# Reworking Memory Management in CRuby

## A Practitioner Report

Kunshan Wang
Australian National University
Beijing, China
u5211824@alumni.anu.edu.au

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

Peter Zhu
Shopify
Toronto, Canada
peter.zhu@shopify.com

Matthew Valentine-House
Shopify
Canterbury, United Kingdom
matt.valentinehouse@shopify.com

## Abstract

Ruby is a dynamic programming language that was first released in 1995 and remains heavily used today. Ruby underpins Ruby on Rails, one of the most widely deployed web application frameworks. The scale at which Rails is deployed has placed increasing pressure on the underlying CRuby implementation, and in particular its approach to memory management. CRuby implements a mark-sweep garbage collector which until recently was non-moving and only allocated fixed-size 40-byte objects, falling back to `malloc` to manage all larger objects. This paper reports on a multi-year academic-industrial collaboration to rework CRuby's approach to memory management with the goal of introducing modularity and the ability to incorporate modern high performance garbage collection algorithms. This required identifying and addressing deeply ingrained assumptions across many aspects of the CRuby runtime. We describe the longstanding CRuby implementation and enumerate core challenges we faced and lessons they offer.

Our work has been embraced by the Ruby community, and the refactorings and new garbage collection interface we describe have been upstreamed. We look forward to this work being used to deploy a new class of garbage collectors for Ruby. We hope that this paper will provide important lessons and insights for Ruby developers, garbage collection researchers and language designers.

***CCS Concepts:*** • **Software and its engineering → Garbage collection**; **Software design tradeoffs**; **Runtime environments**; *Interpreters*; *Scripting languages.*



*ISMM '25, Seoul, Republic of Korea*

ACM ISBN 979-8-4007-1610-2/25/06
https://doi.org/10.1145/3735950.3735960

***Keywords:*** Garbage Collection, Memory Management, Ruby, MMTk

**ACM Reference Format:**
Kunshan Wang, Stephen M. Blackburn, Peter Zhu, and Matthew Valentine-House. 2025. Reworking Memory Management in CRuby: 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, 13 pages. https://doi.org/10.1145/3735950.3735960

## 1 Introduction

Yukihiro (Matz) Matsumoto first started work on the Ruby language in 1993 before publicly releasing it in 1995. It is a dynamically typed, garbage collected scripting language influenced by Lisp and Smalltalk among other languages. At a time where computational efficiency was an increasing concern, Matsumoto took a different path, creating a language "focussed on developer experience, happiness, and productivity" [13]. There are multiple implementations of the language, but the primary one is the one that continues to be led by Matsumoto, often referred to as CRuby [10].

In 2004, David Heinemeier Hansson released Ruby on Rails (a.k.a. 'Rails'), a server-side web application framework built on Ruby. This framework rapidly gained popularity and today is used by many large web sites including GitHub, Airbnb, Twitch, and Shopify. Thus the language which Matsumoto initially designed with a focus on "developer experience and happiness" now plays a critical role in the engine room of today's online economy. This need to run Rails efficiently at scale has driven major efforts targeting Ruby's performance, including the development of the YJIT just-in-time compiler [6, 7] and it is the motivation for the project we describe here.

Ruby's original garbage collector was unusual in a number of ways. As an example, for simplicity, all Ruby objects were managed in fixed sized, 40-byte slots called `RVALUE`s. When an object didn't fit within an `RVALUE`, additional space would be allocated via `malloc` with an indirection left in the

RVALUE. While this choice greatly simplified the garbage collector implementation, it came with notable performance implications including internal fragmentation, poor locality, and runtime overheads associated with calls to malloc and free. The collector was also non-moving (partly because it could afford to be since external fragmentation is much less a concern with uniformly sized slots). As with many design decisions, these choices were embraced and then increasingly depended upon within the runtime and the developer community. For example, since objects did not move, it was reasonable to allow native code to directly access the Ruby heap, even when the collector was running. As long as the native code did not modify pointers while the collector was tracing, the addition of this liberty was not an imposition on the Ruby garbage collector.

The overarching goal of this project is to improve the performance of CRuby through better garbage collection. However, we chose neither to make incremental improvements to the existing collector nor replace it. Instead, we adopted the more ambitious goal of refactoring CRuby to support pluggable garbage collectors, with the existing collector co-existing alongside a selection of alternatives. This objective requires both factoring out a coherent interface, and factoring out decades of assumptions that pervade the runtime and its relationship to memory management.

We have worked closely with the Ruby core team and were able to successfully integrate our major refactoring into Ruby 3.4 at the end of 2024. We have used the MMTk garbage collection framework [3, 9] as a target and are able to choose among multiple MMTk collectors. Specifically, we target the new Rust-based version of the MMTk framework [9] which is designed to be host-language agnostic. This work is ongoing, as we seek to broaden the capabilities of this refactoring to eventually support the integration of high performance collectors such as LXR [25] into Ruby.

Such a major refactoring to a complex, decades-old code base is time consuming, disruptive, and expensive. We believe that its success marks a significant step forward for the Ruby codebase and provides a clear pathway to improved memory management performance in Ruby. We hope that this paper will be helpful to garbage collection experts, language implementors, and Ruby developers alike.

## 2 Background

We now provide a very brief overview of the CRuby runtime and MMTk.

### 2.1 CRuby

CRuby executes Ruby programs mainly with its interpreter. Since the 2.6 release it introduced several experimental JIT compilers. We omit discussion of JIT compilers in this paper, although we plan to optimize for them in the future.

As the name suggests, the CRuby runtime is mainly implemented in C, and third-party Gems (packages) may have components written in C, too. Object layouts can be defined by C structures, and methods can be implemented using C functions. Unlike the Java Native Interface (JNI), which exposes opaque handles and API functions to C programs, CRuby methods implemented in C can hold direct pointers to objects in the GC heap using VALUE, the tagged union type of references and non-reference values. This design choice necessitates the use of conservative stack scanning and object pinning during GC. It also poses the danger that undisciplined third-party Gems can directly access the GC heap in a native thread without holding the global interpreter lock (see Section 4.1), racing with the GC and corrupting the heap.

CRuby started with a simple mark-sweep collector where all objects are 40 bytes in size. Any object that needed more than 40 bytes of space had to allocate extra memory with malloc. CRuby versions 2.1.0 and 2.7.0 introduced generational and copying GC, respectively. Version 3.2.0 introduced *Variable-width Allocation (VWA)* which mitigated the object size limit by introducing a number of fixed sizes [26]. However, despite the continued development effort, some primitive types in CRuby are still not aware of generational or copying GC, and the object size limitation still negatively influences parts of the VM design. Each of these aspects of the CRuby runtime posed significant challenges for our project.

### 2.2 MMTk

The focus of this paper is refactoring the CRuby runtime to support third-party garbage collectors. For concreteness, we use MMTk as our target. However, our refactoring is not specific to MMTk.

MMTk [3] was originally part of JikesRVM [1], an experimental metacircular Java Virtual Machine. MMTk was designed as a highly modular, VM-neutral framework for rapidly building high performance garbage collectors. More recently, MMTk was reimplemented in the Rust programming language [9]. MMTk currently includes multiple garbage collection algorithms, including canonical collectors such as NoGC, MarkSweep, MarkCompact, Immix and SemiSpace, as well as more performant collectors such as GenImmix and StickyImmix.

