pointers to the moved objects are updated

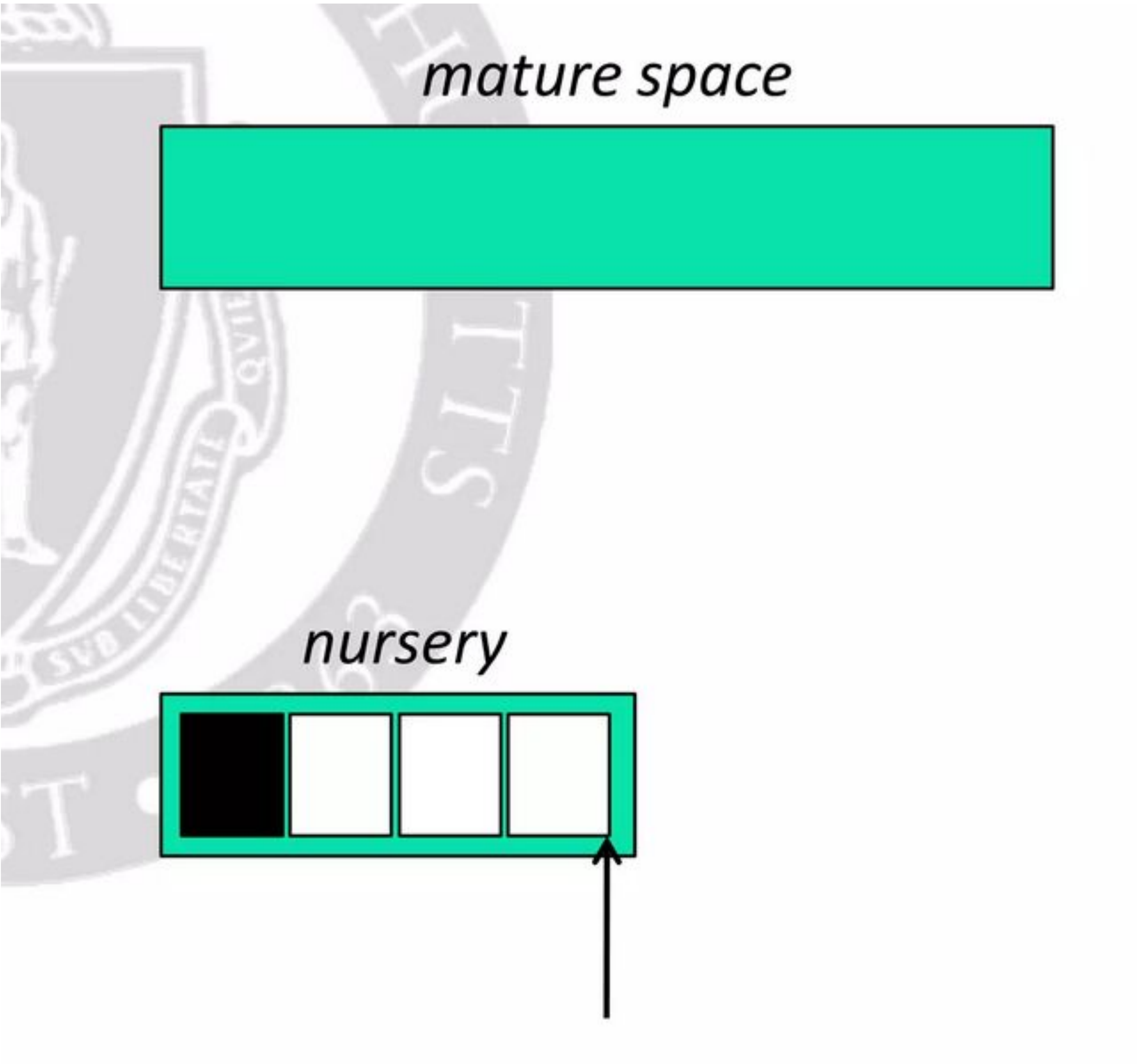


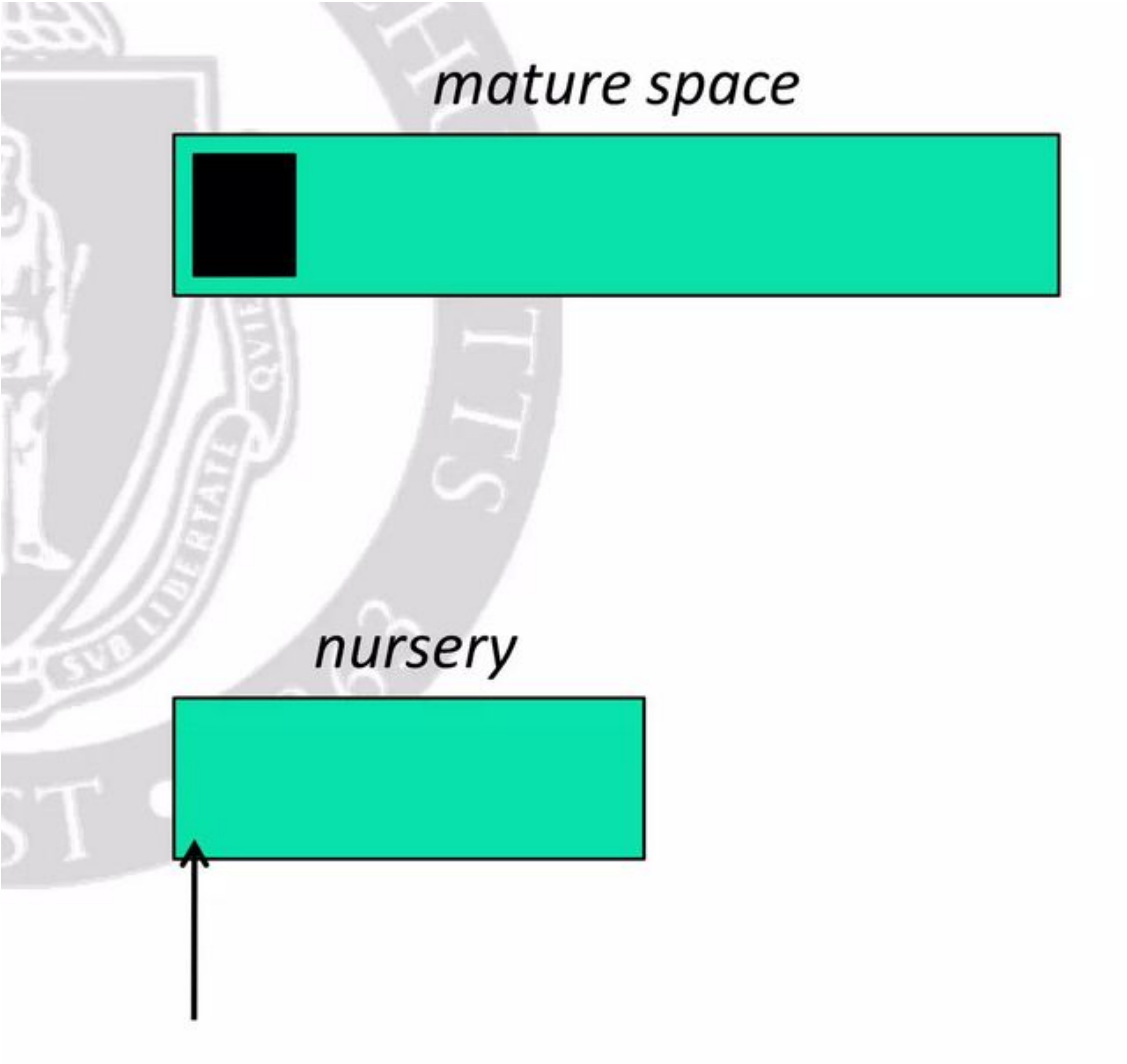


Garbage Collection Algorithms

Generational Collection: Based on the weak generational hypothesis hypothesis, generational collectors are region-based GCs and similar to the semi-space collectors

- The weak generational hypothesis states that most of the objects live for a short time
- Generational GCs partition the heap in two generations, the **nursery** and the **old generation**
- The nursery is frequently collected while the old generation is less often collected





