Reconsidering Garbage Collection in Julia

A Practitioner Report

Luis Eduardo de Souza Amorim
Australian National University
Canberra, Australia
luiseduardo.desouzaamorim@anu.edu.au

Yi Lin
Australian National University
Canberra, Australia
yi.lin@anu.edu.au

Stephen M. Blackburn
Google
Sydney, Australia
Australian National University
Canberra, Australia
steveblackburn@google.com

Diogo Netto
RelationalAI
Berkeley, USA
diogo.netto@relational.ai

Gabriel Baraldi
JuliaHub
Cambridge, USA
gabriel.baraldi@juliahub.com

Nathan Daly
RelationalAI
Berkeley, USA
nathan.daly@relational.ai

Antony L. Hosking
Australian National University
Canberra, Australia
antony.hosking@anu.edu.au

Kiran Pamnany
RelationalAI
Berkeley, USA
kiran.pamnany@relational.ai

Oscar Smith
JuliaHub
Cambridge, USA
oscar.smith@juliahub.com

Abstract

Julia is a dynamically-typed garbage-collected language designed for high performance. Julia has a non-moving tracing collector, which, while performant, is subject to the same unavoidable fragmentation and lack of locality as all other non-moving collectors. In this work, we refactor the Julia runtime with the goal of supporting different garbage collectors, including copying collectors. Rather than integrate a specific collector implementation, we implement a third-party heap interface that allows Julia to work with various collectors, and use that to implement a series of increasingly more advanced designs. Our description of this process sheds light on Julia's existing collector and the challenges of implementing copying garbage collection in a mature, high-performance runtime.

We have successfully implemented a third-party heap interface for Julia and demonstrated its utility through integration with the MMTk garbage collection framework. We hope that this account of our multi-year effort will be useful both within the Julia community and the garbage collection research community, as well as providing insights and guidance for future language implementers on how to achieve high-performance garbage collection in a highly-tuned language runtime.

ISMM '25, Seoul, Republic of Korea

ACM ISBN 979-8-4007-1610-2/25/06
https://doi.org/10.1145/3735950.3735957

CCS Concepts: • **Software and its engineering** → *Garbage collection*; *Software evolution.*

Keywords: Garbage Collection, Memory Management, Julia

ACM Reference Format:
Luis Eduardo de Souza Amorim, Yi Lin, Stephen M. Blackburn, Diogo Netto, Gabriel Baraldi, Nathan Daly, Antony L. Hosking, Kiran Pamnany, and Oscar Smith. 2025. Reconsidering Garbage Collection in Julia: A Practitioner Report. In *Proceedings of the 2025 ACM SIGPLAN International Symposium on Memory Management (ISMM '25), June 17, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3735950.3735957

1 Introduction

Julia aims to combine productivity with performance, particularly in domains such as large-scale numeric and scientific computing [4]. Julia achieves its productivity goals by providing language features that open it to a wide audience, including scientists and others who seek an expressive, easy-to-use language environment without compromising performance [3]. The Julia compiler uses powerful type inference to aggressively optimize code with LLVM using JIT compilation as well as AOT compilation to system images.

Julia is garbage collected, abstracting away the complexity of manual memory management. It also provides low-overhead native interfaces that allow efficient integration with high-performance third-party libraries. Julia's stock garbage collector is a highly-tuned segregated-fits mark-sweep collector. While this has served the language well, it is unavoidably exposed to the same problems as all non-moving garbage collectors: fragmentation and loss of locality. As Julia's popularity has grown, the limitations of being tied to a single non-moving garbage collector have been exposed,

as inevitably users encounter workloads where such a collector is a poor fit.

In 2022 we embarked on what became a multi-year collaboration to refactor Julia to support third-party garbage collectors. Rather than integrate a specific collector implementation, our goal was to create a pluggable API that would support a *range of collectors*, including copying collectors. This was non-trivial since the implementation of Julia's stock garbage collector was deeply integrated into the runtime with pervasive optimizations, many of which are implicitly tied to assumptions about the GC implementation itself. Most notable among them is that no object will move. This paper is a practitioner report describing the process of refactoring Julia, trying to understand and untangle those assumptions, with the final goal of supporting multiple garbage collectors, including copying collectors. As such, it provides numerous insights into Julia's implementation specifically, and into the challenges of GC+runtime integration more generally.

We have successfully refactored Julia to include a garbage collection interface and support multiple collectors with richer semantics, such as copying collectors. We demonstrate this capability concretely by integrating with the MMTk garbage collection framework. We faced many technical challenges along the way, some unique to Julia, but many that are likely transferable to other high-performance runtimes. We hope that this report will be useful for the memory management community, as well as the Julia developer community and language developers more broadly.

2 Background

In this section, we introduce Julia, especially its garbage collector design and implementation. We also introduce the MMTk framework.

2.1 Julia

Julia is a high-level, high-performance dynamic programming language initially designed for numeric computing [2–4]. Julia was designed to bridge the gap between the productivity of high-level dynamic languages like Python and the execution speed of lower-level languages like C [3]. Its design centers on providing both an expressive syntax and the performance required for demanding numeric applications.

At the heart of Julia's performance is its JIT compiler, backed up by LLVM, which transforms high-level code into optimized machine code at run time. By leveraging the LLVM framework, the compiler performs aggressive optimizations tailored to the exact types encountered during execution. This dynamic specialization, underpinned by powerful type inference, allows Julia to minimize execution overhead, delivering speed that is competitive with traditionally compiled languages like C or Fortran [3].

Julia addresses long compilation time resulting from its sophisticated JIT by allowing a precompiled system image — a snapshot of the runtime that includes the core language, essential libraries, and frequently used packages. This system image is generated ahead-of-time when building Julia, allowing the language to bypass much of the initial type inference and method specialization that normally occurs during the first invocation of functions. Moreover, Julia also allows packages to be compiled ahead-of-time, like the system image, and allows loading precompiled package images similar to native shared libraries. These images dramatically reduce the overhead of JIT compilation during startup for Julia.

The language also features modern concurrency and parallelism paradigms, providing native support for multithreading, coroutines, asynchronous programming, and distributed computing. Its lightweight tasks and channels enable developers to write asynchronous code that can efficiently manage I/O-bound operations or take advantage of multi-core processors.

While initially designed for numerical computing, Julia has evolved into a true general-purpose language. Its expressive syntax, extensive standard library, and seamless interoperability with languages like C, Python and R allow developers to build a wide range of applications such as web services, scripts, command-line tools and system-level software. This versatility, combined with the growing ecosystem of packages and a dynamic community, positions Julia as a powerful platform not only for high-performance scientific computing but also for broader software development challenges.

2.2 Julia's Garbage Collector

Julia has a non-moving, partially concurrent, parallel, generational and precise mark-sweep collector [15].