MMTk implements a *strongly* language-neutral core, and enforces that neutrality through well-defined compiler-enforced interfaces. To integrate MMTk into a VM, the developer needs to implement a two-way *binding* between the MMTk core and the VM. One half of the binding is a logical extension of MMTk, specialized to the target runtime, efficiently performing language-specific operations such as scanning stacks and objects. The other half of the binding is a logical extension of the target runtime, specialized to MMTk, efficiently performing collector-specific operations such as allocation and write barriers. Our system thus has four major components: the CRuby runtime, MMTk, and the two halves of the binding. This paper describes the issues we

encountered as we generalized and factored out GC-related code and assumptions *within the CRuby runtime*.

## 3 Objective

The objective of this work is to refactor and modularize CRuby's memory manager so as to make it possible to use *third party* memory managers alongside CRuby's original memory manager. This objective embodies a number of (implicit) requirements:

- R1 We want to have this work integrated into the mainline CRuby codebase, which implies that we must follow the norms of the existing codebase, ensure our changes are minimally invasive, ensure our changes respect Ruby semantics, and ensure that Ruby's performance and functionality is in no way degraded by our changes.
- R2 We need to retain support for CRuby's existing garbage collector without any performance regression.
- R3 We want to allow the integration of various third-party garbage collectors, rather than one specific garbage collector.
- R4 We want to support copying garbage collectors, which means that we have to identify and respond to *all* assumptions of a non-moving collector, throughout the code base.

Engineering-wise, we approached our goal in small steps:

1. We started with *NoGC*, a trivial 'collector' that allocates but does not collect, similar to OpenJDK's Epsilon [22]. NoGC routes all heap allocations to the third-party GC, but never collects garbage. This helped identify the allocation interface of the VM.
2. We then added support for *non-copying GC*. We extracted the interface for synchronizing GC threads with mutators, and identifying roots and object fields.
3. We then added support for *copying GC*. We ensured that reference forwarding and object pinning worked in this step.
4. We finally added support for *generational GC*. We supported write barriers, and also addressed the unique challenge of *write-barrier-unprotected (WB-unprotected) objects* in this step.

Concretely, we targeted MMTk's Immix and StickyImmix [4] as examples of copying and generational algorithms, respectively. Both support object pinning, which is required by CRuby due to conservative stack scanning and the presence of fields that cannot be updated during copying GC.

## 4 Challenges and Lessons

We now outline major challenges that we faced as we worked toward our goal of modularizing CRuby's memory manager with the objective of third party integration.

### 4.1 Yielding for Collection

Most garbage collection algorithms require the program (mutator) to come to a complete stop, even if briefly, so that the collector can coherently establish runtime roots. We needed to expose this functionality to third party collectors.

**Challenge 1.** *Stopping all threads of execution at a coherent point in order to perform a collection.*

This task was one of the easier elements of the refactoring. CRuby has traditionally relied on a global interpreter lock (GIL), which simplifies many aspects of the runtime implementation by providing a single global synchronization mechanism. This makes it straightforward to ensure mutator-collector synchronization. By contrast, runtimes without a GIL often implement quite sophisticated mechanisms to ensure all mutator threads yield to the collector without inducing significant overhead [17].

In 2020, CRuby 3.0 introduced Ractors, an actor-model abstraction that introduces true parallelism, mitigating the limitations of the GIL. Ractors run concurrently to one another, but threads within a given Ractor do not exhibit true concurrency. Ruby's still uses a single GIL that pauses all Ractors. We were able to expose this mechanism to our third party heap API without difficulty.

**Lesson 1.** *CRuby's relatively simple concurrency model and use of the GIL makes it straightforward to yield all mutators for a collection.*

### 4.2 Conservative Stack Scanning

Stacks are part of the runtime execution state that may reference the heap, so collectors must consider them as roots. In CRuby, there are two kinds of stacks: i) Ruby stacks which hold Ruby local variables, and ii) native stacks which support both the Ruby runtime and third-party extension modules.

**Challenge 2.** *The collector must identify and treat as roots all references from stacks, including native stacks.*

While Ruby stacks are easy to scan, native stacks are more tricky because local variables in C programs may hold direct pointers to heap objects, and C compilers do not generate stack maps to identify the offsets at which local variables are held on the stack.

In the GC literature, words on the stack that may reference heap objects are known as *ambiguous references*. Ambiguous references have two important properties: i) since they may be references, each ambiguously referenced object must be conservatively kept alive, and ii) since they may be values rather than references, they must not be changed, which implies that the referent may not be moved.

We found that it was not difficult to reuse CRuby's default GC's existing conservative stack scanning, which made this challenge relatively easy to address. CRuby is able to iterate through all words in stacks and saved registers and use a filter to identify ambiguous references. This leaves two key

functions for the GC to implement, namely: i) a test that determines whether a word references a valid object within the GC heap, and ii) a mechanism to pin ambiguously referenced objects and report them as roots. Shahriyar et al. [21] efficiently implemented conservative garbage collection in the older Java version of MMTk, and we are able to follow their approach in addressing each of these functions.

To determine whether a reference was valid, Shahriyar et al. use an *object map* that encodes the location of all valid heap objects in a bitmap. Rust MMTk had experimental support for this feature via a *valid-object bit* (VO-bit), so we hardened this into a fully supported feature. When the feature is enabled, MMTk sets the relevant VO-bit each time an object is allocated, and clears it when it considers the object dead. Since a word on the stack can be a reference to a heap object *if and only if* the VO-bit is set at that address, we can use the VO-bit to filter candidate stack words.

To pin ambiguously referenced objects, Shahriyar et al. set a pin bit in the object, which is respected during the collection. They remember all pinned objects in a buffer, which allows the pin bits to be cleared before the next collection cycle. We abstract over the details of the pinning mechanism and introduce a new class of roots to the GC interface which we call *pinning roots*. Objects referenced from pinning roots are pinned for the duration of the collection but otherwise are treated like other root-referenced objects.

Although pinning semantics can't be supported in strictly copying collectors such as semi-space, they are trivially implemented in non-moving collectors and are a feature of opportunistic copying collectors, such as Immix [4] and LXR [25]. We implement support for pinning roots by exploiting the fact that Immix and other opportunistically copying collectors use three states to reflect object visitation during a trace: i) *unmarked*: the object is not yet visited, ii) *forwarded*, the object was visited and moved, iii) *marked*, the object was visited and *not* moved. Objects referenced by pinning roots are simply *marked*, which has the effect of ensuring that the object is not moved. Pinning roots must be processed before other roots to ensure that the referenced object has not already been forwarded when visited during the processing of pinning roots.

**Lesson 2.** *CRuby's existing support for stack scanning was easy to reuse. However, native stacks require conservatism which means collectors must provide mechanisms for identifying valid heap objects and pinning ambiguously referenced objects. The latter limits the choice of collectors to those that can support pinning.*

### 4.3 Object Scanning

One of the most fundamental mechanisms for any garbage collection implementation is the discovery of pointers within each object. This is necessary for tracing, in order to perform a transitive closure over the heap graph, and for reference counting, in order to perform recursive decrements. Because it may be executed millions of times in a single garbage collection, it is also one of the most performance-critical mechanisms in a garbage collector. The mechanism requires intimate coordination between the collector and the runtime, since the location of the pointers within any object is a function of the language implementation.

