Arborescent Garbage Collection: A Dynamic Graph Approach to Immediate Cycle Collection

Frédéric Lahaie-Bertrand
frederic.lahaie-bertrand@umontreal.ca
Université de Montréal
Montréal, Canada

Léonard Oest O'Leary
leonard.oest.oleary@umontreal.ca
Université de Montréal
Montréal, Canada

Olivier Melançon
olivier.melancon.1@umontreal.ca
Université de Montréal
Montréal, Canada

Marc Feeley
feeley@iro.umontreal.ca
Université de Montréal
Montréal, Canada

Stefan Monnier
monnier@iro.umontreal.ca
Université de Montréal
Montréal, Canada

Abstract

Reclaiming cyclic garbage has been a long-standing challenge in automatic memory management. Common approaches to this problem often involve extending reference counting with an asynchronous background task to reclaim cycles. While this ensures that cycles are eventually collected, it also introduces unpredictable behaviours, making these approaches unsuitable for applications where deterministic collection is required.

This paper introduces Arborescent Garbage Collection, a synchronous memory management algorithm that immediately reclaims unreachable memory objects, including cyclic structures. Inspired by single-source reachability algorithms on dynamic graphs, it extends the idea of embedding a spanning forest in a program's reference graph to track the reachability of any object from a root. When a reference is removed, the algorithm efficiently rebuilds the forest and immediately reclaims the memory of objects that are no longer reachable. The result is a garbage collection algorithm suitable for applications that require immediate memory reclamation and predictable behaviour.

CCS Concepts: • **Software and its engineering** → **Garbage collection**; • **Theory of computation** → *Dynamic graph algorithms.*

Keywords: automatic memory management, reference counting, cycles, synchronous garbage collection

ISMM '25, Seoul, Republic of Korea

ACM ISBN 979-8-4007-1610-2/25/06
https://doi.org/10.1145/3735950.3735953

ACM Reference Format:
Frédéric Lahaie-Bertrand, Léonard Oest O'Leary, Olivier Melançon, Marc Feeley, and Stefan Monnier. 2025. Arborescent Garbage Collection: A Dynamic Graph Approach to Immediate Cycle Collection. 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/10.1145/3735950.3735953

1 Introduction

Fully automatic memory management techniques are generally divided into two categories: tracing and reference tracking. Tracing consists in finding all reachable objects starting from roots, often stopping the program or needing an additional background or asynchronous process. Examples include Stop-and-Copy and Mark-and-Sweep. On the other hand, reference tracking attaches runtime information to objects and detects when objects become unreachable. Historically, reference tracking has been associated with reference counting [8], which only tracks the number of objects pointing to an object, deallocating objects when nothing points to them. However, other information can be tracked, such as the path to root objects or referrers of an object.

A recurring problem with reference tracking techniques is their inability to collect cycles immediately. Performing a systematic, full tracing from an object to a root has been proposed [18], but it is prohibitively expensive. For this reason, cycle collection is generally deferred to a concurrent process [2], which delays collection. Other works proposed embedding a spanning tree into objects [5–7, 22] to reliably detect when a cycle becomes unreachable without full tracing to a root. However, these techniques are too costly to be used synchronously, thus requiring a concurrent process to maintain the spanning tree [5, 12, 17, 26].

Synchronous and immediate collection reclamation is needed in applications that require deterministic and reliable behaviour. Situations where this can be desirable include:

- Object stores such as graph databases.
- File-systems that allow hard links to directories.

- Reclamation of DOM objects in a browser.
- The interoperability of a garbage collector with non-automatic memory management languages, such as C++ or Rust.
- Precise detection of memory leaks in non-automatic memory management systems.
- Precise measurement of the maximum amount of live heap data during a program execution.
- Reliable execution of finalizers, since programmers are frequently discouraged from using them due to their unpredictable behaviour with asynchronous garbage collection [3, 21, 23, 25].
- Management of control-flow graphs (CFGs) in compilers where the CFG is modified incrementally and were prompt reclamation of unreachable basic blocks can enable further optimizations.

This paper presents the Arborescent garbage collector, a reference tracking algorithm that collects cycles immediately and synchronously. Inspired by Even and Shiloach's single-source reachability algorithms from dynamic graph theory [10], the Arborescent GC encodes spanning trees into objects to treat reachability as an on-line edge-deletion reachability problem. The novelty of the Arborescent GC is to relax the usual notion of rank used by Even-Shiloach algorithm (distance from the root) to reduce the cost of updating ranks when a reference is mutated while still providing enough information to guide heuristics with the efficient reparation of the spanning tree. This greatly reduces the time spent in the collector in comparison to previous synchronous techniques. This optimization allows the Arborescent GC to synchronously reclaim all memory at the cost of a 4.5× slowdown compared to Mark-and-Sweep.

The contributions in this paper are:

- A memory reclamation algorithm that synchronously reclaims memory, including in the presence of cyclic structures, without the prohibitive overhead of previous techniques (Section 3);
- A detailed presentation of the algorithm's implementation and object model to ensure that no memory allocation takes place during the collection phase (Section 4);
- An implementation of that algorithm in the Ribbit System [20] (Section 5).

2 Related Work

One of the first solutions to the cycles problem is from Lins [18] who proposed an incremental mark-scan phase that explores potential cycles and deallocates structures for which no external reference was found. Bacon and Rajan [2] later showed that efficient reference counting can be implemented by deferring cycle collection to a concurrent process, which minimizes mutator pauses. Incremental and concurrent extensions to reference counters avoid long pauses in program execution for collecting cycles, but do not allow immediate cycle collection.

Brownbridge [7] and Piquer [22] introduced mechanisms for maintaining a spanning tree in a reference graph. Any deletion of a reference that is not in the spanning tree can thus be safely ignored by the garbage collector. More recently Brandt et al. [5, 6] described an algorithm for repairing the spanning tree when a reference is deleted, reclaiming unreachable objects in the process. Despite this progress, the cost of tracing the reachability of objects detached from the spanning tree is currently too prohibitive for a synchronous garbage collector. Deletion of a reference potentially induces the exploration of a significant part of the reference graph, even when in reality no object was made unreachable. Hence, in existing implementations, tracing is deferred to an incremental or concurrent process [5, 12, 14, 17, 26].