2.2.1 Allocation and collection.

Like many runtimes, Julia employs different strategies for small and large objects. Small objects are allocated with a segregated-fits free-list allocator, with each underlying 16 KB memory region, known as a 'page', consisting of objects of the same object size class. Memory pages that are found to be empty during a collection may be reused for allocation before they are lazily returned to the operating system using `madvise`. Metadata describing each page's state resides outside the heap. Large objects and some native data structures that back up certain Julia types are allocated with `malloc`. Those objects are considered part of the Julia heap, and are counted into the heap size. `Malloc`'d objects are maintained in thread-local linked lists for the collector to traverse.

Julia's collector performs the standard mark and sweep phases [22]. It performs marking in parallel [11]. During the sweep, the collector identifies pages that no longer contain live objects and can be freed, and it frees `malloc`'d memory. Freed pages will be lazily returned to the operating system, but may be reused before that happens [10].

2.2.2 Generational GC. As Julia's GC is non-moving, surviving objects cannot be copied, therefore an object's age cannot be determined by the memory region in which it resides. Instead, Julia uses sticky mark bits to determine object generation [16].

Julia implements sticky mark bits by using two bits in each object's header to indicate that it has one of the following states: `GC_CLEAN`, `GC_OLD`, `GC_MARKED`, and `GC_OLD_MARKED`. Freshly allocated objects are tagged with `GC_CLEAN`. During GC, the objects that are reachable during tracing are tagged as `GC_MARKED`. Julia uses the `GC_OLD` bit to tag objects that have survived at least one GC cycle. The `GC_OLD_MARKED` tag is used to indicate that an old object has been marked in the current GC cycle. After each collection, it updates the status of all `GC_MARKED` objects to `GC_OLD`, and after a full heap GC it also resets the status of all `GC_OLD_MARKED` objects to `GC_OLD`. Write barriers may also change the `GC_OLD_MARKED` objects into `GC_MARKED`. This allows GC to efficiently identify and collect young objects while preserving old objects that are still reachable. Note that some objects, such as permanent objects and objects in Julia's boot image, are tagged as `GC_OLD_MARKED` after allocation.

Julia intercepts the field write to old objects to build a remembered set used in nursery collections. The write barriers ensure that the GC can identify all objects in the old generation that have references to the young generation, adding those objects to a per-thread remembered set. There are different types of write barriers optimized for specific cases, including, for example, array copying barriers.

2.2.3 Roots. Tracing collectors define the liveness of objects in terms of their reachability from *roots* in the language runtime, such as stacks and global variables. The Julia runtime's handling of roots is distinctive in two respects: i) it uses *shadow stacks* [18], and ii) it elides 'unnecessary' reporting of *dominated roots*. Within the runtime, root-referenced objects can be reported to the GC by adding them to the global roots table, or if they have a known temporal scope, they can be pushed to and popped from the shadow stacks.

Shadow Stacks. Because LLVM's support for precise stack scanning is limited, Julia uses shadow stacks to identify stack roots [18]. The shadow stacks report a precise set of *runtime references to heap objects* to the garbage collector. Although the shadow stacks report the location of the root references, and thus in principle allow the root references to be updated if the collector were to move the referent, in fact the collector cannot move these referents. The reason for this is that although the shadow stacks are precise and complete with respect to the set of root *referents*, they are not complete with respect to the set of root *references*. Completeness with respect to root referent objects ensures that liveness can be soundly maintained. However, lack of completeness with respect to the set of root references means that referent objects cannot be moved since not all references to a moved object are guaranteed to be updated. This means that stack-referenced objects cannot be moved and must therefore be pinned by the garbage collector.

Dominated Roots. As an optimization, Julia introduces the concept of dominated roots and allows developers working on the runtime to elide explicitly reporting them. Unfortunately, this significantly complicates the implementation of moving garbage collection. The key idea is that some directly root-referenced objects are *also transitively* root-reachable via, and thus dominated by, some other directly root-referenced object. In principle, in terms of liveness at least, it is redundant to report an object as root-referenced if the developer knows that it is dominated by some other root-referenced object. This optimization saves the explicit reporting of some roots, and is sound as long as all dominator roots are reported.

In Julia parlance, 'rooting' means explicitly identifying a root-referenced object to the collector (by either pushing the reference to a shadow stack, or adding it to the global roots table) and 'roots' are the subset of root-referenced objects that are thus identified. It is a common practice in the Julia runtime that a developer determines that an object has been 'rooted', and then freely uses objects that are transitively reachable from it without explicitly 'rooting' them, including storing references to them in any runtime context (runtime data structures, globals, or code). Consequently, *the root sets do not reflect the complete set of references held by the runtime.*

This approach is correct for a non-moving collector, since it guarantees the object graph is fully traversed. However, this means that the collector may not be aware of all references to a given object, which would prevent such an object from being moved. This presents a significant challenge for a copying collector, which we discuss in detail later.

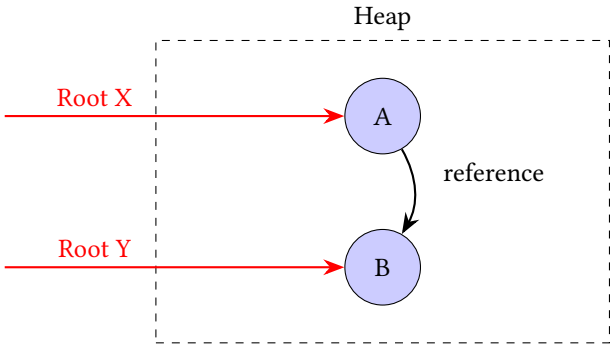
For the sake of clarity, we will use the conventional definition of 'roots' — the set of all non-heap locations within the runtime that contain references into the heap [19] — rather than Julia's definition. When necessary, we'll distinguish between dominant and dominated roots as per the description above.

2.2.4 Foreign Function Interface. Julia's Foreign Function Interface (FFI) is engineered to enable near-zero-overhead calls and facilitates seamless integration of native code into Julia applications. It is common practice for Julia to pervasively use mature and optimized native libraries where possible, including when implementing its standard library.

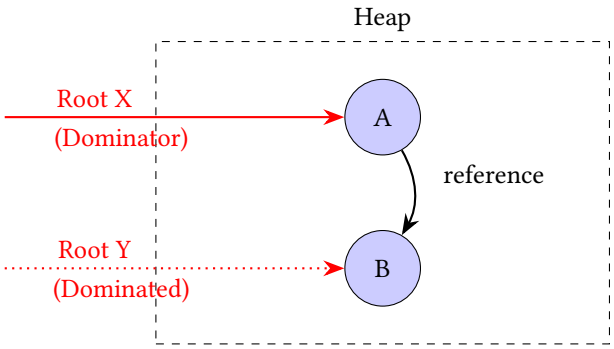
A critical aspect that enables this FFI design, however, is its underlying assumption of a non-moving garbage collector. Native code accesses Julia heap objects through direct pointers rather than handles that could provide a level of indirection and have sufficient information for the garbage collector to guarantee valid access. Under the assumption of non-moving collection, the pointer to a heap object is considered valid as long as the object is alive, thus Julia FFI

only requires the user code to make sure that the object exists when the pointer to it is exposed to native code. In some scenarios, this process can be totally transparent to the garbage collector. For example, code may store an object in any global variable to make sure it is alive, then pass the pointer to native code where the pointer is saved inside native data structures, and then return from the native call. The pointer is a root for the garbage collector, but the collector is oblivious to the reference from the native code. This is valid Julia code.