**Challenge 3.** *The collector must be able to efficiently identify each of the pointer fields within each heap object.*

Classic implementations of object scanning include: i) using a compiler-generated pointer map for each type, ii) using pointer tagging to differentiate pointer fields from non–pointer fields, and iii) performing a conservative scan of the heap object, akin to conservative stack scanning.

CRuby took another approach. Each Ruby type must implement its own marking code. For built-in types, this is `gc_mark_children()`, while for third-party types, the developer implements `rb_data_type_t`, which has a `dmark` function pointer which points to the developer's implementation of the marking routine. When compaction was introduced in Ruby 2.7.0, the design was extended to include `gc_update_object_references()` and `dcompact` for built-in and third-party types respectively. The marking function ensures that each of the object's children are marked, while the compact function ensures that each of the object's children are relocated if necessary and that the objects' pointers to its children are updated accordingly.

CRuby provides developers with a lower level API for mark and compact semantics, including `rb_gc_mark()`, which marks and pins, `rb_gc_mark_movable()`, which marks, and `rb_gc_location()`, which relocates. Listing 1 illustrates how a type `Foo` implements marking and compacting functions in terms of these primitives.

While this design gives the implementers of third-party types freedom to implement type-specific object scanning code, even for non-standard types, it exposes the specifics of CRuby's garbage collector design into every type implementation, including each third-party type. For the set of collectors we initially target, we were able to re-direct `rb_gc_mark()`, `rb_gc_mark_movable()`, and `rb_gc_location()` to the underlying GC implementations in such a way as to correctly achieve object scanning. Although this is workable, our goal is to provide an abstract GC interface capable of supporting multiple GC algorithms, so this approach presents a problem.

We have begun addressing this by introducing *declarative marking* to CRuby. Instead of providing a marking function and a compacting function, a type can provide a list of offsets of reference fields (of `VALUE` type). This list can be conveniently constructed using a macro, as shown in Listing 2. At the time of writing, CRuby only uses declarative marking for the `enumerator` type. This approach solves the problem of embedding the semantics of a single GC in the type declarations, but it does not address the fact that the CRuby marking and compacting code could (and sometimes does)

```c
struct Foo {
    struct RBasic basic;
    VALUE field1; // non-pinning
    VALUE field2; // pinning
    bool status;
    VALUE field3; // conditional
};

void
foo_mark(struct Foo *obj) {
    rb_gc_mark_movable(obj->field1);
    rb_gc_mark(obj->field2);
    if (obj->status) {
        rb_gc_mark_movable(obj->field3);
    }
}

void
foo_compact(struct Foo *obj) {
    obj->field1 = rb_gc_location(obj->field1);
    if (obj->status) {
        obj->field3 = rb_gc_location(obj->field3);
    }
}
```

**Listing 1.** Example of CRuby's field visitors for marking and compacting. Fields of type `VALUE` may contain references to objects. `rb_gc_mark` marks and pins an object. `rb_gc_mark_movable` marks object without pinning, allowing the GC to move the object during compaction. `rb_gc_location` returns the new location of an object if it is moved by the GC. `field1` and `field3` are examples of fields that don't pin their referents, while `field2` pins its referent. `field3` only contains object reference if the `status` field is true.

```c
struct enumerator {
    VALUE obj;
    ID    meth;
    VALUE args;
    VALUE fib;
    VALUE dst;
    VALUE lookahead;
    VALUE feedvalue;
    VALUE stop_exc;
    VALUE size;
    VALUE procs;
    rb_enumerator_size_func *size_fn;
    int kw_splat;
};

RUBY_REFERENCES(enumerator_refs) = {
    RUBY_REF_EDGE(struct enumerator, obj),
    RUBY_REF_EDGE(struct enumerator, args),
    RUBY_REF_EDGE(struct enumerator, fib),
    RUBY_REF_EDGE(struct enumerator, dst),
    RUBY_REF_EDGE(struct enumerator, lookahead),
    RUBY_REF_EDGE(struct enumerator, feedvalue),
    RUBY_REF_EDGE(struct enumerator, stop_exc),
    RUBY_REF_EDGE(struct enumerator, size),
    RUBY_REF_EDGE(struct enumerator, procs),
    RUBY_REF_END
};
```

**Listing 2.** An example of declarative marking. This snippet is taken from the source code of CRuby. Instead of visiting fields using C functions, we simply list reference fields using the `RUBY_REF_EDGE` macro. GC will use this list to find reference fields, mark the children, and update the fields if the referents are moved.

implement type-specific semantics such as a decision to pin a child field, as in Listing 1. Declarative marking happily co-exists with CRuby's old approach so it can be incrementally introduced, allowing on-going support for legacy code. This change makes CRuby more maintainable and allows it to integrate other GC implementations with less difficulty. Fully reflecting such semantics in CRuby in a GC-agnostic way remains future work.

**Lesson 3.** *CRuby's reliance on hand-written per-type mark and compact functions has the side-effect of exposing algorithm-specific GC semantics to developers through type declarations, which makes a change of GC algorithm difficult. Lifting this abstraction is all the more challenging since third-party types encapsulate these semantics, so any change affects all third-party types.*

### 4.4 Reference Enqueuing

A side effect of CRuby's reliance on type-provided marking code is dependence on its field visitors, `rb_gc_mark()`, and `rb_gc_mark_movable()` (Section 4.3 and Listing 1). These visitors each take the *address of the referent* (child) as their argument (e.g. Section 4.3 of Listing 1). This is adequate for a simple mark, however if the collector were to move the referent, it would need to have the *address of the referring field* so that it could update the field with the referent's new location.

CRuby deals with movement by first calling the function `rb_gc_mark_movable()` and then, later in the collection, calling `rb_gc_location()`, which returns the new location of an object, if it was moved. On the other hand, some GCs expect to be passed the address of the referring fields (slots), which allows the collector to directly update the field if and when it moves the referent (see the taxonomy in Figure 5.1 of Atkinson [2]). In fact, by default, MMTk implements what Atkinson's taxonomy refers to as Edge-Slot enqueuing, which means that there is a queue entry for every edge in the object graph, and that edge is represented by a pointer to the field that holds the edge [12]. The fact that CRuby's field visitors don't expose the addresses of the fields to the collector therefore creates a problem.

**Challenge 4.** *Reconcile the unavailability of field addresses in CRuby with underlying collectors' needs for field addresses.*

We added support for Node-ObjRef enqueuing [2] to the MMTk core to complement its existing Edge-Slot enqueuing. The new API allows the VM binding to choose, at the time of scanning an object, whether to enqueue the addresses of its fields, or to directly trace the children referenced by its fields. This allows us to route MMTk's object-scanning callback to the marking functions of CRuby's types (such as `foo_mark`), and route CRuby's field visitors (such as `rb_gc_mark`) back to MMTk's object-tracing function.

The addition of Node-ObjRef allows the object graph to be traversed, but does not directly solve the problem of how fields are updated in the case when an object is moved. In our current implementation, each time an object is scanned, its fields are first marked (e.g. `rb_gc_mark`), and are then immediately updated (`rb_gc_location()`), just as CRuby does.