In the graph theory community, Even and Shiloach proposed a dynamic graph algorithm that efficiently detects which subgraph becomes disconnected after the deletion of an edge [10]. Their original algorithm tracks reachability by maintaining the rank of each node in the graph. This is both expensive and not necessary for memory management. Yet, Even and Shiloach's algorithm inspired the Arborescent GC which incorporates a weaker notion of rank by loosening the requirement that ranks increase by increments of one. This reduces the cost of updating ranks on edge deletion while still storing enough information to guide heuristics when tracking reachability.

More recently, Sotoudeh [27] used a similar approach by designing a garbage collector that maintains a spanning forest of the reachable objects on the heap but using Euler tour trees instead of Even-Shiloach trees. Although their motivation is to prove a theoretical lower bounds of the cost of garbage collection, they also built a garbage collector that can immediately collect garbage.

3 Reachability Algorithm for Synchronous Garbage Collection

This section presents an algorithm for garbage collection that supports immediate cycle collection. At its core, the algorithm establishes objects liveness by maintaining a path from each object to a root, such as a global or stack variable. This is achieved by embedding a spanning forest within a program's reference graph. When a reference in the spanning forest is removed, the algorithm efficiently rebuilds the forest and immediately deallocates any objects that are no longer reachable. Section 3.1 provides some definitions and Section 3.2 details the algorithm.

3.1 Definitions

3.1.1 Reference Graph. The relationships between objects in memory can be modelled by a *reference graph*, which is a directed, possibly disjoint, graph whose nodes represent

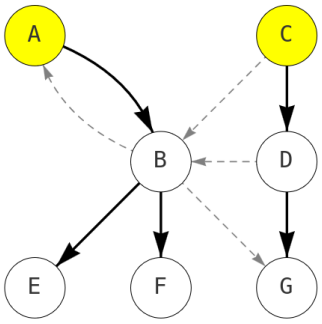


Figure 1. A reference graph with two uncollectable nodes (A and C, in yellow). Bold edges are part of spanning trees and dashed edges are coparent-cochild relations that are not part of the trees. The node B has parent A, and children E and F. Its coparents are C and D, and its cochildren are A and G.

objects and edges represent pointers between objects. In this paper, an *object* refers to any address in memory containing data, including memory locations containing pointers to stack and global variables. Self references are not included in the graph as they do not affect reachability.

A reference graph possesses a set of *uncollectable* nodes, which act as the *roots* for the garbage collector. These nodes typically correspond to global variables and stack variables.

An object is *reachable* if there exists a path from an uncollectable object to this object. Objects' reachability is tracked by inscribing a disjoint collection of spanning trees, with uncollectable objects as roots, in the reference graph. This creates two classes of relations in the graph: edges that are part of a spanning tree (called parent-child edges) and edges that are not.

Definition 3.1 (Coparent and cochild)**.** If an edge from A to B is not part of one of the current inscribed spanning trees in the reference graph, then A is said to be a *coparent* of B, and B is said to be a *cochild* of A.

Figure 1 shows an example of reference graph, including coparent-cochild relations and two inscribed spanning trees, forming a spanning forest.

Section 4 describes an object encoding to embed the reference graph within objects, and efficiently access and distinguish between an object's parent, children, coparents, and cochildren.

3.1.2 Ranks. Each object in the reference graph is assigned a *rank*. This notion of rank is less strict than the usual definition, which is the distance from a root.

Definition 3.2 (Rank)**.** The rank of an object is an integer that must be strictly greater than the rank of the object's parent. If the object is uncollectable, in which case it has no parent, then it can have any integer as its rank.

```
1 function removeEdge(from: Node, to: Node):
2     delete reference from → to
3     if to.parent is from then
4         to.parent ← null
5         if not adopt(to) then
6             drop(to)
```

Algorithm 1: Remove an edge from the reference graph.

The ranks of objects along each branch of the spanning forest are always in strictly increasing order. This is used to locally verify the absence of cycles in the parent-child edges. This often allows to rapidly assert that an object cannot be the descendant of another in the spanning forest. Since ranks do not have to increase by increments of one along a branch, an object can have more than one valid rank. This flexibility can reduce the time spent updating ranks in response to mutations in the spanning forest.

3.2 Maintaining the Spanning Forest

A garbage collector can be implemented by ensuring that, given a reference graph with only reachable nodes, all operations on the reference graph maintain a spanning forest. To achieve this, the runtime system only manipulates references by calling atomic operations handled by the garbage collector. These include adding and removing a reference, making an object collectable or uncollectable, and allocating a new object.

Adding a reference, making an object uncollectable and allocating an object are operations that cannot cause objects to become unreachable. On the other hand, removing a reference and making an object collectable can make objects unreachable, requiring that they be collected. The next sections detail how to implement each operation such that a spanning forest is always maintained on the reference graph and unreachable objects are immediately collected.

3.2.1 Adding an Edge. Contrary to Even and Shiloach's algorithm [10], a GC cares only about reachability and not distance, so the addition of a reference in the graph never needs to modify the spanning forest: the algorithm simply always labels the new edge as a coparent-cochild relation.

3.2.2 Removing an Edge. When a reference is removed, the garbage collector checks whether this reference forms a parent-child edge in the reference graph. If it does not, then removing the edge is guaranteed to preserve reachability, and the reference can safely be deleted while keeping the spanning forest unchanged.