Furthermore, Julia lacks a defined semantics for some parts of the FFI. For example, we were not able to find documentation describing Julia's GC-safe region and GC-safe calls, leading to a very loose definition of such concepts based on their implementation, and each specific use case. This, combined with the assumption of non-moving collection, leads to the existence of a range of gray areas where behavior is observed to work but is not guaranteed by a formal contract.



(a) References from outside the heap to heap objects are roots.



(b) A Julia developer may elide reporting Root Y if they know its referent, B, is reachable from Root X.

Figure 1. Julia developers may elide reporting roots when they know that the referent is *dominated* by another root. The elision of non-dominant roots added significant complexity to the project.

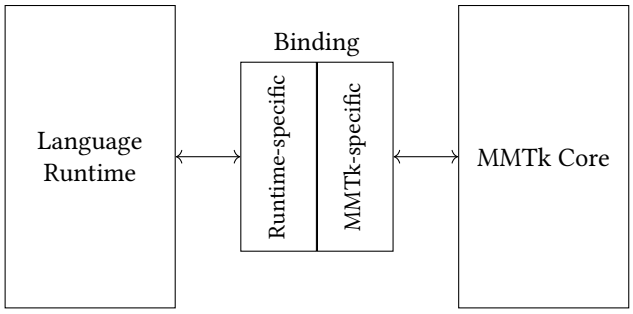


Figure 2. A language runtime works with MMTk through a binding which consists of two parts that are respectively runtime-specific and MMTk-specific.

2.3 MMTk

The goal of our work is to refactor Julia's memory manager to support a third-party heap interface that will allow integration of various collectors. For concreteness, we test and evaluate our work with respect to MMTk.

MMTk is a high-performance language-agnostic memory management framework [5, 6]. MMTk features an abstract interface for garbage collection that externally suits different languages and internally suits different garbage collection algorithms (referred to as *GC plans* or simply *plans* in MMTk). The modular internal design for GC components in MMTk leads to easier implementation of GC algorithms and innovations.

The project started as a portable garbage collector for JikesRVM in 2004. Although the original implementation was ported to a number of runtimes [17], it was fundamentally limited by its use of a particular fully-static dialect of Java [6]. It was rewritten in Rust in 2016. Since then various language runtimes have been ported to MMTk. The ports serve as credible high-performance research platforms (OpenJDK), proof-of-concept prototypes (Android, JikesRVM, V8 and Haskell), or alternative GC implementations that complement the shortcomings of the original GCs (CRuby).

Immix. Immix employs a mark-region heap design that combines the efficiency of region-based memory management with the flexibility of mark-sweep collection, enabling both fast allocation and effective memory reuse [7]. MMTk includes generational variants (`GenImmix` with a copying nursery and `StickyImmix` with sticky mark bits), and is the basis for the high performance LXR collector [28]. Object copying in Immix is opportunistic and by default, is only performed during defragmentation collections. Copying can be disabled for Immix, and the collector without moving still works correctly (with performance loss). Immix naturally supports object pinning. Pinned objects are guaranteed to stay in place and will never be moved.

Language bindings. MMTk refers to its integration with a language runtime as a *binding*, as illustrated in Figure 2.

Since MMTk is language-agnostic, the binding bridges the abstract semantics of MMTk and the specific implementation of the runtime. A binding has two components: one of which is a logical extension of the runtime, written in the same language as the runtime, but which is MMTk-specific, and the other a logical extension of MMTk, written in Rust, but which is language-specific.

3 Reworking Julia's GC

We now describe the journey we took to refactor Julia, as we added support for third-party garbage collectors. We started by implementing a simple non-collecting system called "No GC", then gradually added more capabilities, first with a non-moving collector, then a non-moving generational collector, and finally by adding support for copying objects.

3.1 No GC

A "No GC" collector supports memory allocation but does not implement any memory reclamation mechanism. When the heap is exhausted, the collector simply shuts down the process with an error message. OpenJDK's Epsilon collector [25] is a "No GC" collector. MMTk includes a `NoGC` plan in its suite.

Starting a port with a "No GC" collector is generally a good idea to avoid handling multiple complexities at once — developers need only focus on allocation at this stage. Getting `NoGC` working also provides a solid baseline for both correctness and performance for future work.

3.1.1 Allocation. The key to supporting `NoGC` is to identify the semantics of all the allocation sites. This allows third-party collectors to provide their own implementation for allocation. The stock Julia GC uses two allocators: a free list allocator and `malloc`, which are used for multiple purposes. In the following, we outline our understanding of the abstract semantics of the allocations in Julia, and match these to MMTk's allocation semantics [26].

Small objects Julia allocates small objects up to 2 KB with its free list allocator. This handles most of the object allocations, and needs to be efficient. It is straightforward for us to use the `Default` allocation semantics from MMTk for small object allocation. The actual allocator used for the `Default` allocation semantics is specific to the active MMTk GC plan, such as a bump pointer allocator into a copying nursery space for generational plans, or a free-list allocator for the mark-sweep plan. However, the underlying allocator mechanism is not a concern for developers. Instead, they should find the right semantics for their allocation needs.

Large objects Julia uses `malloc` to allocate large objects, and maintains a list of large objects in the allocator so its collector can identify dead large objects and free the `malloc`'d memory. We use the Large Object Space (`LOS`) semantics to allocate large objects in MMTk.

Permanent allocation Julia allocates permanent memory that should not be collected. It does this in two ways: by allocating permanent raw memory (`jl_gc_perm_alloc`), and by allocating permanent objects (`jl_gc_permobj`). The former allocates raw memory that can be used by runtime data structures, or as system images, and the latter holds exactly one Julia object with a header and related object metadata.[1] We forward both functions to MMTk using its `Immortal` semantics. The `Immortal` semantics guarantees that the objects are allocated within the MMTk heap and will not be reclaimed. We selectively invoke MMTk's `post_alloc`, which initializes per-object metadata, when the permanent allocation is for an object (`jl_gc_permobj`).

Malloc (including counted malloc) Julia both uses and exposes `malloc` functions. An interesting `malloc` variant that Julia uses is called *counted malloc*, which calls `malloc` and counts the allocated bytes as part of the heap size. Julia exposes those functions to users, and also uses them in the native libraries that back up certain Julia types in the Julia standard library. For example, Julia uses `GMP` to implement `BigInt` and uses *counted malloc* to allocate for `GMP` so those native allocations are accounted to the Julia heap. We maintain a counter for the live bytes from `malloc`, and report these live bytes to MMTk as "off-heap memory" (not allocated by MMTk) that is considered as part of the heap.

By abstracting and mapping the semantics of Julia allocations, we clarify exactly what third-party collectors are required to implement to meet Julia's allocation needs. As we use abstract semantics for MMTk, the allocation implementation works not only for `NoGC` but also for the other collectors that will be discussed in this paper.