**Lesson 4.** *CRuby's approach to scanning objects reflects its history and the fact that originally it only marked objects. Generalizing this approach will be important in the future. For now we have generalized MMTk to work with such limitations.*

### 4.5 Finalization and Off-Heap Memory

Finalizers are operations performed when an object is determined to be unreachable by the garbage collector. In CRuby, there are two forms of finalizer, namely: i) the `obj_free` function, and ii) operations registered by the `define_finalizer` function of `ObjectSpace`. We will focus on `obj_free` here, and leave `define_finalizer` to Section 4.7.

Whenever an object in CRuby dies, the `obj_free` function is called to allow clean up of underlying resources associated with the object, such as file descriptors. A crucial use of `obj_free` is freeing off-heap memory allocated by `malloc`.

As mentioned previously, historically the CRuby heap only supported 40-byte slots [27]. When an object larger than 40 bytes was created, CRuby used `malloc` to allocate an off-heap buffer in order to accommodate the object, leaving a pointer to the off-heap buffer within the 40-byte heap object. When such an object dies, CRuby uses the `obj_free` function to free the off-heap buffer. Variable-width allocation introduced in CRuby 3.2.0 [26] allows objects up to 640 bytes, mitigating the problem but not solving it.

While finalization is a feature of many garbage collected languages, the use of a finalizer is typically the exception rather than the rule. However, because most objects in Ruby are instances of types that *may* have an off-heap buffer, nearly every Ruby object is subject to this form of finalization, which is expensive. While CRuby's collector must visit every dead object, most high performance collectors exploit the weak generational hypothesis [16, 23], and only visit live objects. This aspect of CRuby's finalization therefore significantly undermines performance objectives of modern collectors.

**Table 1.** The cost in milliseconds of freeing 100,000,000 × 32 B objects as a function of malloc library implementation and the number of threads freeing the objects. Only mimalloc offers any scalability; the others scale *negatively* as the number of threads grows.

| threads | glibc | jemalloc | tcmalloc | mimalloc |
|---|---|---|---|---|
| 1 | 1,263 | 3,935 | 4,988 | 903 |
| 2 | 5,002 | 11,719 | 13,539 | 493 |
| 3 | 5,787 | 17,606 | 11,374 | 346 |
| 4 | 6,790 | 22,478 | 17,295 | 265 |
| 5 | 8,058 | | 17,785 | 291 |
| 6 | 7,473 | | 19,227 | 243 |
| 10 | 9,400 | | 23,350 | 230 |
| 100 | 11,260 | | 24,195 | 228 |

**Challenge 5.** *CRuby's use of finalization on all objects makes implementing performant garbage collectors challenging, since it requires that every dead object be visited. This requirement stands in the way of efficient collector design since most high performance collectors are designed to avoid visiting dead objects, exploiting the weak generational hypothesis.*

Unfortunately, there is a second significant consequence of Ruby's use of off-heap buffers — finalization does not scale due to `free` not scaling. This is illustrated by the microbenchmark results shown in Table 1. Because CRuby's default collector and the finalization mechanism it embodies is single-threaded, this performance problem is also masked in CRuby. However, our goal is to support a range of garbage collectors and to make the most of available hardware, so scalability is a priority.

**Challenge 5b.** *CRuby's extensive need to deallocate off-heap buffers creates a major performance bottleneck for parallel garbage collectors.*

One of our motivations for introducing a new garbage collection interface to CRuby was that it might address limitations of CRuby's memory management such as the need to augment heap objects with off-heap memory. However, Ruby exposes many implementation details of its default GC to the runtime, including object size limitations. Thus runtime code designed for its default garbage collector must, for now at least, continue to allocate and free off heap memory for any object larger than 640 B, even when using a garbage collector that does not have this limitation.

Nonetheless, we have explored how these challenges could be addressed when using a GC that is not constrained this way. We started by conducting a study of the most prolific uses of off-heap buffers, and found that the built-in types `T_STRING`, `T_ARRAY`, and `T_MATCH` are dominant. These support the heavily-used `String`, `Array`, and `MatchData` types respectively. Because these are built-in types we can, in principle

at least, provide new implementations of these types in the CRuby runtime that avoid off heap memory.

We introduced two new types: `imemo:strbuf` and `imemo:objbuf`. In CRuby, IMEMO types are internal types, whose instances reside in the garbage collected heap but cannot be directly referenced from variables in Ruby programs. The types we introduced are analogous to Java's **`byte`**`[]` and `Object[]` types; they are objects with a *capacity* field followed by an array of *capacity* elements (**`char`** and `VALUE`, respectively). We then refactored the `T_STRING` (`String`) and `T_ARRAY` (`Array`) built-in types to use `imemo:strbuf` and `imemo:objbuf` respectively rather than off-heap memory. We also refactored the `T_MATCH` built-in type (`MatchData`) to use our new `imemo:strbuf` and `imemo:objbuf` types.

This refactoring means that instances of `String`, `Array`, and `MatchData` are all entirely heap-allocated, so no longer need finalization. We can extend this optimization to other built-in types that need finalization due to off-heap buffers, such as `T_HASH` (for the `Hash` type) which also has many instances. Ruby `T_OBJECT` instances (for ordinary classes defined in Ruby code) may grow. When they do so, to retain referential integrity of the object, CRuby may need to allocate an off-heap buffer to hold the additional fields. Because this happens frequently, CRuby includes a mechanism named *object shapes* that can predict the eventual size of objects, avoiding the need for off-heap allocation during growth. This predictor works so well that in practice, we found that very few instances of `T_OBJECT` ever require off-heap buffers.

We were able to demonstrate that the problem of finalization due to off-heap memory allocations could be addressed very effectively with limited effort by introducing the `imemo:strbuf` and `imemo:objbuf` types. However, our approach is contingent on the memory manager not being limited to 640 B heap allocations, thus it is currently incompatible with CRuby's default memory manager and therefore has not been upstreamed to CRuby.

**Lesson 5.** *Early choices such as fixed cell sizes can lead to costly practices such as the systematic use of off-heap memory becoming deeply embedded in the runtime and even the language. This leads to a vicious cycle by making it hard for the language to retreat from the original decision, locking a language into severely limiting design choices [15].*

### 4.6 Copying Garbage Collection

The performance of any non-moving garbage collector is ultimately limited by the spatial and temporal effects of fragmentation, which is inevitable. With the release of 2.7.0 at the end of 2019, CRuby augmented its non-moving fixed-size 40-byte mark-sweep collector with compaction, using Edwards' two-finger algorithm [18, 20].

The two-finger algorithm compacts regions of same-sized objects by moving two 'fingers', one from each end of the region. The top finger advances downward to the first occupied cell, while the bottom finger advances upward to the first unoccupied cell. The cells are then 'swapped': the occupied cell is moved to the unoccupied cell, and a forwarding reference is left behind. This repeats until the fingers meet. Once complete, all occupied cells will have been compacted to the bottom of the space. Memory must then be scanned to redirect pointers to objects which were moved, using forwarding pointers. (The compaction algorithm was extended to accommodate multiple fixed sizes when variable-width allocation was introduced in CRuby 3.2.0 [26].)

Recall that historically, CRuby performed marking by having each type implement the `rb_gc_mark` field visitor (Listing 1). As mentioned previously, the addition of compaction led to the need for a `rb_gc_location` field visitor which could be used to redirect pointers when objects were moved. However, because legacy C extensions may reference heap values and may not be aware of the possibility of them moving, CRuby needed a way to ensure that heap objects referenced by C extensions would not be moved.