On the other hand, if the removed edge is part of the spanning forest, a subtree of the spanning forest was disconnected and its nodes need to be either collected or reconnected to the forest. This forms the core of the algorithm. The nodes of this subtree are now said to be *loose* and all the external

```
1  function drop(node: Node):
2      anchors ← new Queue
3      todo ← new Queue
4      node.loose ← true
5      todo.enqueue(node)
6      while node ← todo.dequeue() do
7          foreach child of node do
8              if not adopt(child) then
9                  child.loose ← true
10                 todo.enqueue(child)
11         foreach coparent of node do
12             if not (coparent.loose or coparent in anchors) then
13                 anchors.enqueue(coparent)
14     catch(anchors)
15     collect(node)
```

Algorithm 2: Traverse spanning tree to find potentially unreachable nodes.

```
1  function catch(anchors: Queue):
2      while anchor ← anchors.dequeue() do
3          if not anchor.loose then
4              foreach cochild of anchor do
5                  if cochild.loose then
6                      cochild.loose ← false
7                      cochild.parent ← anchor
8                      cochild.rank ← anchor.rank + 1
9                      anchors.enqueue(cochild)
```

Algorithm 3: Repair the spanning with a traversal from anchor nodes.

```
1  function collect(node: Node):
2      if node.loose then
3          foreach child of node do
4              collect(child)
5          dealloc node
```

Algorithm 4: Deallocate unreachable (loose) nodes.

references to them are called *anchors*, i.e. nodes that are not loose but that have a loose node as cochild. The algorithm proceeds by finding all the anchors and reattaching all the loose nodes that it can before collecting the left overs.

The entry point is the `removeEdge` procedure (Algorithm 1). If the edge is a parent-child edge, the deletion of the reference causes the child object to lose its parent. To try and avoid traversing the whole subtree of loose objects, the garbage collector first attempts to find, among its coparents, a node which could be used immediately as a new parent, i.e. a coparent whose rank is smaller. This is done by the `adopt` procedure, which will be detailed in Section 3.2.3. If adoption fails, the garbage collector calls the `drop` procedure to repair the spanning forest, collecting any unreachable objects in the process.

The `drop` procedure (Algorithm 2) is given a collectable object that lost its parent after the deletion of a reference. This object is marked as *loose*, meaning it may potentially have become unreachable.

Any child whose parent is loose must become loose as well. The spanning tree is thus traversed in a breadth-first order to recursively mark descendants as *loose* and, at every step, the `adopt` procedure attempts to stop the recursion early.

Additionally, `drop` adds all the loose object's coparents that are not loose themselves to a queue called the *anchors* queue, which will later be used to reattach loose (but reachable) objects to the spanning forest. Since coparents can have more than one loose cochild, the coparent is checked to ensure they aren't already in the anchors queue before enqueuing (Section 4 explains how to efficiently implement this).

When the `drop` traversal terminates, the anchors queue is passed to the `catch` procedure (Algorithm 3). This procedure traverses the spanning forest starting from the anchors and reattaches the referenced loose objects. Since some objects may have been added to the anchors queue prior to having been marked as loose, all dequeued object must be checked and skipped if loose. If a dequeued object is not loose, then the garbage collector looks for its loose cochildren. Such cochildren are unmarked as loose, adopted by the object, and queued in the anchors queue for traversal. The objects contained in the anchor queue will generally have a bigger rank than their adoptees, since otherwise `adopt` would have succeeded. So as to maintain a strictly increasing order of ranks along the spanning tree, the rank of these adopted objects is set to one more than the rank of their new parents.

When the `catch` phase is over, a final traversal takes place starting from the object that was targeted by the initial call to `removeEdge` and visiting only objects that are still marked as loose. These objects are unreachable and can be deallocated (Algorithm 4), which includes removing them from the set of coparents of their cochildren. Figure 2 illustrates the state of the reference graph after the `drop`, `catch` and `collect` phases of an edge removal.

3.2.3 Adoption. The `drop` procedure traverses whole spanning trees, which becomes prohibitively expensive as the reference graph grows. This section describes an optimization called *adoption* that attempts to quickly find a more suitable parent for objects whose parent is loose or was removed.

To pick a new parent (called an *adopter*) for the object (called the *adoptee*), the `adopt` procedure (Algorithm 5) first

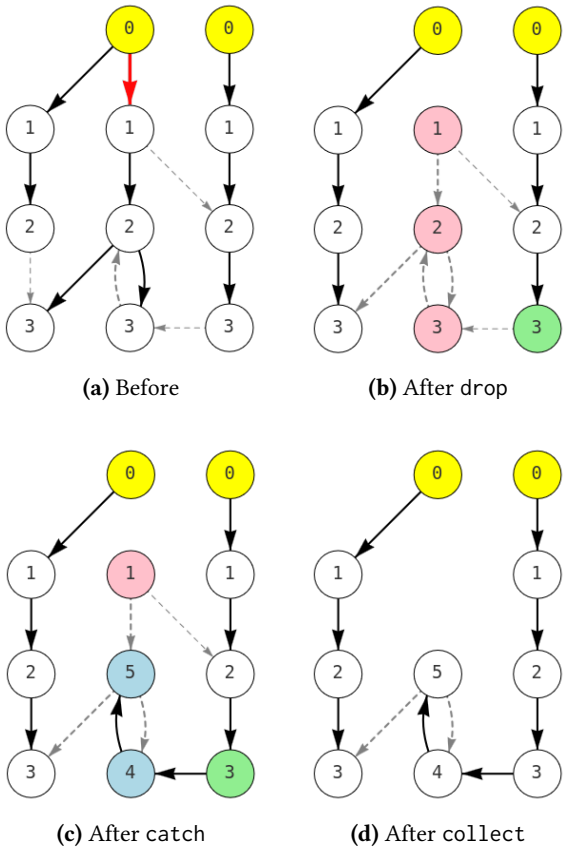


Figure 2. State of the reference graph after each phase of a parent-child reference removal. Each object is labelled with its rank, uncollectable objects are highlighted (in yellow). (a) is the initial graph with the edge to be removed (bold red). (b) is the graph after the `drop` phase with loose nodes (in pink) and anchors (in green). Note how the bottom-left node is not loose due to being successfully adopted. (c) is the state of the graph after the `catch` phase with previously loose node that were caught (in blue). (d) is the final graph after deallocation of the unreachable node.

looks for a coparent that has a rank smaller than that of its potential adoptee. This ensures that the adopter is not a descendant of the adoptee, since descendants have greater ranks. If such an adopter is found, it is set as the new parent of the adoptee, which interrupts the traversal of the adoptee in the `drop` phase.