Challenges for all Algorithms

- Handling conservative references
- Performance (latency and throughput)
- Visitation Order
- Number of passes over the heap
- Locality
- Fragmentation
- Parallelism

Garbage Collection and its Economics

Programming Languages

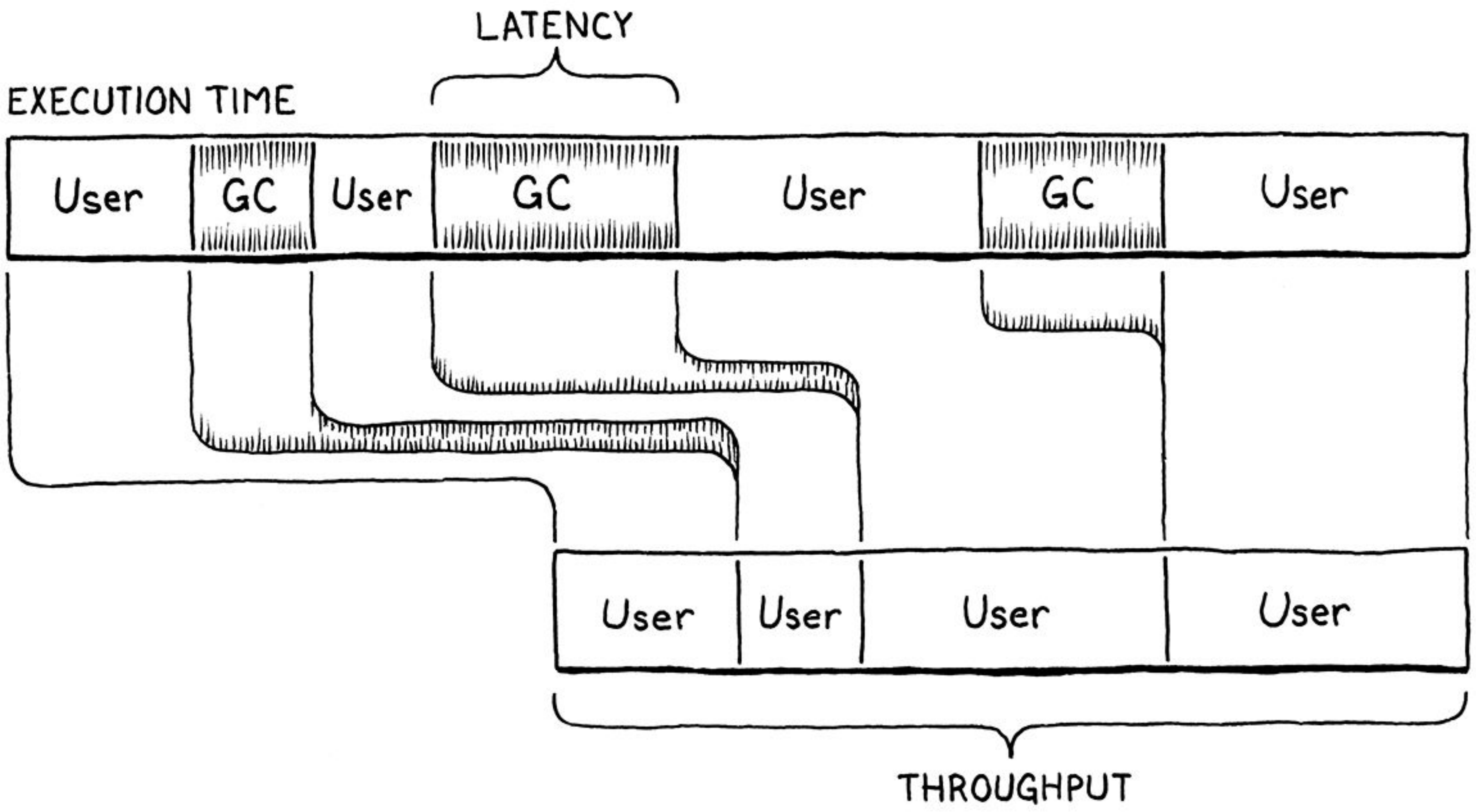
Design choices lead to GC cost

Architectures

Modern architectures can have memory overhead

Applications

The runtime behaviour affects GC cost



Source: Nystrom, R., 2021. *Crafting interpreters*. Genever Benning.

Potential Paths: To understand and optimize garbage collection

Start with identifying garbage collection gaps?