CRuby solved this by implementing pinning, and *making it the default*, giving `rb_gc_mark` the semantics of pinning the referent. Moving was thus implemented by adding `rb_gc_mark_movable` which has the semantics of allowing the referent to be moved. This way, types which can support movable referents do so explicitly, by implementing the new visitor, while legacy types are by default safe since without the new visitor their references are implicitly pinned.

In addition to legacy C extensions, a second reason for pinning is that certain object fields must be treated conservatively. For example, the field `u3` in `struct MEMO` in the CRuby runtime is an untagged union which sometimes holds a reference, and sometimes holds an un-tagged integer indistinguishable from references. The `u3` field is thus an ambiguous reference, so must be treated conservatively and its (potential) referent cannot be moved.

CRuby integrates this requirement for pinning into Edwards' two-finger compaction algorithm [20] by using three steps: i) it performs a full trace of the heap to find and pin all objects which must not be moved, ii) it uses the two-finger algorithm to compact the space, and iii) it visits each object in the heap to update references to objects that were moved. However, for other garbage collection algorithms, this requirement presents a problem, since in general the collector cannot know whether a child will be pinned until *all* of its parents have been visited.

**Challenge 6.** *In the CRuby heap, a child can be pinned due to the type of any of its parents.*

As we saw above, this challenge is addressed by CRuby's default compacting collector by performing a complete trace over the heap to find pinned children before moving anything. A simple solution to the challenge would therefore be to do the same for every moving collector that might be integrated with CRuby, but this is unattractive due to is high cost.

Instead, we introduced the concept of *potentially pinning parents*, or *PPP* for short. A potential pinning parent may contain reference fields that cannot be updated. Because those fields may contain `nil` or other non-reference values, they do not always have children to pin (i.e. they *potentially* pin their children). We record all PPP objects in a global list so that at the beginning of a GC, the GC can iterate through this list to identify all the children pinned by PPPs.

However, since the PPP iteration occurs *before* the GC, the parent may in fact be unreachable, so marking each child as live without knowing whether the parent was live would be wasteful. We addressed this in our implementation by separating pinning from liveness. Thus the PPP iteration merely pins the referent; it does not mark it. Then during the main trace, when a pinned object is reached it is left in place, but unreached pinned objects are collected as usual. At the end of each GC, live objects' pinning bits must be cleared and dead objects must be removed from the PPP list.

Fortunately, in CRuby, very few types are PPPs. Only some IMEMO types and third-party `T_DATA` types are PPP types at allocation. The `Hash` type only becomes PPP when it starts using object identity as a hash code (which is a consequence of not implementing *address-based hashing* which we will discuss in Section 4.8). We are still working on reducing the number of PPPs, and we should ultimately eliminate all the unnecessary pinning inside the CRuby runtime.

However, we cannot change legacy third-party Gems (packages) built before CRuby 2.7.0. At the time of writing, even some actively maintained Gems, such as `nokogiri` [8] and `json` [11], still do not fully support moving GC. This means that any garbage collector built for CRuby must support fields that cannot be updated. Meanwhile, we expect that third-party Gems that care about GC performance will start to implement the updating functions or start using the *declarative marking* mechanism we discussed earlier in Section 4.3.

**Lesson 6.** *CRuby has fields that cannot be modified by the collector, which complicates copying garbage collection since it means that an object may only be moved once it is known that the object is not referenced by any such field. We work around this by introducing potentially pinning parents (PPP). However, this constraint will continue to be a problem for CRuby if it wishes to support legacy code unaware of moving GC.*

### 4.7 Global Weak Tables

Weak tables reference heap objects without keeping the heap objects alive. CRuby implements a number of weak tables internally. For example, the *generic instance variable table* maps objects to lists of instance variables, and is used to hold instance fields for objects whose types are not derived from `BasicObject`, `Class` or `Module`.[1] The *finalizer table* maps objects to lists of user-defined finalizers registered by the `define_finalizer` method of `ObjectSpace`. CRuby uses the *frozen string table* to implement string interning and the *global symbol table* to canonicalize symbols.

Once an object referenced by a weak table dies, the weak table entry needs to be cleaned up to prevent a dangling reference. However, CRuby's weak tables are not implemented using Java-style weak references, which are automatically cleared by the garbage collector. Instead, CRuby relies on finalizers to clear references from the weak tables. Notice that this implementation implies that since any object might be referenced from one of these weak tables, all objects must be finalized. This implies that the garbage collector must visit every dead object, which works against any effort to exploit the weak generational hypothesis [16, 23] (Lesson 5).

Complicating things further, CRuby implements lazy sweeping to recover memory, which means that finalizers are not invoked promptly. This in turn means that weak tables may contain references to dead objects. CRuby mitigates this problem by having all accesses to these tables perform a check for an invalid entry.

**Challenge 7.** *CRuby's reliance on finalizers to clean up weak table entries rather than cleaning them automatically via the garbage collector means that all objects are subject to finalization, complicating the CRuby runtime.*

We implemented a new method for cleaning up global weak tables that does not rely on all objects being finalized, and that is not performed lazily. Once the collector has finished identifying all live objects, it iterates over each weak table, removing entries that point to dead objects, a task that we trivially parallelize. Since this is done during GC, mutators no longer need to check for dangling weak table references. We successfully upstreamed this change.

At the time of writing, CRuby's default GC uses this new approach during compacting GCs, while ordinary non-moving GCs still rely on finalization to remove entries. Third-party GC modules, such as MMTk, can use this new method without making every object finalizable. We generalized MMTk's API, which previously only supported Java-style weak references, to allow client VMs to implement the cleaning strategy we implemented in CRuby.

We also added a function `rb_gc_mark_weak` [28] to CRuby's default GC so that it now understands weak reference semantics. This allows it to discover weak reference fields in objects and process them after tracing. It was initially designed for internal object fields in the CRuby runtime that should have weak reference semantics, such as the reference from a call cache to its callable method entry [5]. However, at the time of writing, it is only used for the `WeakMap` class in the standard library.

**Lesson 7.** *If the garbage collector does not support weak reference semantics, the VM is likely to resort to using a finalizer to perform cleanup, harming performance and complicating the*

[1] In CRuby, when types not derived from `BasicObject`, `Class` or `Module` are extended, additional fields are *not* simply included as part of the instance as they are in other languages, but rather, are held in a large map within the runtime, with the instance as the key and a list of field values as the value.

*VM implementation. This is another example of how using a simple GC implementation early in the life of a language may have broader impacts on the design and healthy development of a language runtime.*

### 4.8 Hashing on Addresses

In CRuby, some weak tables, such as the generic instance variable table and the finalizer table, use object identities as the key. CRuby naively implemented an identity hash by hashing the object address directly. It was not a problem when CRuby was still using non-moving GC. However, when moving GC happens, addresses of the key objects will change, and such hash tables will have to be re-hashed.

CRuby's default GC addressed this problem on an ad hoc basis. For example, objects with finalizers registered with `ObjectSpace.define_finalizer` are pinned so that the finalizer table does not need rehashing. On the other hand, objects with out-of-object instance variables are handled specially. When the GC moves such an object, it removes the old entry from the generic instance variable table, and re-inserts a new entry with the new address as the key and keeps the old value. In this way, the generic instance variable table only contains valid entries after GC.