If no adopter is found, the garbage collector attempts to rerank a coparent such that it becomes a valid adopter. Given an object of rank R_{obj} and a coparent of rank $R_{co} \geqslant R_{obj}$, decrementing the rank of the coparent to $R_{obj} - 1$ would turn it into a valid adopter.

There are two situations where ranks can be decremented without breaking the increasing order of ranks in the spanning forest. First, since uncollectable objects have no parent, their rank can be decremented (but not incremented) freely.

```
1  function adopt(node: Node):
2      foreach coparent of node do
3          if (not coparent.loose and
               coparent.rank < node.rank) then
4              node.parent ← coparent
5              return true
6      foreach coparent of node do
7          if heuristic(coparent) then
8              if rerank(coparent, node, node.rank - 1)
                 then
9                  node.parent = coparent
10                 return true
11     return false
```

Algorithm 5: Attempt to pick a valid parent for a node.

```
1  function rerank(node: Node, origin: Node, to: int):
2      if node.loose or node == origin then
3          return false
4      else if (node.uncollectable or
                node.parent.rank < to or
                rerank(node.parent, origin, to - 1)) then
5          node.rank = to
6          return true
7      return false
```

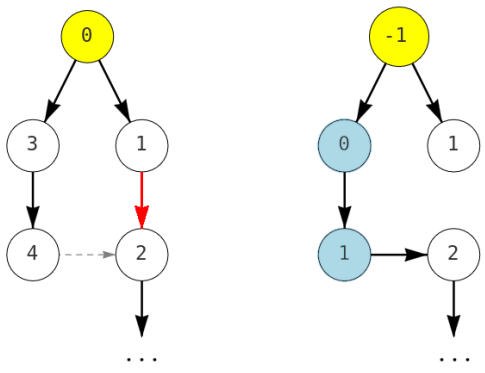
Algorithm 6: Attempt to rerank ancestors to allow adoption.

Second, whenever the rank of an object is not exactly one more than that of its parent, it can be decremented by up to $R_{ch} - R_{pa} - 1$, where R_{ch} is the rank of the child and R_{pa} is the rank of its parent.

The procedure `rerank` (Algorithm 6) implements the attempt to rerank a coparent to make it a valid adopter. It recursively explores the ancestors of the given coparent, looking for opportunities to decrement ranks. If sufficient gaps between ranks are found or an uncollectable object is reached, then reranking takes place, as illustrated in Figure 3. If the original adoptee is found among ancestors, then the coparent has been found to be a descendant of the adoptee and cannot be reranked. For each coparent, a `heuristic` determines whether the reranking should be attempted.

In the experiment presented in Section 5, reranking is only attempted for the first five objects traversed by the `drop` phase. While extremely simple, this heuristic yields decent performance. A better heuristic could consider parameters such as objects ranks, types, or information from static program analysis.

Note that an adoption does not prevent an adoptee from later being marked as loose. This can happen when the `drop` procedure later reaches the adopter. For this reason `drop` is



(a) Before `rerank` **(b)** After `rerank`

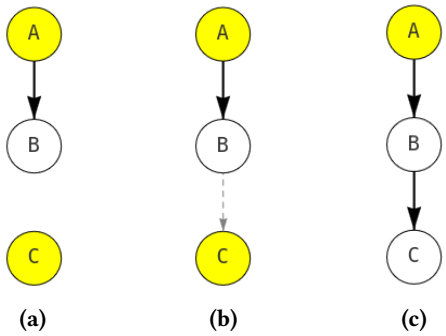
Figure 3. State of the reference graph before and after a successful reranking by the `adopt` phase. Each object is labelled with its rank, the uppermost object (in yellow) is uncollectable. (a) is the initial graph with the edge to be removed (bold red). (b) is the graph after a successful reranking. The `rerank` procedure explores the dropped object's coparent (bottom left) with the request that its rank be decremented to 1. The rank of the root is decremented by one, which in turn permits decrementing the rank of its descendants (in blue) by three, thus allowing the bottom right object to be adopted.

careful to traverse the subtree in breadth-first order, which tends to traverse objects in increasing order of their rank and thus reduces the risk of this occurring. While it is still possible that an object's adoption only defers the moment when it is marked loose, in practice this heuristic frequently prevents the traversal of a significant part of the reference graph.

3.2.4 Making Uncollectable. An object can be made uncollectable, for instance if the runtime system needs to acquire it and ensure that it is kept alive. When an object is labelled as uncollectable, its parent is relabelled as a coparent, replacing the corresponding edge from the spanning tree to form a coparent-cochild edge and making it the root of a new spanning tree in the reference graph.

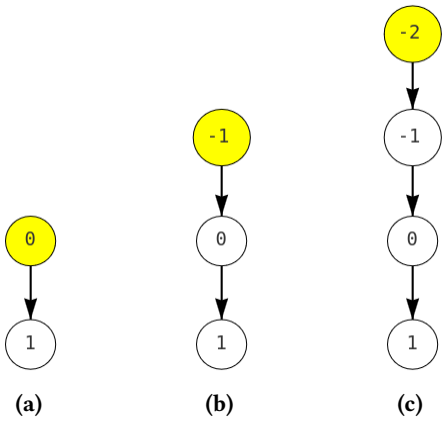
3.2.5 Making Collectable. When an uncollectable object is made collectable, it stops being the root of a spanning tree. This causes the object to become a collectable object with no parent. To repair the spanning forest and find newly unreachable objects, the `drop` procedure (Algorithm 2) is called on this object, as in the case where a parent-child edge is removed.

3.2.6 Object Allocation. A newly allocated object is initialized as uncollectable. This ensures that all objects, even newborn ones, are reachable from a root. In most cases, this



(a) **(b)** **(c)**

Figure 4. (a) Object `C` is allocated and initialized as uncollectable (denoted by yellow). (b) `A` reference to `C` is added to object `B`. (c) Object `C` is finally made collectable, making it part of the spanning tree rooted at object `A` (for instance the stack) and setting its rank to that of `B` plus one.