GC Gaps: Modularity

Complexity is the Enemy of Security and Performance
Steve Blackburn, MPLR 2020, Keynote

Effects of this Complexity:

- ❖ Compromises security
- ❖ Complicates maintenance
- ❖ Hinders analysis

GC Gap: Performance

```
Samples: 2K of event 'page-faults', Event count (approx.): 91453
  Overhead       Samples  Command           Shared Object         Symbol
-   17.08%           198  uwsgi             uwsgi                 [.] collect.part.7
   - collect.part.7
      - _PyObject_GC_New
         + 98.54% PyDict_New
         + 1.46% list_iter
+   13.43%           356  uwsgi             _scrypt.so            [.] crypto_scrypt
+   11.99%           444  uwsgi             libc-2.20.so          [.] _int_malloc
+    7.11%           291  uwsgi             uwsgi                 [.] PyObject_Malloc
+    5.91%           218  uwsgi             uwsgi                 [.] PyObject_GenericGetAttr
+    4.68%           176  uwsgi             uwsgi                 [.] PyEval_EvalFrameEx
+    4.61%           167  uwsgi             uwsgi                 [.] PyFrame_New
+    2.37%            27  mc-eccc-pool      libc-2.20.so          [.] _int_malloc
+    2.29%            96  uwsgi             libc-2.20.so          [.] __memcpy_sse2_unaligned
+    2.25%            48  cfgator-sub       libc-2.20.so          [.] _int_malloc
```

<table>
<tr><th>Problem</th><th>Patch</th></tr>
<tr><td>ActiveRecord::Base#create calls attributes_with_quotes twice, ActiveRecord::Base#update once. #attributes_with_quotes calls #attributes, which clones all attribute values. None of these clones are necessary.

Impact:
When copying 120 tasks in Acunote this costs 650M. Patch improves performance from 14s to 6s.
Notes:
Helps when you do a lot of creates and updates through ActiveRecord.</td><td>Rails 1.2


```ruby
module ActiveRecord
    class Base
    private
        def attributes_with_quotes(include_primary_
            result = {}
            @attributes.each_key do |name|
                if column = column_for_attribute(na
                    result[name] = quote_value(read
                end
            end
            result
        end
    end
end
```

Rails 2.0

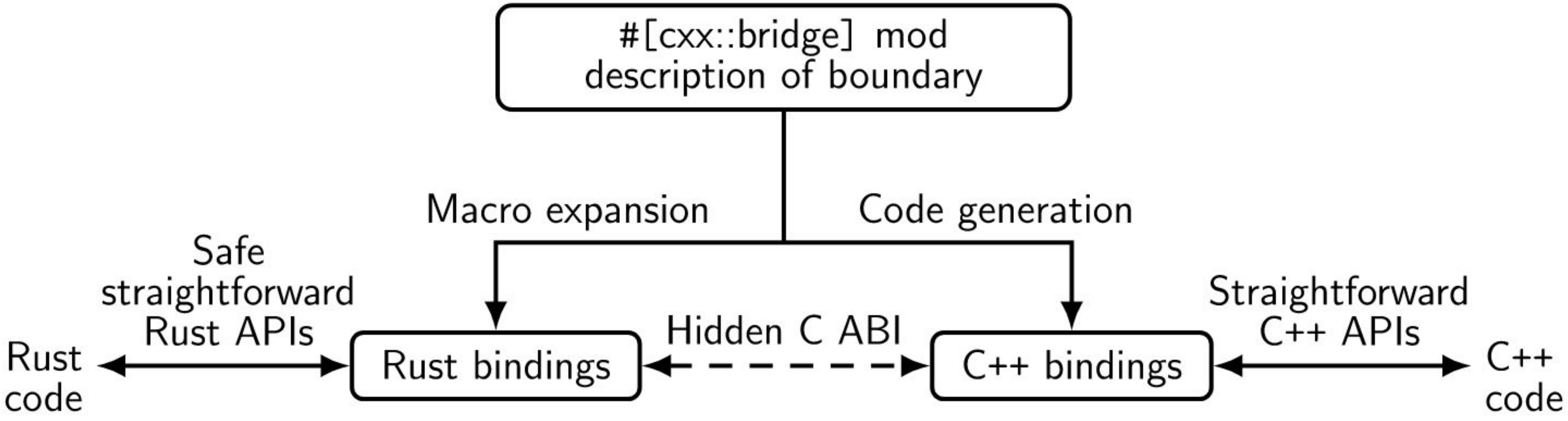

```ruby
module ActiveRecord
    class Base
    private
        def attributes_with_quotes(include_primary_
            quoted = {}
            @attributes.each_pair do |name, value|
                if column = column_for_attribute(na
                    quoted[name] = quote_value(read
                end
            end
            include_readonly_attributes ? quoted :
        end
    end
end
```

</td></tr>
</table>

Source: Instagram, 2017. *Dismissing Garbage Collection at Instagram*. Chenyang Wu, Min Ni.

Source: Acunote, 2008. *Garbage Collection is Why Ruby is Slow*. Gleb Arshinov.

GC Gap: Support for Native Extensions



Source: Google Android Team, 2024. *Comprehensive Rust*.

Technological Challenges:

- ❖ Memory model compatibility
- ❖ Pointer stability
- ❖ Lifetime complexity
- ❖ Generation and verification

Some Recent Contributions to these Gaps

Modularity

- Eclipse OMR-based GC [CASCON 2021]