3.1.2 Code Generation. Julia uses LLVM as its compiler backend, and defines a few custom LLVM passes that perform GC-related analysis and optimizations called `LateGCLowering` and `FinalGCLowering`. Most of the code in these passes is GC-agnostic and can be shared among different GC implementations: for instance, when identifying live variables and emitting GC-safe points. There is GC-specific code for which a third-party collector needs to provide its own implementation. This includes emitting the allocation sequence and write barriers. Allowing GC-specific code in these passes is essential. Our refactored GC interface conditionally compiles different source code based on the GC in use, allowing

[1] Interestingly, during our work we identified one particular type of permanent object (`jl_sym_t`) incorrectly allocated via `jl_gc_perm_alloc` rather than `jl_gc_permobj`. The object's metadata was properly initialized, so it appeared to function correctly despite this error. We submitted a pull request and fixed the inconsistency.

each GC to provide implementations for a predefined set of GC-specific functions.

The passes emit a call to the allocation function that does the allocation for Julia's own GC. This is a correct implementation for third-party collectors as well, since they are required to implement their own allocation function. Nonetheless, we chose to directly emit the instruction sequence for our allocator's fast-path for better allocation throughput. The GC plans that we support and describe in this paper all share the same bump pointer allocation sequence which is usually just under 20 instructions. Letting the compiler inline the allocation fast-path enables substantial performance gains with a low cost in terms of its implementation.

We also use the same mechanism in code generation to implement write barriers for the sticky Immix plan, and different `GC.@preserve` semantics for copying plans as we will see in Sections 3.3 and 3.4.

3.1.3 Thread-Local Storage.

A mutator thread includes thread-local storage for the allocators and the collectors. For example, Julia's stock GC includes various thread-local free lists of different size classes, metadata for pages used by the free list allocator, a linked list of large objects allocated by `malloc`, a remembered set, and a few other data structures used by the collector. Clearly, those are not generic enough for a build with third-party GCs. Similar to our approach for GC-specific code in code generation, we conditionally include a definition of thread-local storage specific to each GC implementation.

3.2 Non-Moving Immix

Implementation of `NoGC` laid a good foundation for most aspects of the GC interface by isolating and separating GC-specific code. We set our next goal as supporting a properly garbage-collected plan in MMTk.

We chose a variant of `Immix`, *non-moving Immix*. Immix is an ideal target for this stage of our work, as it is relatively simple, yet allows practical use and leads to easier adoption of more advanced GC algorithms based on Immix. Because copying in Immix is opportunistic, we were able to disable any form of copying at this stage of work, avoiding the extra complexity from allowing object movement in Julia, since the language was not originally implemented with copying collection in mind (as we will discuss in more detail later).

Nonetheless, non-moving Immix is a fundamentally limited algorithm that we chose just for this early stage of the work. Without copying, there is no defragmentation, leading to inevitable heap fragmentation. Still, this is an ideal stepping stone for our work. We were able to mitigate the impact of fragmentation somewhat by reducing the Immix block size, which increases the likelihood of a block being completely free and thus reusable by other parts of the system.

When introducing different third-party garbage collectors to Julia, we focused on the differences between their collectors and the intrinsic semantics. We found that much of Julia's GC-related infrastructure can be shared and reused among different GC implementations. We tried to reuse code as much as possible. For example, we reuse the existing GC-safe points, the stop-the-world mechanism, the GC disabling mechanism, and the weak reference and finalizer implementation from Julia's stock GC, adapting the code to work with MMTk where necessary.

3.2.1 Root Identification and Object Scanning.

Despite the key difference in root semantics described in Section 2.2.3, we are able to use the same list of GC roots in Julia's stock GC for non-moving Immix, since it does not move objects either.

Enumerating references in an object for scanning requires intimate knowledge about the type information of the object. Ideally, a language runtime provides such a function to enumerate references in an object and only calls to the collector for recording its liveness and reporting its references. However, Julia's object scanning code is heavily entangled with its marking mechanism, the generational behavior from the collector and the underlying memory pages for the pool allocator. Since object scanning is performance-critical, refactoring the code to be general while maintaining good performance was challenging. Furthermore, MMTk is implemented in Rust while Julia's runtime is implemented in C/C++. Naive reuse of the runtime code would result in frequently crossing the language boundary, causing overhead in performance-critical code. In the end, we use our own implementation of object scanning. We generate Rust bindings for the Julia runtime types that are needed for object scanning, and reimplemented object scanning code in Rust. Although this results in better performance, it adds considerable maintenance overhead, as the code has to remain synchronized across changes to Julia's type information. A refactoring to expose the object scanning function from the Julia runtime for third-party GCs remains important future work.

3.2.2 Buffers (Disjoint Objects).

An unusual class of objects encountered in Julia are *buffers*.[2] Buffers are allocated with normal allocation functions, and have the typical GC states described previously. However, buffers only have a tag to indicate their buffer type, and do not include any type information for the collector to scan their payload. Thus, when scanning an object that owns a buffer, the collector is required to use the type information from the object that refers to the buffer to scan the buffer payload.

[2] Julia initially used buffers to implement arrays when we started our work, and buffers were pervasive then. Julia is phasing out the use of buffers, and now uses a properly typed object `jl_genericmemory_t` to back up arrays. However, Julia still uses buffers in other parts of the runtime at the time of writing.

We model this as *disjoint objects*. A disjoint object consists of one parent object and one or more buffer objects. Both the parent object and the buffer objects are allocated as separate objects and have their own GC states, although they form a *logical* disjoint object with only the parent object knowing the type information for the logical object. The parent is responsible for identifying all the references in the disjoint object, including the references to the buffer objects and the references inside the buffer objects' payload.

We implemented buffers as disjoint objects with the current MMTk's object scanning API with no issue. The only caveat is that MMTk requires the runtime to report the object size for every object (including buffer objects) but a buffer object has no type information to know its own size [3]. For now, we allocate an extra word to store the buffer size with the buffer object. Since buffers tend to be large, the overhead from an extra word is insignificant.

3.2.3 Offset Slots. There are two common ways for collectors to queue the edges in the object graph during transitive closure and retrieve object references from the queued edges: *node* enqueueing, and *slot* enqueueing. Node enqueueing directly puts object reference values into the work queue, while slot enqueueing stores the addresses (of object fields) that contain object references. The obvious implication is that if an object is enqueued as a node, it cannot be moved by the collector, as the collector does not know where the reference is stored and cannot update the reference when it is forwarded. We use both enqueueing methods for Julia in our MMTk port while preferring slot enqueueing where possible to facilitate the later copying collector work.

In addition, there are cases in Julia where an object X is referenced by an internal reference R. The internal reference is represented as a slot S for the base reference and an offset value Off, and the collector is supposed to load the base reference from the slot and apply the offset to compute the internal reference R, i.e. $X = \text{load}(S), R = X + Off$. In a non-moving collector, enqueueing the base reference would be sufficient to keep the referenced object X alive. However, our ultimate goal is copying collection. We need to supply both the slot to the base reference and the offset to the collector, so that if the referenced object X is moved to X', we can compute the new internal reference R' as $R' = X' + Off$, and use the slot to update the forwarded reference.