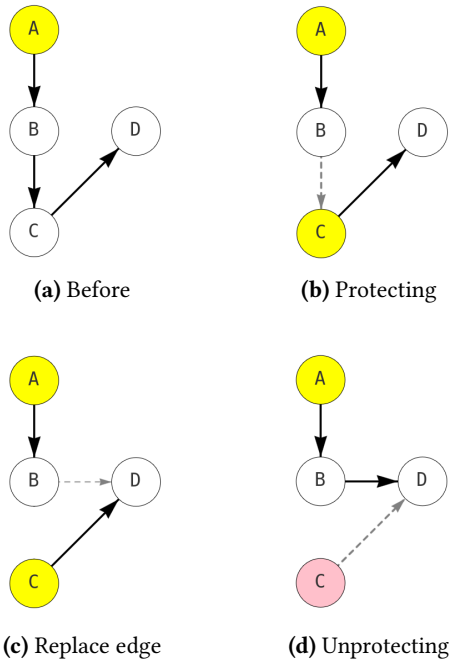
(a) **(b)** **(c)**

Figure 5. Creation of a linked-list with recursive consing. A global counter is used to assign ranks to newly allocated objects. The counter is decremented after each allocation to ensure that the head (in yellow) has the smallest rank among all objects and can thus adopt the tail of the list.

uncollectable state is temporary. The runtime system typically acquires the object, adds a reference to it (on the stack for instance), then makes it collectable. This calls the `drop` procedure (Algorithm 2), which adjusts its rank to one more than that of its new parent, as illustrated in Figure 4.

This works well for structures built from top to bottom, such as iteratively appending objects to a linked-list. However, for objects built from bottom to top, for instance creating a linked-list with recursive consing, the rank of the allocated object matters. The initial rank of the object dictates whether it will be a valid adopter or need reranking.

To address this issue, a global counter, initialized at 0, is kept for ranking new objects. The rank of allocated object is set to the counter's current value. After each allocation, the counter is *decremented* (negative ranks are allowed). Consequently, right after its allocation, an object always has a



(a) Before **(b)** Protecting

(c) Replace edge **(d)** Unprotecting

Figure 6. Replacing an edge requires (b) protecting the old referee (`C`) by making it uncollectable, (c) updating the new reference, (d) and finally unprotecting the old referee. In the above example, failing to protect `C` would lead to the premature deallocation of `D`.

lower rank than objects allocated before it. This ensures that objects created from bottom to top, such as in Figure 5, are allocated in linear time.

With this initialization scheme, the initial rank of an object is a proxy for its age. This means that in the absence of object mutation, since an object can point only to objects older than itself, `adopt` will always succeed immediately without the need for `rerank`. In other words, `drop` only needs to be used when removing a reference that was added by mutation.[1]

3.2.7 Updating References. Overwritting a pointer can be implemented as the addition of a new edge followed by the removal of an old one. In practice, however, objects have a fixed number of fields to store outgoing edges and coparents (more on that in Section 4), which requires that the removal of the old edge takes place first. Yet, removing the old edge may cause objects to be immediately deallocated, including objects that would be reachable after adding the new edge. To solve this issue, when an edge is replaced, the old referee must first be protected by making it uncollectable. Only after the new edge has been added is the old referee made collectable again. As illustrated in Figure 6, this has the effect of delaying the `drop` phase until the edge has been replaced.

[1]Of course, when `catch` reattaches the objects, it may cause immutable fields to point to objects with a lower rank.

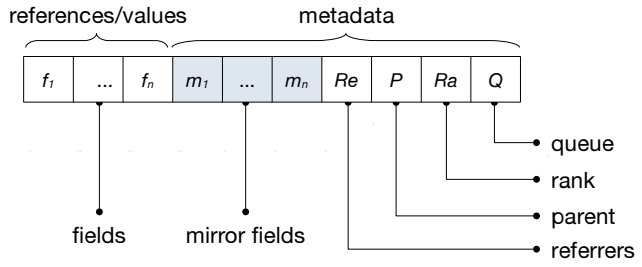


Figure 7. Layout of an object's garbage collection metadata. The first n fields (f_1 to f_n) contains actual references to objects. The following fields contain metadata. Fields m_1 to m_n are mirror fields used to chain the object's referrers. Field *Re* points to the first object in its chain of referrers. Field P points to the parent of the object. Field *Ra* contains the rank of the object. Field Q is used to chain values in the queue and anchor queue during the `drop` and `catch` phases.

4 Implementation

This section describes a memory representation that embeds the necessary metadata within objects to efficiently execute the algorithm from Section 3 and ensure that the garbage collector does not allocate memory during collection. This includes extra space for back references, queues, ranks, and tags for loose objects.

4.1 Encoding the Reference Graph

In addition to its actual fields (simply called *fields* in this section), which contains data such as references, objects are augmented with metadata to inscribe the reference graph and spanning forest. The referrers of an object (parent and coparents) are chained together with the use of special fields, called *mirror fields*. Given an object with n fields, it is extended with n mirror fields and one *referrers field*, which stores its first coparent. Each object also has a *parent field*, which points to its parent. Uncollectable objects are denoted by a `NULL` parent field. Additionally, each object has a *rank field* to store its rank, an integer. However, in practice, a full 64-bit is not required to store the rank. It is thus convenient to use a 63-bit rank and reserve one bit for marking loose objects. Finally, a *queue field* is used for implementing queues (discussed in Section 4.3). Figure 7 illustrates the layout of an object's metadata.