Performance

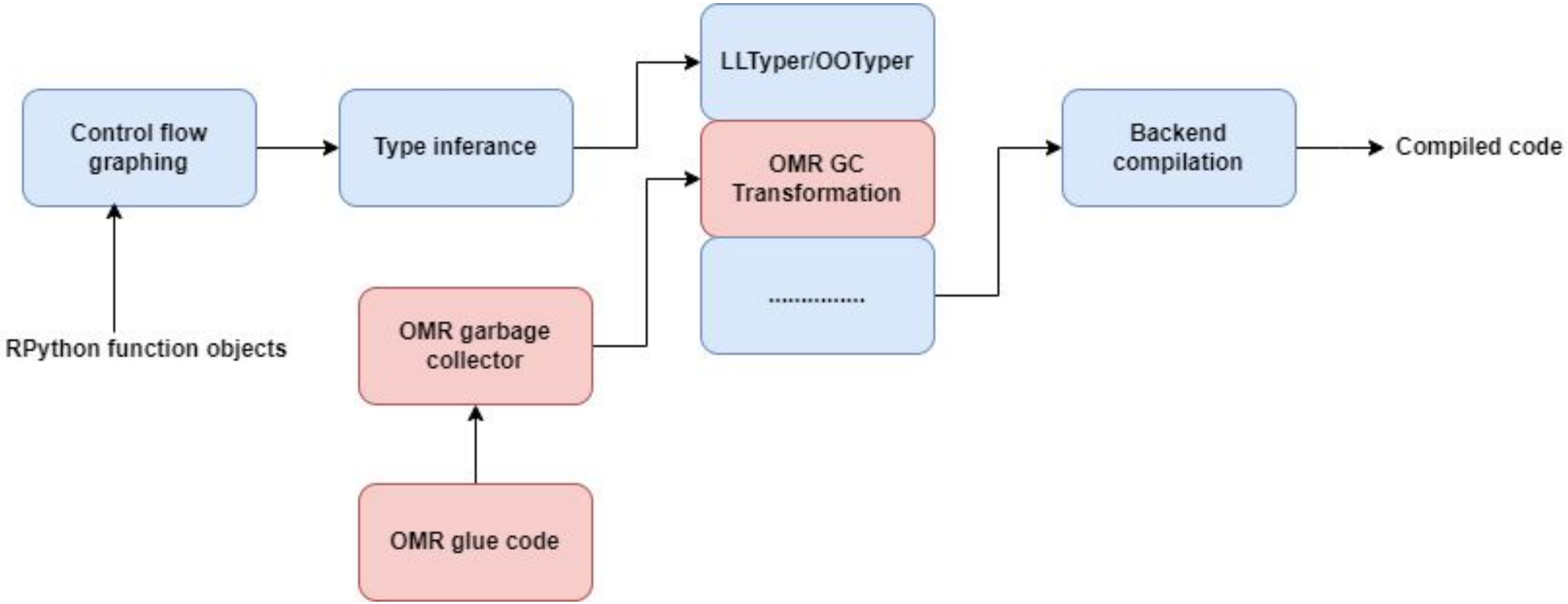
- Understanding the GC Cost [CASCON 2022]
- Optimal JIT Trace Sizing [US Patent 2023]
- Type-based Stores and Context Aware Presizing [JOT 2024]
- Type-based Stores [DEF, Anti-patent 2024]
- Context Aware Presizing [DEF, Anti-patent 2024]

Support for Native Code

- CyStck and Migration Tooling [ICOOOLPS 2023]
  Artifacts Available / v1.1

DEF: Defensive Publication

IBM OMR GC Modularity for RPython VMs



Joannah Nanjekye, David Bremner, and Aleksandar Micic. 2021. *Eclipse OMR garbage collection for tracing JIT-based virtual machines*. **CASCON '21**. IBM Corp., USA, 244–249

Raw Code Metrics, LCOM and MI

	OMR Garbage Collector	RPython Garbage Collector
Number of methods	22	26
Number of bytes	11.5kb	26kb
LOC	338	630
LLOC	224	538
SLOC	249	567
$\sum CC$	42	98
$\overline{CC}$	1.75	3.5

	OMR GC	Framework GC
Lack of cohesion of methods (LCOM)	2	1.5
Maintainability Index (MI)	31.28, A	14.12, B

$$MI = max\{0, 100\frac{171 - 5.2\ln V - 0.23G - 16.2\ln L + 50\sin\sqrt{2.4C}}{171}\}$$

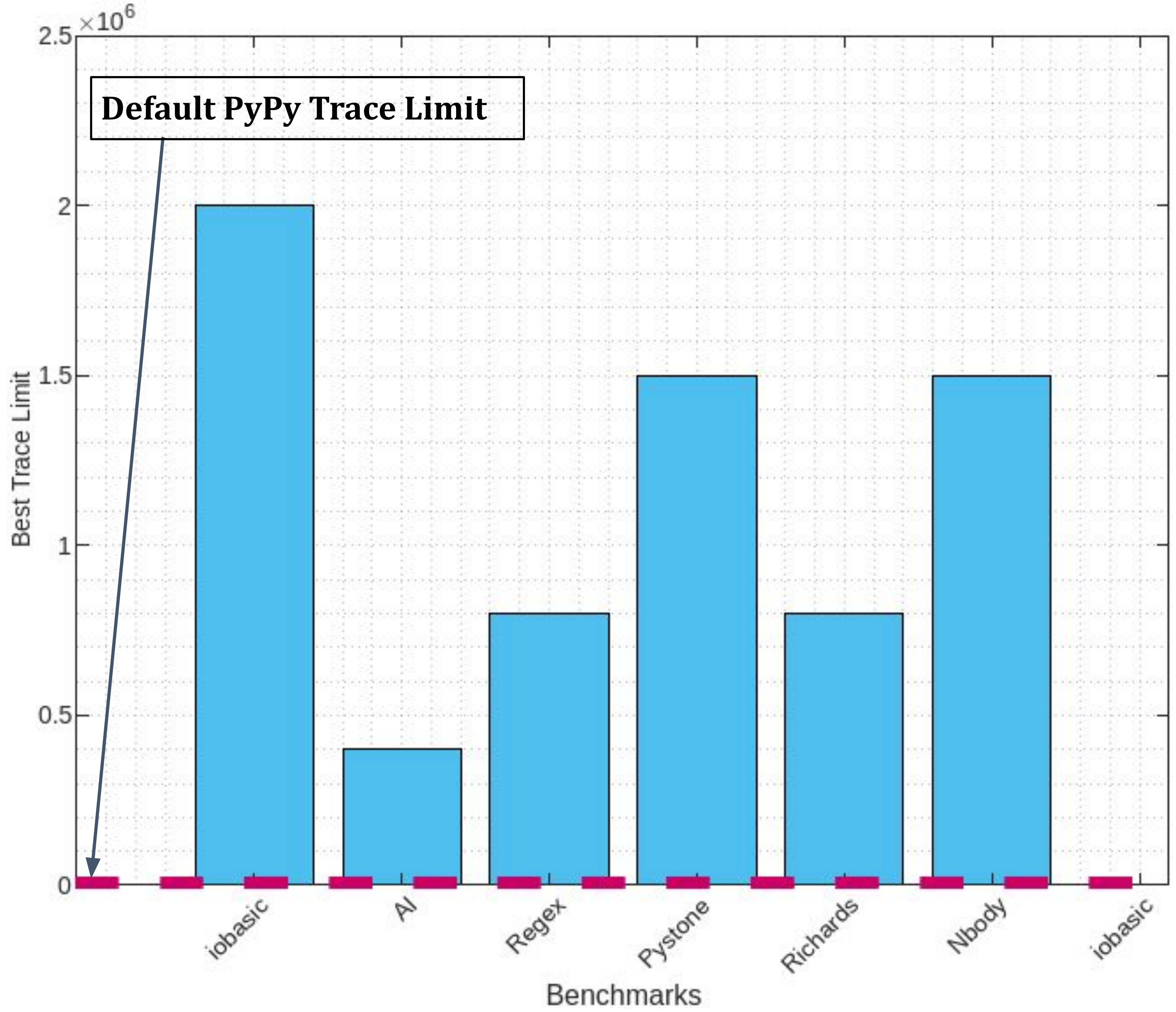
Where:

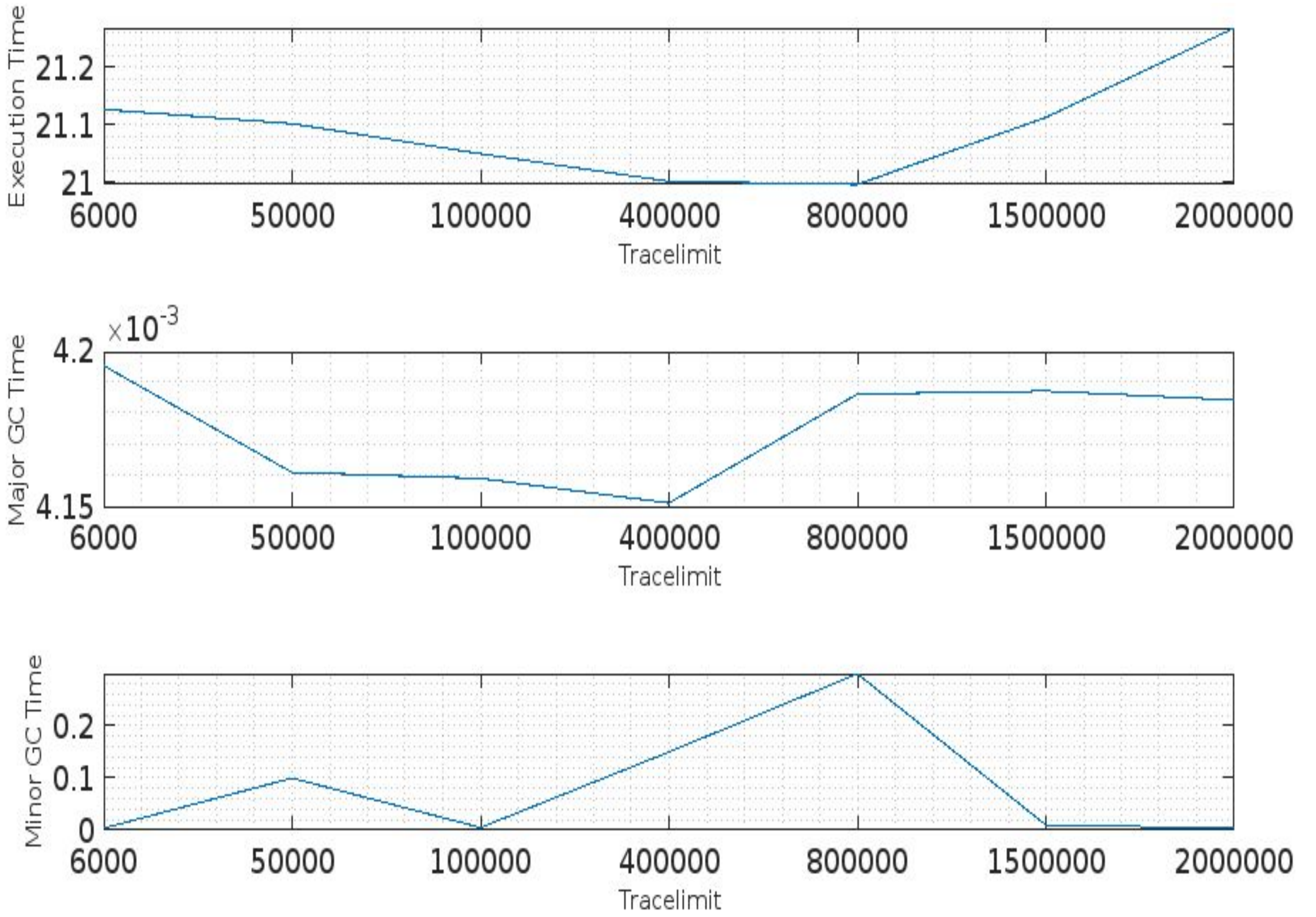
(1) V = Halstead volume
(2) G = Total cyclomatic complexity
(3) L = SLOC
(4) C = Percentage of comment lines in radians

A (20 - 100): Good, B (10 - 19): Moderate, C (0 - 9): Low

Joannah Nanjekye, David Bremner, and Aleksandar Micic. 2021. *Eclipse OMR garbage collection for tracing JIT-based virtual machines*. **CASCON '21**. IBM Corp., USA, 244–249