MMTk allows a binding to implement its own `Slot` type for customized slot enqueueing, in addition to node enqueueing. Thus, we support both `SimpleSlot` and `OffsetSlot` for Julia. Though `OffsetSlot` seems unnecessary to support a non-moving collector, we bear in mind that MMTk is a suite of different GC algorithms, and we follow MMTk's principles to be general and abstract to facilitate our later work for different collectors (including copying collectors).

3.2.4 Heap Size Limit. Julia does not support user-defined hard heap size limits, such as Java's `Xmx`, for which the heap size is guaranteed to be within the specified range. For the Java case, a garbage collection is triggered when the heap size is reached, and an out-of-memory error would be thrown if the collector fails to reclaim memory, bringing the heap size below the threshold. At the early stages of our work, Julia did not implement hard heap size limits, triggering GCs mainly based on allocation volumes and a few other heuristics. It later switched to a tweaked variant of *MemBalancer* [13, 20]. Recently, Julia also introduced a command line argument for soft heap size limit via `--heap-size-hint` [14] and a callback function named `jl_gc_cb_notify_gc_pressure_t` [12] which gets called when the heap size is 'under pressure' if the user registers their own callback. However, neither option achieves precise heap size control in Julia, which is important in performance evaluation for GCs. Performance evaluation for Julia's stock GC involves evaluating the combination of the GC triggering heuristics, the heap resizing heuristics and the GC algorithm itself.

This has two implications. First, it means that a fair comparison between the stock Julia GC and any other is problematic since there is no way to directly control heap size, which is one half of the fundamental time-space tradeoff that all collectors make. Second, attaining full compatibility between the stock Julia GC and any other third-party GC is significantly complicated since these heuristics are specific to that GC.

In our work, we allow directly setting a heap size with MMTk, bypassing Julia's restrictions on heap size limiting. MMTk's built-in heap size limiting allows either setting a fixed heap size, which is useful for determining the minimally required heap size of workloads and for performance evaluation, and setting a dynamic heap size, which is useful for multi-tenancy systems. Furthermore, we also implemented Julia's GC triggering heuristics as a `Delegate` GC trigger policy for MMTk in our binding side. With the delegated triggers, MMTk behaves similarly to the stock GC in terms of memory usage. This turned out to be very useful for us to run MMTk-based Julia builds as a drop-in replacement in cloud environments which are pre-configured based on Julia's stock GC.

3.3 Sticky Immix

To support third-party generational garbage collectors in Julia, we need to implement a few more abstractions and functions in the GC interface, the most important among these are write barriers.

Certain essential aspects for generational collectors may be encapsulated within the collector implementation — they do not need to appear in the GC interface. For instance, the *collector* needs to distinguish young objects from old objects;

[3] This is not an issue for Julia's GC, as objects are allocated in the segregated fit lists and an upper bound on size can be obtained by checking the freelist's metadata.

it may do so by using address space separation, or metadata such as the *log* bit and the *sticky* bit [7, 16]. However, the *runtime* does not need the ability to identify object generations, and furthermore, to do so would amount to abstraction leakage since the idea of generations is specific to particular GC algorithms. For this reason, we only include the essential write barrier functions in the interface.

3.3.1 Write Barriers. The GC interface we introduce defines a few different write barrier functions that must be implemented by a third-party GC. The barrier functions include the canonical write barrier, such as `jl_gc_wb` for updating a *target* reference in the *source* object. Some write barriers carry more information for the objects and can be optimized as a `noop` for certain collectors, such as `jl_gc_wb_fresh` that guarantees the source object is always freshly allocated, for which the write barrier can often be omitted in generational collectors. When using MMTk, we keep these collector-specific optimizations as `noop` to avoid the call overhead to MMTk, implementing the remaining barriers.

Write barrier implementation is performance critical [27]. We use the pattern of fast-path and slow-path for write barriers, similarly to what we did for allocation. For barriers in the runtime code, we implement the fast-path as inlined functions in C to minimize the call overhead, and only call MMTk with the corresponding slow-path write barrier functions. Note that besides the runtime write barriers, a generational collector also needs to consider write barriers that are inserted during code generation, as seen in Section 3.1.2. Therefore, each GC must also provide its own implementation when generating code that implements the write barrier logic. We emit fast-path code to inline the write barriers in code generation.

Write barrier functions also include array copying barriers (`jl_gc_wb_genericmemory_copy_ptr`), which are used when the runtime copies the generic memory that backs up an array's storage and may include object references. We implement MMTk's interface of `MemorySlice` to represent arrays for Julia and instruct MMTk on how to iterate through Julia arrays, delegating the array copying barrier call directly to MMTk.

3.3.2 Permanent and System Image Objects. Since the write barrier must capture all references into the nursery, references from not only promoted (mature) objects, but also from Julia's permanent objects must be remembered. The semantics of the write barrier used by Sticky Immix requires that when permanent and system image objects are initialized, their log and sticky bits are set as if they were mature objects. These semantics are specific to the Sticky Immix plan, yet the GC interface must remain algorithm-agnostic. We address this by adding `jl_gc_permobj`, `jl_gc_notify_image_alloc`, and `jl_gc_notify_image_load` to the GC interface, which allows the collector to implement the necessary semantics (such as setting these bits), whenever such an object is allocated. Furthermore, the functions allow GC implementations to perform this in bulk (e.g., by bulk setting the log bits in a side table in MMTk).

3.4 Moving Plans

Adding support for moving collectors (such as (moving) Immix and Sticky Immix[4]) in Julia is a significant leap in terms of complexity from the GC implementations described previously. As mentioned, the stock GC is implemented as a non-moving generational mark-sweep collector, which, despite its specific characteristics, is not inherently different from our non-moving Sticky Immix implementation. However, since Julia was not designed to work with moving collectors, we had to make several changes to the runtime to support object movement.

One of the major challenges was to identify missing roots (a.k.a. dominated roots discussed in Section 2.2.3). We use different techniques for finding missing roots: e.g., by conservatively scanning stacks and registers at run time, or by manually investigating the code and looking for references that may be constructed from those roots but are not necessarily passed to the GC. Note that this process is tedious and error-prone, so ideally, we should use the help from a static analyzer that checks for dominated roots, but that has not been implemented yet. Nevertheless, once the set of dominated roots is identified, we still need to decide how to handle each root. In most cases, we use pinning as a quick solution to avoid moving objects that may be referenced by them. However, other strategies may be applied, for instance, by allocating objects in a *non-moving space*.

For statically-known runtime types that are frequently pinned or types that may be referenced by other Julia runtime types that are not traced by the GC, we can simply allocate each object into a non-moving space for efficiency. Note that this is specific to the Julia runtime code and that we cannot do that for user types, nor user code.