Figure 8 illustrates how referrers are chained with mirror fields. Given an object A whose i^{th} field contains a reference to a child (or cochild) C, then the i^{th} mirror field of A is reserved for chaining the referrers of C. Iterating over referrers is done by recursively following the corresponding mirror fields of each referrer, starting from the first one and ending when the referrer's mirror field contains `NULL`. In the case where the object A contains more than one reference to C,

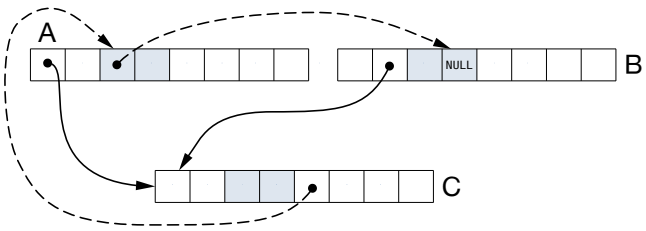


Figure 8. Memory layout of the chaining of referrers for objects containing two fields. Solid edges represent pointers from objects' fields (first two slots of objects). Dashed edges represent edges from objects' mirror fields (next two slots of objects, in grey). Object C has two referrers, A and B. The referrers field (next slot after mirror fields) of C points to its first referrer (A). Since A points to C with its first field, then its first mirror field points to B, the next referrer to C. Object B points to C with its second field and is the last referrer of C, thus its second mirror field contains `NULL`.

the corresponding mirror fields are all made to point to the same successor in the referrers chain.

An offset is added to pointers in mirror fields to make them point directly to the next mirror field in the chain. For this to work, objets need to be aligned in memory. Figure 8 illustrates an example of referrers chaining with 64-byte aligned objects. In this example, the object *A* refers to *C* from its first field. Hence, an offset is added to the referrers field of *C* to point to the first mirror field of *A*. Since *A*, *B* and *C* are 64-byte aligned, masking the low bits of mirror fields allows recovering an object's address. In cases where this solution is not applicable, an implementation can add extra fields to each object to store offsets. Alternatively, for objects with few fields, it may be sufficient to scan references to compute the offset instead of storing it.

Iterating over an object's children or cochildren is done by visiting its referees, which are canonically stored in its fields. Children are such referees whose parent is the object itself while cochildren are referees that are either uncollectable or whose parent field points to some other object.

4.2 Removing Referrers

Removing a reference from object *A* to object *B* requires removing *A* from *B*'s chain of referrers. While costly when an object has many referrers, in practice, most objects have few referrers. Moreover, there are cases where the traversal of the chain of referrers can be skipped altogether.

One such case is objects that are known to be permanently uncollectable, for instance singletons such as `true`, `false` and `none` that are heap-allocated by some implementations. The referrers of these objects do not need to be tracked since they will never need to be adopted. By creating two classes of uncollectable objects, temporary and permanent, tracking these objects' referrers can be avoided. This is especially convenient since these are likely to have numerous referrers.

Another optimization to referrer removal happens at object deallocation. An unreachable object is no longer a valid coparent and has to be removed as a referrer. Implicit in the `dealloc` operation of Algorithm 4 is the traversal of cochildren of loose objects to remove any loose objects in their referrers. However, any cochild that is itself loose can be skipped as it will also be deallocated.

4.3 Queues and Traversal

Queues are implemented by using the *queue field* for chaining objects, and keeping pointers to both extremities of each queue. This allows efficient enqueue, dequeue, and checks that an object is not already in the queue by checking that the object is neither the tail of the queue nor does it have a successor stored in the corresponding queue field (done before enqueuing in *anchors* in Algorithm 2).

The `drop` phase of the algorithm requires two queues (*todo* and *anchors* in Algorithm 2). However, the *todo* queue only contains loose objects, whose rank will never be read before being updated in the `catch` phase. Consequently, the rank field can be used to implement the *todo* and only one field must be added for the *anchors*.

4.4 Full Graph Reranking

Since objects' ranks are limited to fixed size integers, overflows are possible. Furthermore, as described in Section 3.2.6, the garbage collector maintains a decremented counter for the rank of newly allocated objects, which introduces the possibility of underflows. This requires the garbage collector to check for overflows and underflows when ranks are updated and the counter is decremented respectively.

In case of an overflow, or underflow, the garbage collector is interrupted to balance the whole reference graph. This entails resetting the global counter (possibly, but not necessarily, to zero), choosing a new rank for each uncollectable object, reranking the graph from its roots, and recursively reassigning ranks. If the interruption occurred in an edge removal, objects that were loose are unmarked and the `drop` phase restarts from the beginning after the reranking.

On 64-bit architectures, these occurrences are expected to be very rare. In fact, they never occur in any benchmarks from Section 5, but could happen on long-running programs.

5 Evaluation and Benchmarks

This section evaluates the runtime performance of Arborescent garbage collection in comparison to Mark-and-Sweep, a well-established memory management technique. It demonstrates that while Arborescent GC incurs an overhead for immediate cycle collection, it yields performance that can make it suitable in domain-specific applications that require immediate reclamation.

The Arborescent GC presented in this section is implemented in Ribbit Scheme. Ribbit has an extensible memory model that allows the efficient implementation of new garbage collectors [19]. It already has a Mark-and-Sweep GC, which is commonly used as a comparison for new garbage collection algorithms as it has been widely studied and is known to be efficient.

Section 5.1 describes Ribbit in more details, Section 5.2 shows experimental execution time results and Section 5.3 presents profiling data of the Arborescent GC.

5.1 The Ribbit System

Ribbit is an ahead-of-time compiler for the Scheme programming language, which supports the R4RS standard. It compiles programs to Ribbit Virtual Machine (RVM) instructions that can then be interpreted by a RVM implementation. RVM's have been written in more than 25 different programming languages, but the experiment in this section was done using the more performant C RVM.

The RVM interpreter has an unusual design in which both the stack and code and all Scheme objects are implemented as linked structures called ribs, objects consisting of three fields. As a result, most of the interpreter's memory management activity is not directly driven by the source program, but by the interpreter's own execution mechanics. For instance, the source-level creation of a rib with the expression `(##rib 1 2 3)` actually allocates a total of 5 ribs because 4 ribs are used to store the arguments and result on the stack. Ribs are also accessed to push and pop the values from the stack and advance the *program counter* in the course of execution (each RVM instruction is itself stored in a rib containing a link to the next instruction rib).

In essence, the Ribbit interpreter behaves like a compiled program where the majority of execution time is spent on operations that affect the heap, i.e. memory allocations and mutations to heap objects and root pointers, rather than on data-level operations. This makes it a good GC torture-test from a performance standpoint: it exhibits a high ratio of memory management overhead relative to actual computation.