JIT Tracing and Garbage Collection





H1: The effective or best trace limit is application specific

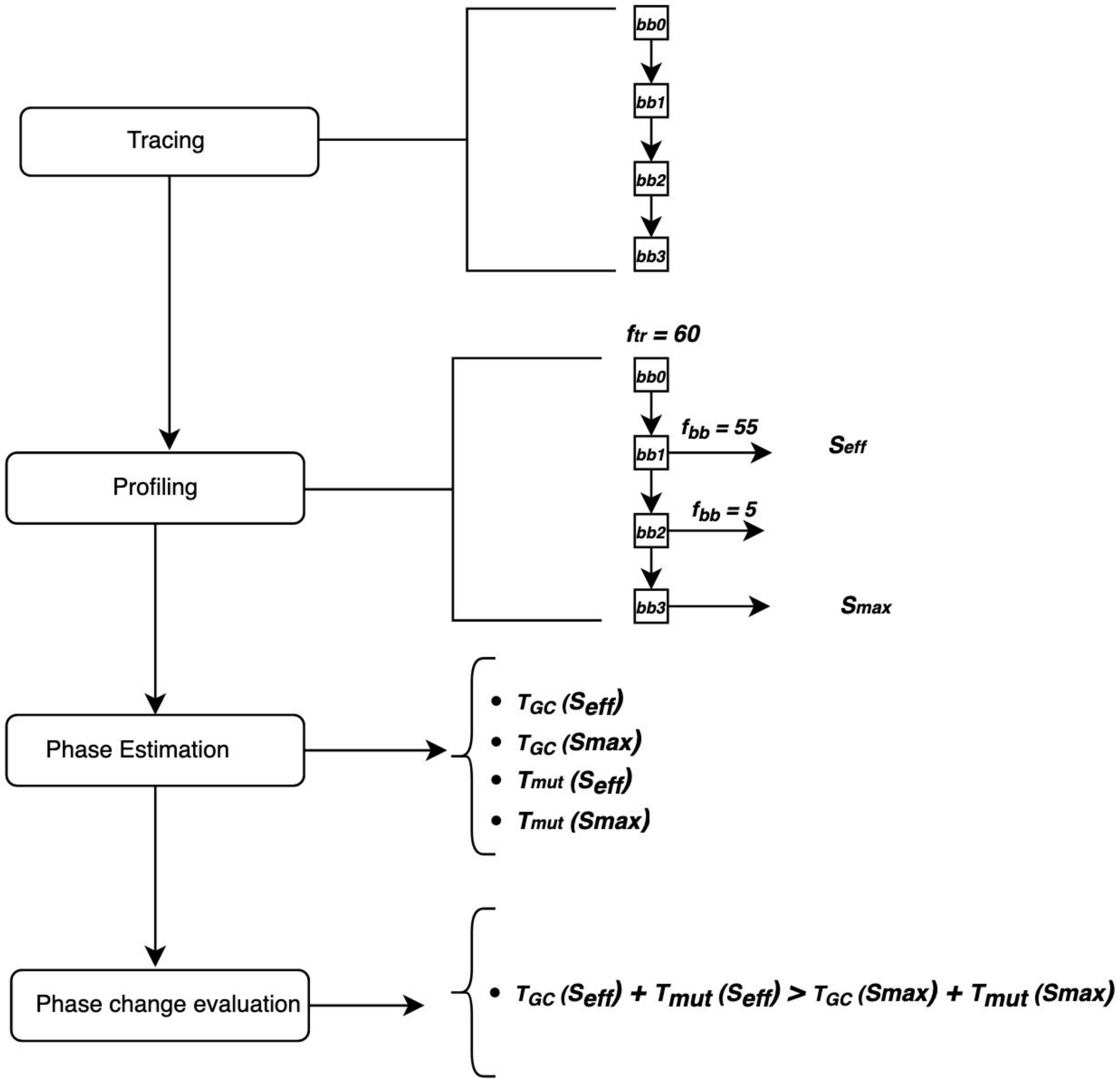
H2: Increasing the trace limit improves performance to a degree, after which GC pressure degrades it

Joannah Nanjekye, David Bremner, and Aleksandar Micic. The Garbage Collection Cost For Meta-Tracing JIT-based Dynamic Languages. **CASCON '22**. IBM Corp., USA, 140–149

Optimal Trace Sizing for Virtual Machines

We propose a technique that utilizes profiling information during formation of a trace:

- A new trace is not compiled immediately, it is *profiled* first
- We identify hot exits of the trace, which is the *effective trace size* estimation phase
- We then estimate the *total execution time* for a program at this trace size
- The estimated total execution time at this trace size, can be used to decide to either continue trace formation, or trigger a trace abort



Joannah Nanjekye, David Bremner, and Aleksandar Micic. The Garbage Collection Cost For Meta-Tracing JIT-based Dynamic Languages. **CASCON '22**. IBM Corp., USA, 140–149

Language C API and Garbage Collection

Objects are in form of a C struct. The C APIs have the following challenges:

- A non-moving object model
- Non-opaque Object structs
- Tight coupling with GC implementation
- Borrowed references

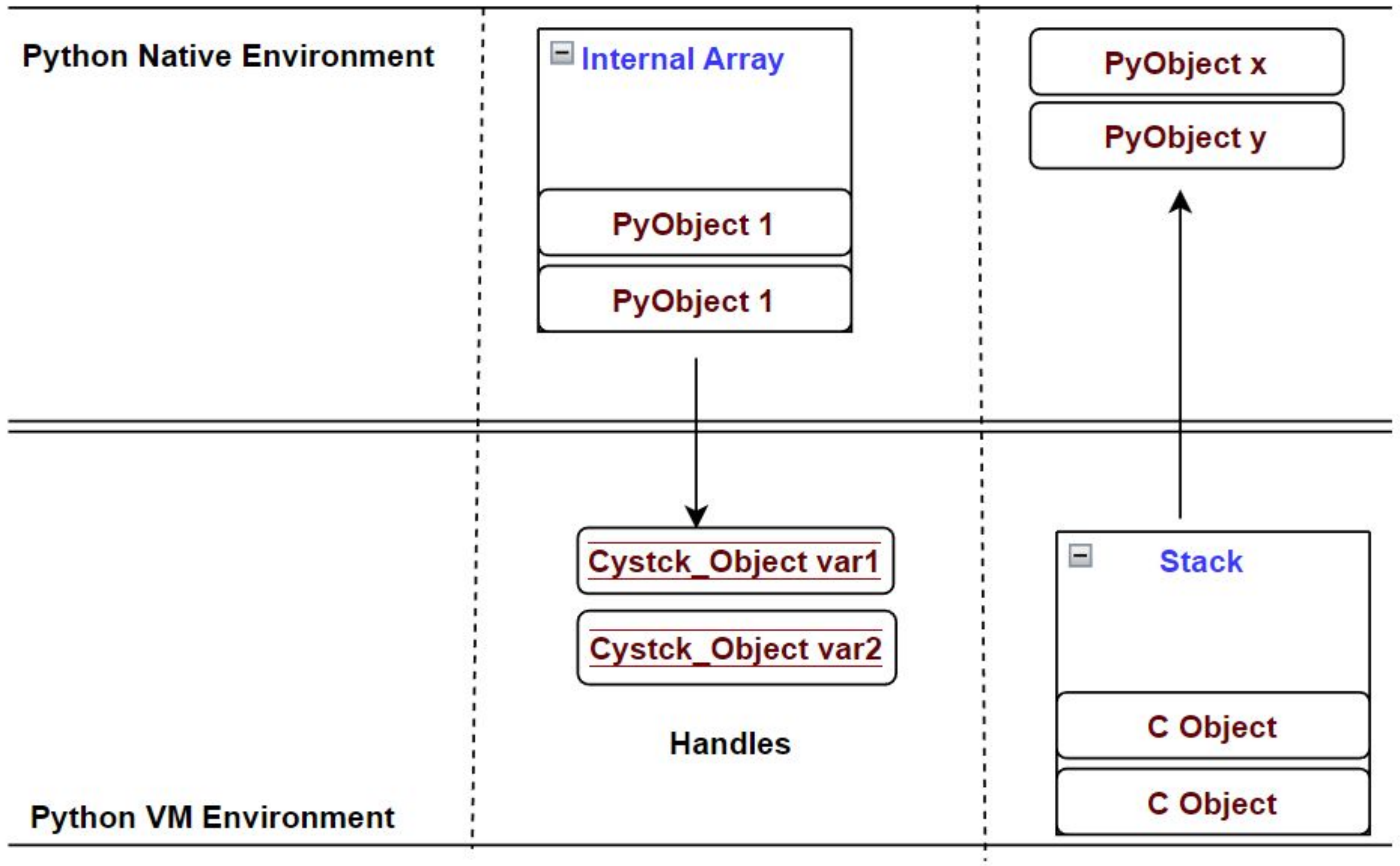
Main Problems: Pointer Stability, Lifetime Complexity, Memory Model Compatibility