Finally, another strategy to enforce that some dominated roots do not move is to use a level of indirection similar to what is implemented by the Java Native Interface (JNI) [21].

3.4.1 Transitive Pinning. As mentioned in Section 2.2.3, Julia has a definition of roots that is not compatible with moving GCs. The Julia runtime only identifies dominant roots, and may skip dominated roots altogether. To move an object, the GC must identify *all* possible direct references to the object, not only a subset of them. Otherwise, the missing referents cannot be updated by the collector, resulting in dangling pointers.

We initially introduced a mechanism, called *transitive pinning*, to address this issue. Transitive pinning pins an object and all objects that are transitively reachable from it. So

[4]Supporting moving for a generational plan such as Sticky Immix requires little extra effort once moving is supported for Immix.

when supplied with a list of all the dominant roots, transitive pinning guarantees that all the dominated roots will be pinned, avoiding dangling references.

We implement transitive pinning at a very low cost by considering the set of references that require transitive pinning as a different set of roots. We then perform an initial trace starting from these roots, keeping all visited objects in place. A second trace starts from the remaining roots and is allowed to move objects as usual. All objects that have already been marked in the first trace will remain in place and will not need to be traced again. This is an efficient way to implement transitive pinning, since each object is still only traced once.

While this solution is correct, in practice the set of objects transitively reachable from the dominant roots includes most of the heap, leaving very few objects unpinned. Nonetheless, it is still a useful mechanism in our implementation for limited cases, as we will discuss in Section 3.4.3.

3.4.2 Root Identification and Conservative Pinning.

As stated previously, for a copying collector, *we must identify all root references.* Identifying all root-referenced objects is not sufficient. An obvious solution would be to thoroughly go through the language implementation and carefully identify all of the dominated roots. However, such massive refactoring that may affect every runtime function on a code base with hundreds of thousands of lines of code was not only beyond our resources, but such a change would be impractical to land in a production language. Instead, we used an approach that only requires refactoring a small proportion of the code base.

For the stack and registers, we implemented *conservative pinning*, a novel technique that finds and pins the referents of ambiguous roots [16].[5] This approach does not suffer from false retention that affects traditional conservative stack scanning (pinning does not keep objects alive), and the traditional limitation that conservatively referenced objects may not be moved [16] is irrelevant since root-referenced objects can't be moved anyway (Section 2.2.3). We added support to conservatively identify all possible references by identifying the base reference from internal ones by using a valid object (VO) bit that is set for the address representing the beginning of an object [24]. By conservatively pinning potential roots in the stack and registers, we guarantee that the references will still remain valid after the GC. The conservative pinning is made relatively expensive by the need to identify internal references. However, we only need conservative pinning for collections that may move objects for defragmentation, which are infrequent, making the overall cost insignificant.

In principle, a compiler could hide a reference from a conservative collector, for example, by spilling partial registers to the stack [8]. However, this turns out not to be a problem in practice, which is why conservative collectors for C and C++ continue to be widely used [16].

In addition to stacks and registers, Julia may also hold dominated roots in global variables. Julia has an annotation `JL_GLOBALLY_ROOTED` for global variables that are roots (those global variables are direct references to heap objects), though Julia's stock GC only treats a dominant subset of them as roots. In our case, we pass all such global roots to the collector. Note that the objects referenced by those global roots are still required not to move, since there is no guarantee that there are no other references to them. To enforce this, we can either pin those objects or use node enqueueing which keeps these objects in place.

Julia may have dominated roots in its JIT generated code that are not reported to the collector. Fortunately, Julia's code generation only emits pointers to heap objects through several functions. We identify those functions, and make sure the objects being referenced in the code will not move.

Finally, we also need to consider references from the native global memory and the native heap. We undertook a simple manual search in the codebase for native runtime types that are stored in the global memory and in the heap, and that may include references to the managed heap. Those references are essentially dominated roots that have been ignored by the stock GC. Since the uses of native global memory and native runtime types appear only in a small part of the codebase, manually analyzing the code for overlooked dominated roots was tractable. Such roots could either be pinned and reported to the collector, or handles could be introduced into the code, allowing the collector the freedom to move the referents. For now, we just pin them. An alternative approach to our manual search would include conservatively scanning all the native global memory and the heap memory, similar to our conservative pinning for stacks, but this might be too expensive, especially if the number of references to the managed heap from the native heap is not that large.

With the above techniques combined, we are able to identify all missing roots in the Julia codebase. Though the referent objects must all be pinned, most of the heap remains unpinned. The results are workload-dependent but varies from 70% to 99%. The main reason for low movable object ratios in some workloads was the heavy use of `malloc`, and the fact that `malloc`'d objects are not managed by the moving collector.

Even though we have found a practical approach to address the missing roots as described previously, we do hope that our efforts can encourage the Julia community to address this issue systematically, especially as we continue working on evaluating the benefits of moving collectors in Julia. We have identified this aspect of the runtime as the biggest obstacle for Julia to adopt moving collectors.

3.4.3 Object Pinning.

Allowing objects to move is a significant change. The fact that not all objects may be moved

[5] Ambiguous roots are values of unknown type that could plausibly be pointers to heap objects.

restricts the GC algorithms that can be used in Julia, since many algorithms, including canonical algorithms such as semi-space and mark-compact, require moving all objects. As mentioned in Section 3.2, using the Immix family of collectors is particularly interesting since it allows us to do opportunistic copying and pin objects that must not be moved.

There are many different reasons why an object may need to be (transitively) pinned. For instance, we may pin all the dominated roots discovered using the strategies shown before, guaranteeing that those objects will not be moved. However, there are also other cases as we describe below.

Hashed objects. Julia may use the object's address as a seed to generate keys in hash tables. A solution to this problem is to implement proper address-based hashing [1], a technique that keeps track of the hash ID of the original object when moving it. However, this requires two bits in the object header, and an extra word in the case where a hashed object is moved. An alternative when pinning is already well supported is to simply pin all objects that are hashed.

3.4.4 Non-moving Allocation.

Some objects of particular types may never be moved. In this case, we can simply allocate such objects into a non-moving space. This is an extension for the allocation semantics we discussed in Section 3.1.1. For example, the objects of `jl_datatype_t` and `jl_typename_t` should never be moved, since they are used as metadata for scanning objects. Another scenario involves references from the native heap, which may be treated using pinning, but if we know that all the references are of a particular type, we can ensure that all objects of that type are allocated in a non-moving space. Doing so has the same semantics as pinning the object indefinitely, however, it does not require pinning each object individually, and it helps reduce fragmentation in the moving space.

3.4.5 Impacts on Julia FFI.

As discussed in Section 2.2.4, Julia's FFI design prioritizes seamless integration with native languages under the assumption of non-moving collection, and some aspects of Julia's current FFI behavior are implementation-defined. We need a solution that generalizes the Julia FFI with explicitly specified semantics to work with various collectors (including moving collectors) while minimizing breaking changes and semantics to Julia user code. We are still working with the Julia community on a plausible FFI extension that will be acceptable to the community. This section is a mix of what we have implemented in our prototype and what we proposed and discussed with the community.