**Challenge 8.** *CRuby uses object addresses as hash keys, which requires rehashing when keys are moved.*

When using MMTk, we reconstruct such weak hash tables using forwarded references and replace the old hash table. This approach is no more elegant than the default GC, but it avoids pinning objects. It also avoids updating those thread-unsafe hash tables when moving objects, which can happen in parallel in multiple GC threads.

A standard algorithm for implementing identity hash code in a moving GC is *address-based hashing* [1]. The first time an object's hash is taken, its address is returned as the hash and a bit is set to indicate that the object has been hashed. The first time the garbage collector moves a hashed object, it allocates an extra word in which the hash value (its old address) is stored, and a second bit is set to indicate that the saved hash should be used in the future rather than the object's address. This approach requires two spare bits per object. Ideally, these bits would be located in the object's header, but the current `flags` field does not have any free bits to spare, so they would need to be implemented in side metadata, imposing a 2/64 (3%) space overhead.

CRuby's default GC makes address-based hashing difficult to implement because the addition of the extra word in front of the object effectively changes the object's shape, unlike in other memory managers where the runtime need not be aware of any space between objects or metadata injected in front of them. Nonetheless, we are planning to add address-based hashing into CRuby to eliminate the VM-wide complexity of the identity hash.

**Lesson 8.** *The problem of hashing on object addresses is another example of how deeply held assumptions of non-moving garbage collection have extensive implications for a runtime, making moving beyond such a simple collector very difficult.*

### 4.9 Write Barriers and Generational Garbage Collection

Generational garbage collectors avoid tracing mature objects, but to do so, must conservatively treat all mature objects as live, and *remember* all references from mature objects to young objects.

**Challenge 9.** *Generational collectors must be able to efficiently identify references from mature objects to young objects.*

Fortunately, CRuby already supports generational garbage collection and implements write barriers for *most* field updating operations. In the few cases where write barriers are missing, CRuby identifies the parent objects as *write-barrier-unprotected (WB-unprotected)* objects, which we discuss below.

CRuby implements a variation on the classic object-remembering barrier [24]. The first time a reference field in a mature object is modified, the object is remembered (by setting a bit in a metadata table). During young generation garbage collection, fields of the remembered objects are treated as roots, so all young objects transitively reachable from them are retained. As each mature object is processed, the remembered bit is reset so that the object will be remembered the next time it is modified.

This barrier is sufficiently general to support generational garbage collection, so we were easily able to use it in our MMTk port.[2]

As elaborated in Section 4.5, we replaced some `malloc` buffers with IMEMO objects. For example, instances of `imemo:strbuf` and `imemo:objbuf` are garbage collected objects, referenced from `String`, `Array` and `MatchData`. We therefore need to apply the write barrier when we assign such buffers to their owners. Meanwhile, if the payload of an `Array` is held in a child `imemo:objbuf`, write operations of array elements must apply write barriers to the `imemo:objbuf` object instead of the `Array` itself.

**Lesson 9.** *The fact that CRuby had already inserted most write barriers and the generality of CRuby's object-remembering barrier made integration with other generational collectors fairly straightforward.*

However, it was still necessary to deal with CRuby's WB-unprotected objects mentioned earlier.

[2] In the future, we will need to refactor CRuby's use of array write barriers to support more advanced GC algorithms, such as the concurrent LXR [25] collector. That will affect a small part of the CRuby runtime, mostly involving the implementation of `Array`.

The idea of having objects that were not subject to write barriers began as a pragmatic choice when making the major change of introducing generational collection to CRuby. CRuby 2.1.0 introduced generational GC (RGenGC), but the code base was too large to insert write barriers everywhere at once. To allow write barriers to be introduced gradually, CRuby objects were allowed to be either WB-protected or WB-unprotected. Write operations to fields of WB-unprotected objects may not apply write barriers, while write operations to WB-protected objects must always apply write barriers. The developers started with almost all types being WB-unprotected, and gradually made more and more types WB-protected.

**Challenge 10.** *Identify references from mature objects to young objects without using write barriers.*

We adopt a simple strategy. At the start of each minor collection, we conservatively treat all WB-unprotected objects as if they had been modified during the preceding mutator phase. We do this by simply keeping a list of all mature, live, WB-unprotected objects and treating their fields as roots into the young generation during minor collections. This approach has the advantage of not requiring any change to the collector being integrated.

Our approach is somewhat less complex than the handling of WB-unprotected objects in CRuby's default GC, which involves: i) not promoting WB-unprotected objects into the mature space, ii) adding WB-protected objects into the remembered set only when referenced by an old object, and iii) demoting a mature object if it suddenly becomes WB-unprotected (for example, after its interior pointer is leaked to a C extension). We assume that this strategy emerged because it began when the majority of all objects were WB-unprotected, and keeping all WB-unprotected objects in the remembered set was impractical. Nowadays, the number of WB-unprotected objects is greatly reduced, and the overhead of listing all WB-unprotected objects is no longer significant.

Unfortunately, it seems that CRuby will have some WB-unprotected objects for the foreseeable future. Reference fields can be leaked, unprotected by barriers, to third-party C extensions by some public API functions, such as `RARRAY_PTR`. Although strongly discouraged by the documentation, legacy C extensions still use those legacy APIs, and access reference fields without write barriers. This means CRuby must always expect the presence of WB-unprotected objects if it wants to maintain compatibility with such third-party C extensions.

**Lesson 10.** *It was not hard to support WB-unprotected objects for third-party collectors by listing the fields of mature WB-unprotected objects as roots during minor collections. The fact that CRuby leaks reference fields, unprotected, to third-party C extensions makes it impossible to make all objects WB-protected.*

### 4.10 Performance Analysis

When integrating a third party collector, even if that collector is highly tuned, it may exhibit previously unseen performance behaviors once integrated into a new runtime such as CRuby. This may be for many reasons, including: i) workloads that run on CRuby have distinctly different profiles to the workloads on which the collector was previously optimized, ii) aspects of the CRuby language and runtime implementation lead to different bottlenecks and tradeoffs within the collectors, or iii) because the integration with the CRuby runtime is itself a bottleneck. Performance analysis and debugging is therefore a first order concern when integrating a third party collector.