Joannah Nanjekye, David Bremner, and Aleksandar Micic. 2023. Towards Reliable Memory Management for Python Native Extensions. **ICOOOLPS 2023**. ACM, USA, 15–26

CySteck

A new alternate stack-based C API for Python as a solution:

- We combine a *stack* and *light-weight* handles
  - The stack and handles are used for communication between C and Python
  - As well as aid with garbage collection
- CyStck provides scope gates for functions that may generate many objects
- For object lifetime management we use:
  - A manual reference mechanism
  - Process introspection



Another Problem: Reachability alone is not enough to determine when to collect an object

Joannah Nanjekye, David Bremner, and Aleksandar Micic. 2023. Towards Reliable Memory Management for Python Native Extensions. ICOOOLPS 2023. ACM, USA, 15–26

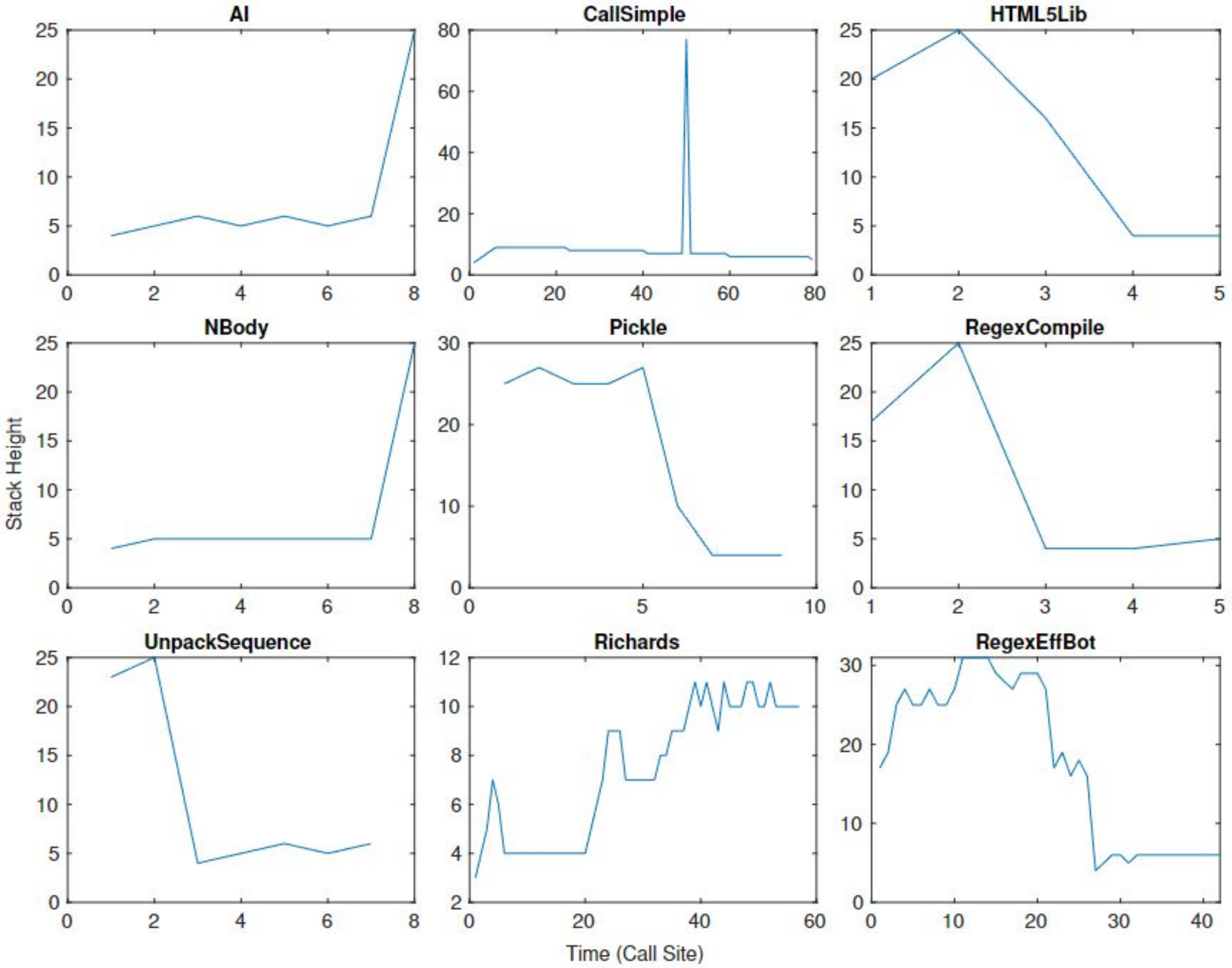
Process Introspection

Algorithm 4: Deallocation: ObjectLifeTimeAnalysis(obj)