Pointer Validity The current FFI focuses on the liveness of referenced objects, and uses words such as "extending the lifetime", or "making sure the object exists" in the documentation. It implies that as long as an object is alive, the pointer to it remains valid. This is fine for non-moving collectors but is *not* a strong enough guarantee when such objects may move. On top of object liveness, we consider *pointer validity*, which refers to the validity of references and pointers for live objects. In the documentation where lifetime is discussed, we extend the contract to cover pointer validity.

Unsafe Pointer Conversions Julia applications may make unsafe conversions of pointers using functions like `Base.pointer`, `Base.pointer_from_objref`, or `Base.unsafe_convert` to obtain references to objects in the heap as raw pointers. As the description from those functions states, "[users] must ensure that the object remains referenced for the whole time that the pointer will be used", it is clear that it is the user's responsibility to keep the object alive. However, in the context of moving collection, the functions must also guarantee pointer validity. Since the object is the argument for those functions, the liveness of the object is implied at the function call, and users may not do anything particular to make sure the object is alive before the call. A moving collection may happen immediately after the unsafe pointer conversion call and invalidate the resulting pointer. To be minimally disruptive to user code, we require the unsafe pointer conversion functions to guarantee pointer validity by pinning the objects whose pointers are exposed in a moving collector build.[6]

GC.@preserve and ccall Julia applications may root objects explicitly by calling the `GC.@preserve` macro. As stated in the documentation, this macro "is used to protect dynamically allocated objects from garbage collection". `@preserve x` can be used when for example, passing `x` to a `ccall` to run native code, accessing the memory pointed by `x` directly in Julia itself via a `Ptr` (raw pointer), or when using resources of `x` which would be cleaned up in the finalizer. In all of these cases, it is also possible and common that any object reachable from `x` may be accessed. For that reason, `GC.@preserve` needs to guarantee not only object liveness, but also *transitive* pointer validity. We use the mechanism in Section 3.4.1 to transitively pin the affected object and all the reachable objects. Julia also states that the arguments of `ccall` are guaranteed to be GC preserved, thus transitive pointer validity is guaranteed for `ccall` arguments.

GC Guarantees on Dynamic Scope `GC.@preserve` begins a code block, and the GC guarantees are only effective for the block scope. Users sometimes need to dynamically extend this guarantee for objects for an arbitrary scope. For example, a user may store the pointer of a heap object in native data structures, then return from the native call, which ends the `GC.@preserve` scope.

[6] Pinning is a noop for a non-moving collector build.

The current Julia documentation invites "[users] to make a global variable of type `Array{Ref,1}` to hold these values until the C library notifies you that it is finished with them." Unfortunately, this is insufficient for moving garbage collectors. In the example above, the saved pointer is an unidentified root to the collector, and is reached by the collector as a normal object during a transitive closure. The collector does not have any information to know that the pointer needs to be kept valid and the object cannot be moved. We propose to add a pair of functions, `GC.retain` and `GC.release`. `GC.retain` guarantees object liveness and transitive pointer validity until `GC.release` is called to release those guarantees. Moving collectors can use those objects as roots for transitive pinning. This is a breaking change for the user code if they intend to run with a moving collector. However, we argue that such functions will ultimately be necessarily.

GC-Safe Regions Users may invoke `@ccall gc_safe=true`, or rely on functions such as `jl_gc_safe_enter` and `jl_gc_safe_exit` to allow some native code to run concurrently with garbage collection. For GC-safe regions, the runtime does not guarantee anything with respect to object liveness. It is the users' responsibility to make sure the pointers and the objects they access are valid and additionally make sure that code complies with Julia's definition of 'GC-safe'. With a moving collector, GC-safe regions are risky. The semantics of Julia's GC-safe regions and GC-safe `ccalls` are not well defined in terms of moving collectors, thus it is hard to predict what would work and what would not. There are a few other options that we may consider, such as postponing a moving collection if any thread is running in a GC-safe region.

Our current approach minimizes the introduction of breaking changes, and preserves the efficient and seamless FFI design of Julia as much as possible. However, it heavily relies on transitive pointer validity for FFI. Applications that use FFI frequently and expose key objects that may reach most objects in the heap, may end up with most objects transitively pinned. However, we haven't seen such cases among the workloads we have evaluated, and we will be interested evaluate more sophisticated workloads.

3.5 Summary

We presented our experience of refactoring Julia to expose a well-defined GC interface. Our implementation leverages MMTk along with a selected subset of its GC algorithms to demonstrate the viability of this interface. The choice of GCs reflects the current constraints of the Julia runtime. For instance, Julia's FFI semantics require efficient support for object pinning, which excludes most copying collectors. Collectors like Immix (which supports opportunistic copying) and non-moving collectors like mark-sweep are well-suited for integration with Julia today. Additionally, because the Julia runtime currently supports only write barriers, GC algorithms requiring read barriers remain unexplored.

Though we used MMTk in our work, the interface should straightforwardly support any GC implementation that satisfies Julia's constraints. The GC interface lays the groundwork and opens up more opportunities for broader GC experimentation in Julia, enabling future research into performance, memory utilization, and latency across Julia's characteristic workloads.

4 Related Work

The challenges we discussed throughout the paper are particular to Julia, and follow from design decisions made as the language and its runtime have evolved. Throughout the project, we added relevant references to the Julia documentation, community posts, and the JuliaLang GitHub repository to provide further context.

The use of shadow stacks [18] and conservative stack scanning [9] are the primary approaches that have been used in the past to implement languages when there is no easy way to precisely identify pointers on the runtime stacks. In our case, we leverage conservative stack scanning to perform pinning, since Julia's shadow stack may not contain all the stack dominant roots.

More work has been done in the context of adding GC features to uncooperative languages. For example, to improve Ruby's collector, *RGenGC* [23] devised a way of incrementally introducing write barriers, conservatively scanning those objects that remained unprotected by write barriers. In our case, we try to support legacy code as much as possible, but for Julia's FFI for instance, we might need to introduce some changes to support moving objects that are passed to C code.

5 Conclusion

Julia is an important modern programming language. The original Julia implementation was built with a well tuned non-moving mark-sweep garbage collector. While this collector has served the language well, like all non-moving collectors, it is fundamentally and unavoidably exposed to fragmentation and reduced locality. This work explores a multi-year collaboration to extend Julia with a well defined garbage collection interface, giving the runtime the opportunity to choose among collectors. Among the challenges faced by this work, the largest was the provision for moving garbage collection. We have successfully implemented strategies to manage the various challenges faced by moving collection and have successfully implemented a binding to the MMTk framework.

We hope that this work will serve the memory management community, the Julia community, and the wider programming language design and implementation community with useful insights and lessons.