**Challenge 11.** *Quickly identify garbage collection performance bottlenecks.*

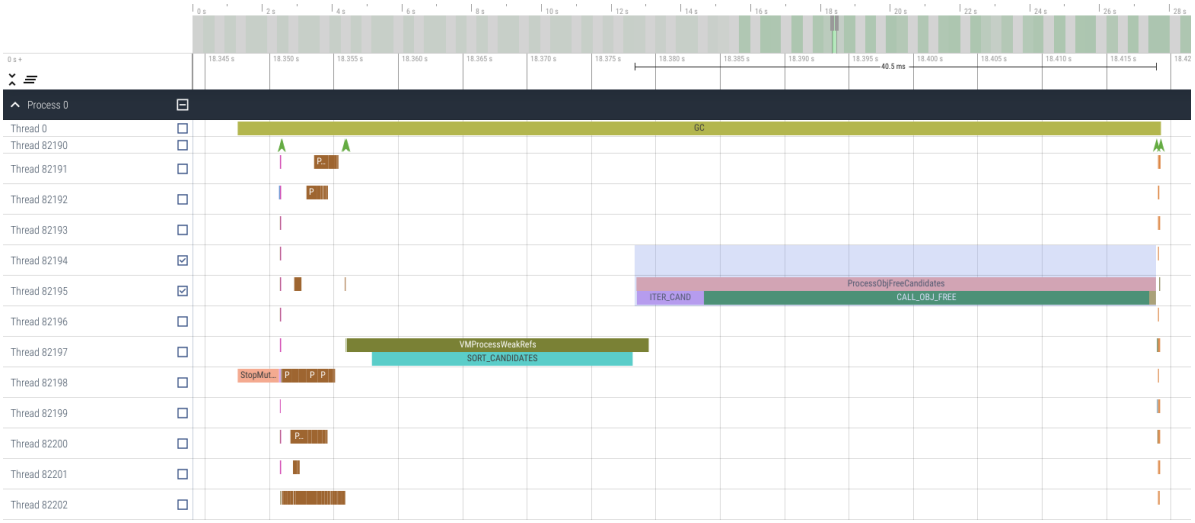
Our initial target, MMTk, offers very good eBPF-based support for high fidelity performance analysis [14], complementing traditional tools such as `perf`. MMTk performs all work, both within its core and in the VM binding in work packets, which execute work items as fine grained as marking an object and as coarse grained as scanning a stack. MMTk uses *User Statically-Defined Tracepoints (USDTs)* which allows the timing of each collection and every work packet to be captured at very low overhead using the `bpftrace` command line tool. Traces generated this way can be directly imported into standard tools such as Perfetto [19], enabling detailed fine-grained analysis. MMTk's framework is trivially extensible, allowing us to capture CRuby-specific information, such as the number of objects finalized, and the number of WB-unprotected objects processed.

For example, Figure 1 clearly illustrates that: i) calls to `obj_free` dominate collection time, and ii) parallelizing the finalization process makes the situation *worse*. Alarmingly, this also told us that our goal of improving the GC algorithm itself would not yield any noticeable performance benefit as long as finalization remained a bottleneck. This, along with confirmation via a microbenchmark made reducing the number of objects that were finalized a very high priority for our project (Section 4.5).

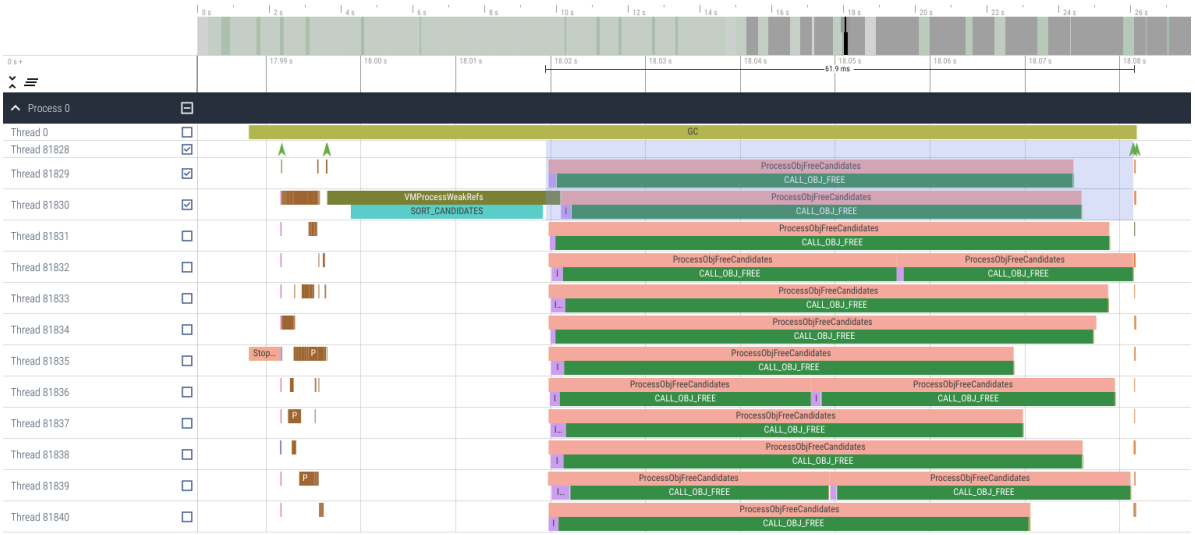
**Lesson 11.** *A good performance analysis and visualization tool is invaluable to the task of identifying bottlenecks and optimization targets.*

### 4.11 Performance Optimization

The overarching goal of this work was to allow CRuby to use different garbage collectors, so that CRuby would have alternatives in the face of limitations specific to its existing collector. One such limitation is CRuby's use of a free-list allocator. Free list allocators suffer a significant performance penalty compared to bump pointer allocators, which are both simpler and offer better locality [4]. However, because allocation is performance-critical, efficiently integrating a new allocator into a runtime is not trivial.




**(a)** Single-threaded processing of `obj_free` calls.




**(b)** Multi-threaded processing of `obj_free` calls.

**Figure 1.** Understanding `obj_free` overheads and scalability using eBPF-based tracing tools, viewed through the Perfetto UI. These are time series plots, with time progressing on the x-axis, and garbage collector threads constituting the rows. Even without understanding the details, both plots make it immediately visually obvious that `obj_free` (right, peach/dark green) is far more costly than the full-heap trace (left, brown), dominating the cost of the collections. A comparison of the upper and lower screenshots shows the alarming *negative* scalability of the `obj_free` operation, increasing from 40.5 ms to 61.9 ms when parallelized (Section 4.5).

**Challenge 12.** *Efficiently implement a new allocator in the CRuby runtime. Allocator implementation is one of the most performance-critical aspects of implementing a new garbage collector.*

Because they are performance-critical, most high performance allocators are implemented using a fast path/slow path pattern. The fast path is designed to as efficiently as possible implement the common case for allocation, while the slow path manages complexity that arises less frequently, such as replenishing a thread-local allocation buffer (TLAB) from which the fast path allocates. For performance, the fast path is typically inlined and contains no synchronization and no calls other than a conditional call to the slow path.

MMTk's API exposes allocation fast and slow paths. This design allows an initial non-performant implementation to simply make a call to MMTk's API implementation of the fast

path, while an advanced implementation can re-implement the fast path in the target runtime and call the API only for the slow path.

CRuby mainly uses an interpreter, and implements most primitive functions such as accesses to strings and arrays in ahead of time compiled C code. It appeared at first glance that object allocation should not have too much overhead compared to the rest of the interpreter, but performance analysis showed overheads of 3-4%, so we implemented the bump pointer allocation fast path in C inside CRuby, calling the API for the slow path.

**Lesson 12.** *Despite being mostly interpreted, the performance of the allocation fast path is still important for CRuby.*

## 5 Future Work

The most important part of our work, namely refactoring the runtime to implement and enforce a modular GC interface, has been upstreamed. Nonetheless, there remains plenty to be done by way of improving CRuby's memory management. Above all, incorporating better support for object movement and greater opportunities for moving garbage collection is essential. There are also many other secondary issues remaining, which we have not touched on in this paper, including: i) there are too few bits available in the object header to conveniently implement forwarding and address-based hashing; ii) CRuby implements *object capacity* by reporting the size of the free list cell containing the object to the language level, thereby exposing implementation details, which is limiting; and iii) CRuby's use of *zombie* objects prevents timely memory reclamation. These remain as opportunities for future improvement.

## 6 Conclusion

We refactored CRuby to allow it to integrate third-party garbage collector implementations. Our multi-year effort addressed many challenges imposed by the CRuby runtime, most of which can be traced back to CRuby's original decades-old GC implementation.