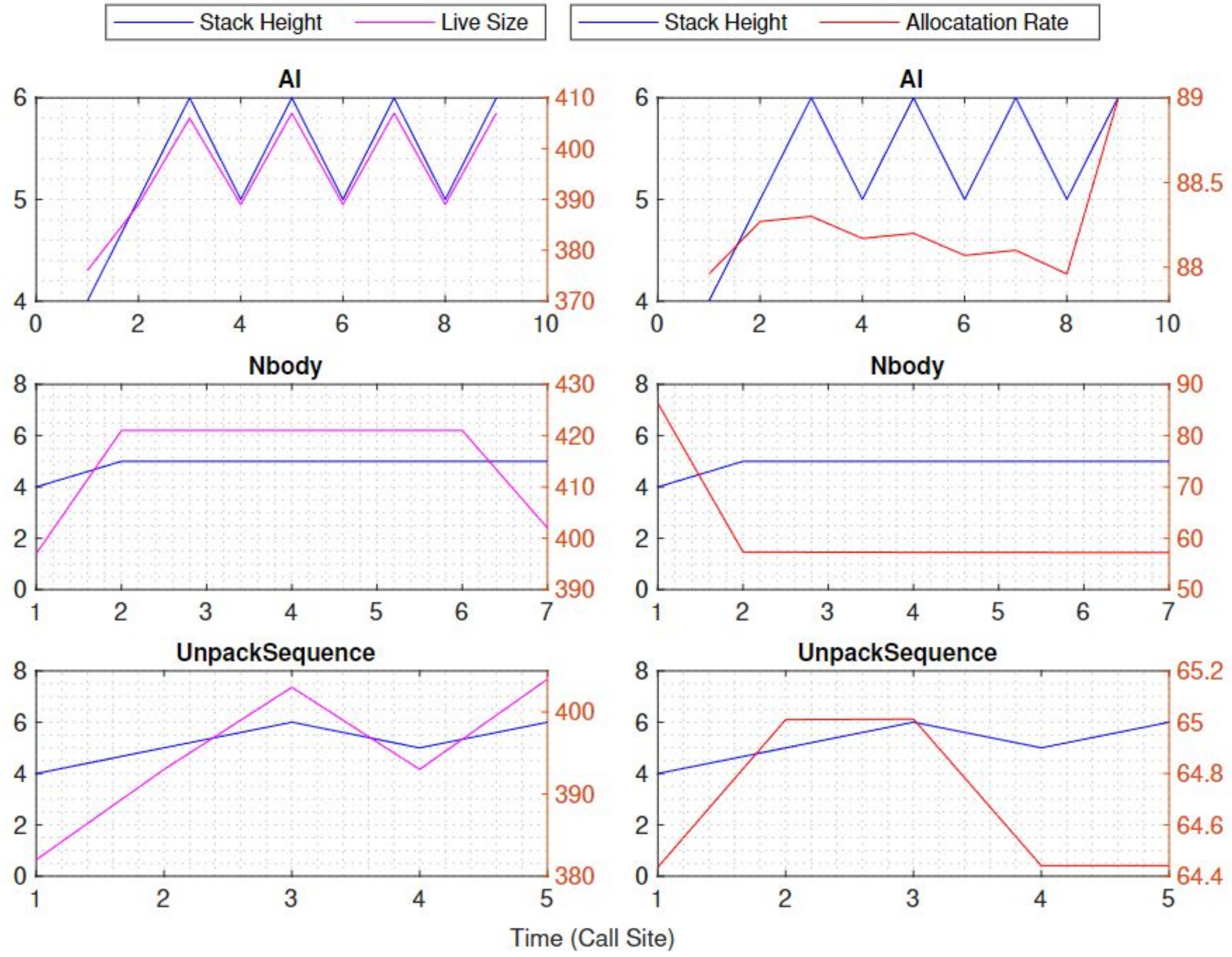
```
Data: Input: Let obj be the object
Result: An accurate deallocation of obj
use liballocs.h;
/*deallocate an object*/;
if CreatedFromPython(obj) then
    if refcount == 0 then
        detachRefCountPolicy();
        free(obj)();
    end
end
if !CreatedFromPython(obj) then
    if refcount == 0 then
        detachRefCountPolicy(obj);
    end
    if isExplicitFreeCalled() then
        detachExplicitFreePolicy(obj);
    end
    if !has_policy(obj) then
        free(obj);
    end
end
```

Joannah Nanjekye, David Bremner, and Aleksandar Micic. 2023. Towards Reliable Memory Management for Python Native Extensions. **ICOOOLPS 2023**. ACM, USA, 15–26

Garbage Collection and Phase Analysis



Live Size and Allocation Rate



Potential GC Work Paths

Optimal Heap Limits: The heap limit algorithm by Kirisame et al., can be modified to be based on the stack height instead of live size:

$$M = kS + \sqrt{kSg/cs}$$

Reduced Pause Time: Phase-based GC triggering based on the stack height can be investigated with server workloads to reduce server timeout for requests

Memory Safety of FFIs: Towards better VM/C++ interoperability, study memory safety concerns and write validation tools that are able to isolate safety issues

Eclipse OMR-based GC
[CASCON 2021]

Modularity

Understanding the GC Cost
[CASCON 2022]

Optimal JIT Trace Sizing
[US Patent 2023]

Type-based Stores and Context Aware Presizing
[JOT 2024]

Type-based Stores
[DEF, Anti-patent 2024]

Context Aware Presizing
[DEF, Anti-patent 2024]

Performance

CyStck and Migration Tool
[ICOOOLPS 2023]

Artifacts Available / v1.1

Support for Native Code

Thank You!

DEF: Defensive Publication

1. [Reference Counting]. Simulate reference counting with this Java-like code, tracking allocated objects, reference counts, and deallocation:

```
class Link {
    int value;
    Link next;
}

Link makeList() {
    Link x = new Link();
    x.next = new Link();
    Link y = new Link();
    x.next = y;
    return y;
}

Link z = makeList();
z.next = z;
z = null;
```

17-363/17-663: Programming Language Pragmatics, In-Class Exercises October 21, 2024
Andrew ID: ______________________________

2. **[Copy Collection].** Simulate copy collection on the following memory from-space, and show the resulting to-space. We've simplified things so that the heap has only pairs in it, and each pair has an additional "forwarding address" space. That means every heap location is a multiple of 3. We reserve 0 for the null pointer. Assume all non-zero values are pointers. Assume we have one global variable, x.

Value of global variable x: address 3

From-Space:

From-Space:

Address	**Forwarding address**	**First**	**Second**
0			
3		9	15
6		12	15
9		9	0
12		6	0
15		18	0
18		0	15

----------------------------- After Copy Collection ----------------------------

Value of global variable x: ____________

To-Space:

Address	**Forwarding address**	**First**	**Second**
30			
33			
36			
39			
42			
45			
48			