5.2 Experimental Results

Arborescent garbage collection was implemented in the `C` RVM and compared to the Mark-and-Sweep GC already available in Ribbit. Performance was evaluated by comparing execution time with the R7RS Scheme benchmark suite [1]. Since Ribbit only supports the R4RS Scheme specification, benchmarks that extensively use features from later specifications were excluded. In total, 34 out of the 57 benchmarks from the R7RS suite could be executed. Benchmarks were run on a machine with an Intel Xeon Phi CPU 7210, 112 GB of RAM, and under CentOS 7 with Linux kernel version `3.10.0-1160.119.1.el7.x86_64 SMP`. Each benchmark was parameterized so that executions lasted at least 30 seconds and each execution was repeated five times for both garbage collectors. Execution times presented in this section are the average of these five runs.

Benchmark	Heap (kB)	Benchmark	Heap (kB)
ack	574	array1	472
boyer	5,795	browse	1,247
cpstak	241	ctak	260
deriv	287	destruc	328
diviter	247	divrec	245
earley	1,899	equal	19,570
fibc	243	fib	220
gcbench	9,057	graphs	11,495
lattice	349	matrix	611
mazefun	496	maze	1,097
mperm	39,923	nboyer	19,520
nqueens	300	ntakl	280
paraffins	43,901	peval	1,303
primes	69,337	puzzle	1,125
quicksort	538	sboyer	5,240
scheme	1,654	string	1,920
sum	229	tak	224

Figure 9. Minimum heap size required for each benchmark.

The Arborescent and Mark-and-Sweep implementations allocate objects from a freelist whose size is parameterized but fixed at runtime. An interesting capability of the Arborescent garbage collector is to seamlessly compute the maximum number of objects that are alive at any given time. This is achieved by profiling a separate run of each benchmark and incrementing/decrementing a counter whenever an object is allocated/deallocated. Since objects are deallocated immediately when unreachable, the maximum value reached by that counter corresponds to the minimum required size of the freelist. Multiplying by the size of an object (11 machine words in Ribbit with the Arborescent GC[2]) then yields the minimum required heap size in bytes, which is shown in Figure 9.

Each benchmark is executed with the minimum heap size required by the Arborescent GC for this benchmark (from Figure 9). The same heap size is used for the Mark-and-Sweep GC execution. Since objects with Mark-and-Sweep take 3 machine words (11 with the Arborescent GC), this ensures that at most 3/11 of the heap contains live data at any point.

Figure 10 shows the relative execution time between benchmarks executed with each GC. Arborescent garbage collections makes for an execution that is at most 8.6× slower (`paraffins`, discussed in Section 5.3) than Mark-and-Sweep, with a median of about 4.5× slower.

[2]Figure 7 shows that only 10 fields could be used by sharing the same field for `rank` and the *todo* queue. However, Ribbit does not use this optimization.

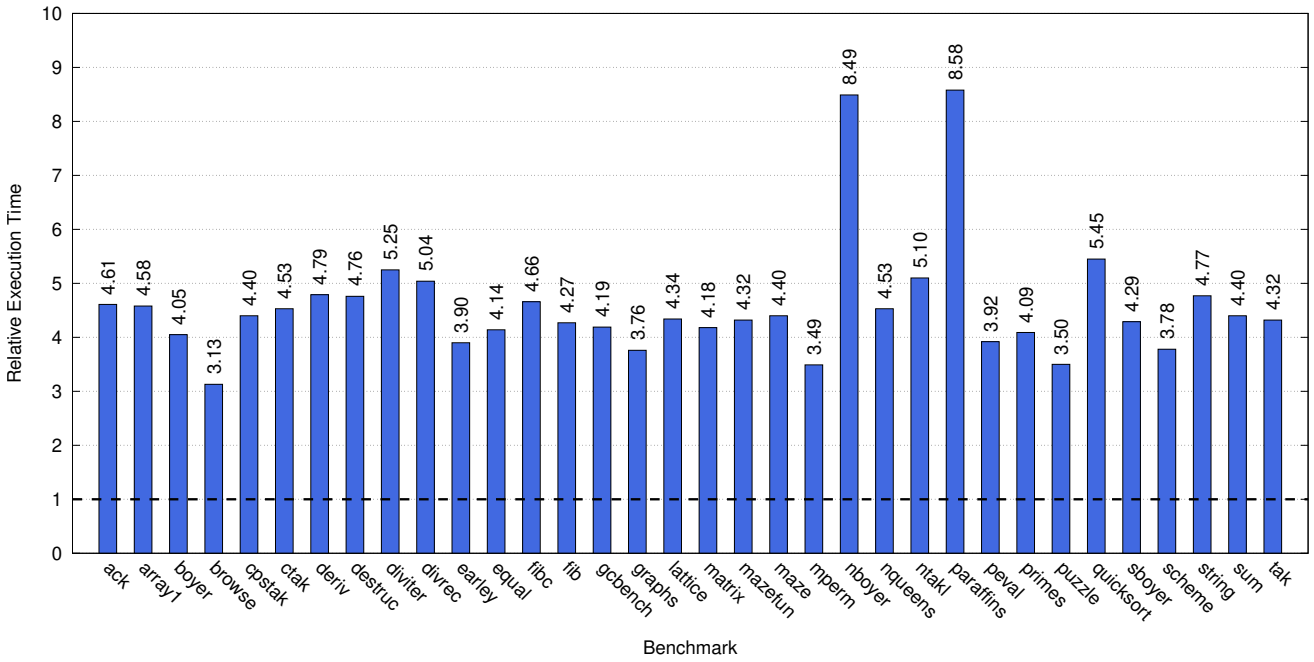


Figure 10. Execution time with Arborescent garbage collection relative to Mark-and-Sweep measured with R7RS benchmarks. The heap size used for each benchmark is found in Figure 9 and is the same for both garbage collectors for a given benchmark. A ratio of 1 (dashed line) indicates an execution time equal to that of the Mark-and-Sweep implementation. Lower is faster.