Of the many lessons learned, the overarching one was that early choices such as fixed cell sizes can lead to costly practices such as the systematic use of off-heap memory becoming deeply embedded in the runtime and even the language. This leads to a vicious cycle by making it hard for the language to retreat from the original decision, locking a language into severely limiting design choices [15]. Put differently, the centrality of a garbage collector to a runtime such as CRuby, and its performance-critical role means that unless strong and rich abstractions are implemented and maintained, the initial GC implementation will over time become more and more deeply integrated into the runtime, shackling its performance to design choices that often prove to be anachronistic.

We hope that the lessons outlined here will be useful for Ruby developers and garbage collection researchers alike.

## Acknowledgments

This paper is dedicated to the memory of Chris Seaton, who was instrumental in establishing and co-leading this project from its inception. Our collaboration began in May 2020 when Chris expressed his keen interest and subsequently secured the essential financial support from Shopify, without which this ambitious undertaking would not have been possible. We remember with deep gratitude Chris's technical vision, perseverance, and the friendly, collegial spirit that defined our many productive meetings.

We thank Angus Atkinson who, as an undergraduate at ANU, did the original work of getting MMTk's NoGC to work in CRuby.

## References

[1] Bowen Alpern, C. R. Attanasio, Anthony Cocchi, Derek Lieber, Stephen Smith, Ton Ngo, John J. Barton, Susan Hummel Flynn, Janice C. Sheperd, and Mark Mergen. 1999. Implementing Jalapeño in Java. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*. ACM, 314–324. doi:10.1145/320384.320418

[2] Angus Atkinson. 2023. Software Cache Prefetching for Tracing Garbage Collectors. https://www.steveblackburn.org/pubs/theses/atkinson-2023.pdf

[3] 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

[4] 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 ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, Rajiv Gupta and Saman P. Amarasinghe (Eds.). ACM, 22–32. doi:10.1145/1375581.1375586

[5] Jean Boussier. 2023. Call Cache for singleton methods can lead to "memory leaks". https://bugs.ruby-lang.org/issues/19436

[6] Maxime Chevalier-Boisvert, Noah Gibbs, Jean Boussier, Si Xing (Alan) Wu, Aaron Patterson, Kevin Newton, and John Hawthorn. 2021. YJIT: a basic block versioning JIT compiler for CRuby. In *VMIL 2021: Proceedings of the 13th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages, Virtual Event / Chicago, IL, USA, 19 October 2021*, Gregor Richards and Manuel Rigger (Eds.). ACM, 25–32. doi:10.1145/3486606.3486781

[7] Maxime Chevalier-Boisvert, Takashi Kokubun, Noah Gibbs, Si Xing (Alan) Wu, Aaron Patterson, and Jemma Issroff. 2023. Evaluating YJIT's Performance in a Production Context: A Pragmatic Approach. In *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes, MPLR 2023, Cascais, Portugal, 22 October 2023*, Rodrigo Bruno and Eliot Moss (Eds.). ACM, 20–33. doi:10.1145/3617651.3622982

[8] Mike Dalessio, Aaron Patterson, Yoko Harada, Akinori MUSHA, John Shahid, Karol Bucek, Sam Ruby, Craig Barnes, Stephen Checkoway, Lars Kanis, Sergio Arbeo, Timothy Elliott, and Nobuyoshi Nakada. 2025. Nokogiri. https://github.com/sparklemotion/nokogiri

[9] MMTk Developers. 2025. The MMTk Project. https://mmtk.io

[10] Ruby Developers. 2025. The Ruby Programming Language. https://github.com/ruby/ruby

[11] Florian Frank. 2025. JSON implementation for Ruby. https://github.com/ruby/json

[12] Robin Garner, Stephen M. Blackburn, and Daniel Frampton. 2011. A comprehensive evaluation of object scanning techniques. In *Proceedings of the 10th International Symposium on Memory Management, ISMM 2011, San Jose, CA, USA, June 04 - 05, 2011*, Hans-Juergen Boehm and David F. Bacon (Eds.). ACM, 33–42. doi:10.1145/1993478.1993484

[13] Heroku. 2011. Matz joins Heroku. https://blog.heroku.com/matz_joins_heroku

[14] Claire Huang, Stephen Blackburn, and Zixian Cai. 2023. Improving Garbage Collection Observability with Performance Tracing. In *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR 2023)*. ACM, New York, NY, USA, 85–99. doi:10.1145/3617651.3622986

[15] Ivan Jibaja, Stephen M Blackburn, Mohammad R. Haghighat, and Kathryn S McKinley. 2011. Deferred Gratification: Engineering for High Performance Garbage Collection from the Get Go. In *Proceedings of the 2011 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness* (San Jose, California) *(MSPC '11)*. ACM, New York, NY, USA, 58–65. doi:10.1145/1988915.1988930

[16] Henry Lieberman and Carl Hewitt. 1983. A Real-Time Garbage Collector Based on the Lifetimes of Objects. *Commun. ACM* 26, 6 (1983), 419–429. doi:10.1145/358141.358147

[17] Yi Lin, Kunshan Wang, Stephen M. Blackburn, Antony L. Hosking, and Michael Norrish. 2015. Stop and go: understanding yieldpoint behavior. In *Proceedings of the 2015 ACM SIGPLAN International Symposium on Memory Management, ISMM 2015, Portland, OR, USA, June 13-14, 2015*, Antony L. Hosking and Michael D. Bond (Eds.). ACM, 70–80. doi:10.1145/2754169.2754187

[18] Aaron Patterson. 2019. Manual Compaction for MRI's GC (`GC.compact`). https://bugs.ruby-lang.org/issues/15626

[19] Android Open Source Project. 2017. Perfetto. https://perfetto.dev/

[20] Robert A. Saunders. 1964. The LISP System for the Q-32 Computer. In *The Programming Language LISP: Its Operation and Applications*, Edmund C. Berkeley and Daniel G. Bobrow (Eds.). The M.I.T. Press, 220–238.

[21] 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

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

[23] David M. Ungar. 1984. Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, Pennsylvania, USA, April 23-25, 1984*, William E. Riddle and Peter B. Henderson (Eds.). ACM, 157–167. doi:10.1145/800020.808261

[24] Xi Yang, Stephen M. Blackburn, Daniel Frampton, and Antony L. Hosking. 2012. Barriers reconsidered, friendlier still!. In *International Symposium on Memory Management, ISMM '12, Beijing, China, June 15-16, 2012*, Martin T. Vechev and Kathryn S. McKinley (Eds.). ACM, 37–48. doi:10.1145/2258996.2259004

[25] Wenyu Zhao, Stephen M. Blackburn, and Kathryn S. McKinley. 2022. Low-latency, high-throughput garbage collection. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, Ranjit Jhala and Isil Dillig (Eds.). ACM, 76–91. doi:10.1145/3519939.3523440

[26] Peter Zhu. 2022. Optimizing Ruby's Memory Layout: Variable Width Allocation. https://shopify.engineering/ruby-variable-width-allocation

[27] Peter Zhu. 2023. Garbage Collection in Ruby. https://blog.peterzhu.ca/notes-on-ruby-gc/

[28] Peter Zhu. 2023. Weak References in the GC. https://bugs.ruby-lang.org/issues/19783

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