References

[1] Bowen Alpern, C. Richard Attanasio, John J. Barton, Michael G. Burke, Perry Cheng, Jong-Deok Choi, Anthony Cocchi, Stephen J. Fink, David Grove, Michael Hind, Susan Flynn Hummel, Derek Lieber, Vassily Litvinov, Mark F. Mergen, Ton Ngo, James R. Russell, Vivek Sarkar, Mauricio J. Serrano, Janice C. Shepherd, Stephen E. Smith, Vugranam C. Sreedhar, Harini Srinivasan, and John Whaley. 2000. The Jalapeño virtual machine. *IBM Syst. J.* 39, 1 (2000), 211–238. doi:10.1147/SJ.391.0211

[2] Jeff Bezanson. 2015. *Abstraction in technical computing*. Ph. D. Dissertation. Massachusetts Institute of Technology, Cambridge, MA, USA. https://hdl.handle.net/1721.1/99811

[3] Jeff Bezanson, Jiahao Chen, Benjamin Chung, Stefan Karpinski, Viral B. Shah, Jan Vitek, and Lionel Zoubritzky. 2018. Julia: dynamism and performance reconciled by design. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 120:1–120:23. doi:10.1145/3276490

[4] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. 2017. Julia: A Fresh Approach to Numerical Computing. *SIAM Rev.* 59, 1 (2017), 65–98. doi:10.1137/141000671

[5] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. 2004. Myths and realities: the performance impact of garbage collection. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems, SIGMETRICS 2004, June 10-14, 2004, New York, NY, USA*, Edward G. Coffman Jr., Zhen Liu, and Arif Merchant (Eds.). ACM, 25–36. doi:10.1145/1005686.1005693

[6] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. 2004. Oil and Water? High Performance Garbage Collection in Java with MMTk. In *26th International Conference on Software Engineering (ICSE 2004), 23-28 May 2004, Edinburgh, United Kingdom*, Anthony Finkelstein, Jacky Estublier, and David S. Rosenblum (Eds.). IEEE Computer Society, 137–146. doi:10.1109/ICSE.2004.1317436

[7] Stephen M. Blackburn and Kathryn S. McKinley. 2008. Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 22–32. doi:10.1145/1375581.1375586

[8] Hans-Juergen Boehm. 1996. Simple Garbage-Collector-Safety. In *Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation (PLDI), Philadephia, Pennsylvania, USA, May 21-24, 1996*, Charles N. Fischer (Ed.). ACM, 89–98. doi:10.1145/231379.231394

[9] Hans-Juergen Boehm. 1993. Space efficient conservative garbage collection. *SIGPLAN Not.* 28, 6 (June 1993), 197–206. doi:10.1145/173262.155109

[10] Julia contributor: Diogo Netto. 2023. Concurrent page sweeping. https://github.com/JuliaLang/julia/pull/48969. Accessed: 28 Feb 2025.

[11] Julia contributor: Diogo Netto. 2023. Run GC on multiple threads. https://github.com/JuliaLang/julia/pull/48600. Accessed: 28 Feb 2025.

[12] Julia contributor: Gabriel Baraldi. 2023. Add under pressure callback. https://github.com/JuliaLang/julia/commit/15b34a5768f330d581472c461be2d663b794f5fa. Accessed: 27 Feb 2025.

[13] Julia contributor: Jameson Nash. 2023. gc: add some guard rails and refinements to MemBalancer. https://github.com/JuliaLang/julia/pull/52197. Accessed: 27 Feb 2025.

[14] Julia contributor: rssdev10. 2022. added new command line option heap_size_hint for greedy GC. https://github.com/JuliaLang/julia/pull/45369. Accessed: 27 Feb 2025.

[15] Julia contributors. [n. d.]. Garbage collection in Julia. https://docs.julialang.org/en/v1/devdocs/gc/. Accessed: 28 Feb 2025.

[16] Alan Demers, Mark Weiser, Barry Hayes, Hans Boehm, Daniel Bobrow, and Scott Shenker. 1989. Combining generational and conservative garbage collection: framework and implementations. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '90)*. Association for Computing Machinery, New York, NY, USA, 261–269. doi:10.1145/96709.96735

[17] Nicolas Geoffray, Gaël Thomas, Julia Lawall, Gilles Muller, and Bertil Folliot. 2010. VMKit: a substrate for managed runtime environments. In *Proceedings of the 6th International Conference on Virtual Execution Environments, VEE 2010, Pittsburgh, Pennsylvania, USA, March 17-19, 2010*, Marc E. Fiuczynski, Emery D. Berger, and Andrew Warfield (Eds.). ACM, 51–62. doi:10.1145/1735997.1736006

[18] Fergus Henderson. 2002. Accurate garbage collection in an uncooperative environment. In *Proceedings of The Workshop on Memory Systems Performance (MSP 2002), June 16, 2002 and The International Symposium on Memory Management (ISMM 2002), June 20-21, 2002, Berlin, Germany*, Hans-Juergen Boehm and David Detlefs (Eds.). ACM, 256–263. doi:10.1145/512429.512449

[19] Richard Jones, Antony Hosking, and Eliot Moss. 2011. *The Garbage Collection Handbook: The Art of Automatic Memory Management* (1st ed.). Chapman & Hall/CRC.

[20] Marisa Kirisame, Pranav Shenoy, and Pavel Panchekha. 2022. Optimal heap limits for reducing browser memory use. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 160 (Oct. 2022), 21 pages. doi:10.1145/3563323

[21] Sheng Liang. 1999. *Java Native Interface: Programmer's Guide and Reference* (1st ed.). Addison-Wesley Longman Publishing Co., Inc., USA.

[22] John McCarthy. 1960. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Commun. ACM* 3, 4 (1960), 184–195. doi:10.1145/367177.367199

[23] Koichi Sasada. 2019. Gradual write-barrier insertion into a Ruby interpreter. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on Memory Management* (Phoenix, AZ, USA) *(ISMM 2019)*. Association for Computing Machinery, New York, NY, USA, 115–121. doi:10.1145/3315573.3329986

[24] Rifat Shahriyar, Stephen M. Blackburn, and Kathryn S. McKinley. 2014. Fast conservative garbage collection. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, Andrew P. Black and Todd D. Millstein (Eds.). ACM, 121–139. doi:10.1145/2660193.2660198

[25] Aleksey Shipilëv. 2018. Epsilon: A No-Op Garbage Collector (Experimental). https://openjdk.org/jeps/318. Oracle Corporation, JEP 318.

[26] MMTk Team. 2023. AllocationSemantics (MMTk API Documentation). https://docs.mmtk.io/api/mmtk/plan/enum.AllocationSemantics.html. Accessed: 24 Feb 2025.

[27] Xi Yang, Stephen M. Blackburn, Daniel Frampton, and Antony L. Hosking. 2012. Barriers reconsidered, friendlier still!. In *Proceedings of the 11th International Symposium on Memory Management, ISMM 2012, Beijng, China, June 15 - 16, 2012*. ACM.

[28] Wenyu Zhao, Stephen M. Blackburn, and Kathryn S. McKinley. 2022. Low-latency, high-throughput garbage collection. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 76–91. doi:10.1145/3519939.3523440

Received 2025-03-19; accepted 2025-05-03