These results demonstrate that while an Arborescent GC is generally slower than other memory management techniques, it is not prohibitively expensive like previous synchronous techniques: even in Ribbit where every instruction of the VM incurs a minimum of one mutation to the heap, the slowdown is less than a factor 10. This makes it usable in applications that require immediate memory reclamation or a predictable GC behaviour.

5.3 Profiling the Overhead

This section breaks down the overhead in the Arborescent GC by profiling in which collection phase (`drop`, `catch`, `collect` or `adopt`) the execution of each benchmark is spent.

Since Arborescent garbage collection frequently executes short collections, profiling time spent in each phase would be noisy due to the significant overhead of the timer itself. Consequently, CPU cycles are measured as a proxy for time spent in each phase. Profiling CPU cycles is achieved by calling the `rdtsc` x86 instruction, which returns the processor's time-stamp counter [9]. The time-stamp is read at the start and end of each phase and the difference is taken to obtain the CPU cycles spent in that phase. The total CPU cycles spent in each benchmark is also measured.

This profiling is first applied to benchmarks with Mark-and-Sweep to recover the proportion of CPU cycles spent in the mutator and the collector. Total CPU cycles of the mutator correspond to CPU cycles spent to execute the benchmark barring any memory management. This count is used as the threshold above which cycles are considered memory management overhead in both implementations.

The same profiling is done with the Arborescent GC to profile total CPU cycles and cycles spent in the `drop` (excludes `adopt`), `catch`, `collect`, and `adopt` (includes `rerank`) phases.

Figure 11 presents the memory management overhead of both garbage collectors (the coloured part of each bar correspond to overhead). In the case of the Arborescent GC, the `drop`, `catch`, `collect` and `adopt` phases do not account for the whole overhead. The difference, which accounts for about 30% of the overhead on average, (pink and labelled *other*) corresponds to the overhead of managing the lists of coparents outside `drop`, `catch`, `collect` and `adopt`, for instance when adding a reference or removing a reference that is not a parent-child relation.

Another significant portion of cycles (about 25% of the overhead) is spent in the `collect` phase (green). This is because the `collect` phase needs to remove all coparent-cochild relations when an object is deallocated.

Hence, a large part of the overhead of the Arborescent collector comes from managing coparents, either in the `collect` phase or when mutating an object in a way that does not affect reachability. This stems from the need to traverse an

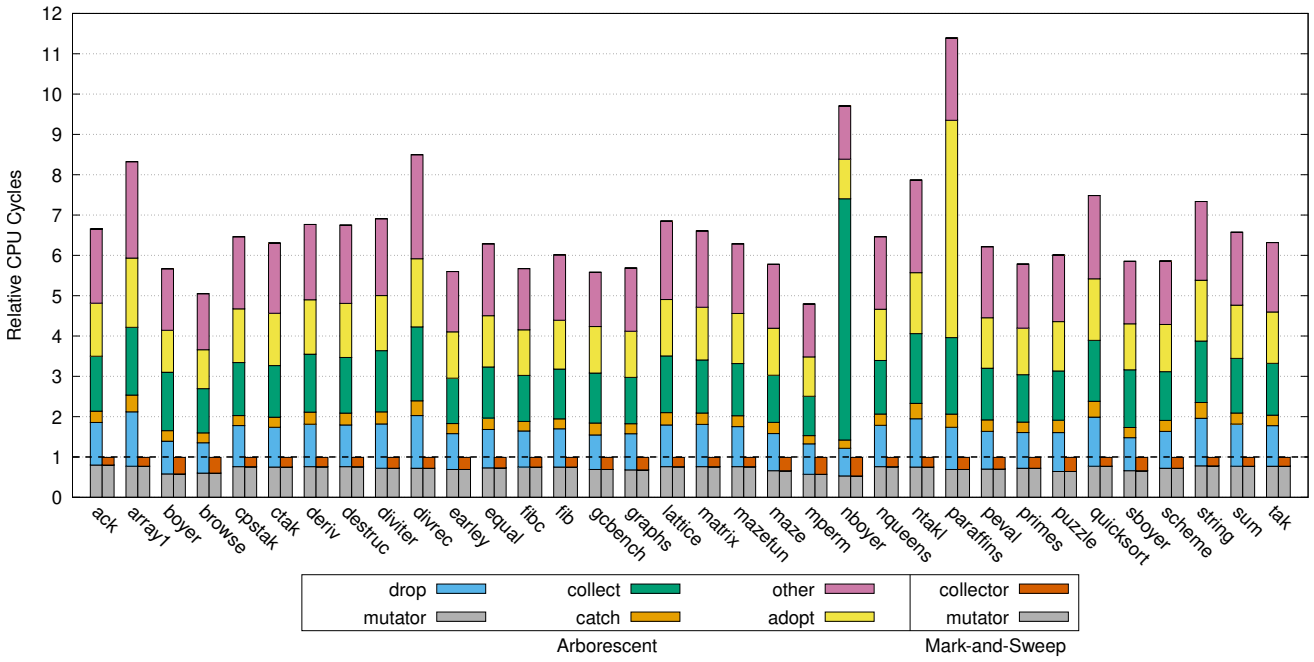


Figure 11. CPU cycles relative to the total CPU cycles count of Mark-and-Sweep measured on R7RS benchmarks. Each benchmark has two bars that correspond to the profiling of each GC. The left bar shows the proportion of CPU cycles spent in each phase of the Arborescent collector (from bottom to top: mutator, `drop`, `catch`, `collect`, `adopt`, and other). The right bar shows the proportion of CPU cycles spent in the mutator (bottom segment) and collector (top segment) of Mark-and-Sweep. The bottom segment (grey) of each bar corresponds to CPU cycles spent in the mutator, everything above is the overhead of the corresponding GC.

object's linked-list of referrers when removing one of its coparent. While this list is usually small, this removal operation is ubiquitous, making it a good optimization candidate to improve performance.

The `drop` and `adopt` phases also accounts for a significant part of the overhead (both about 20% on average). The `catch` phase accounts for a considerably smaller part of the overhead (about 5%).

Two outliers stand out in Figure 11, `nboyer` and `paraffins`, providing an opportunity to discuss possible improvements to the Arborescent GC.

The execution of `nboyer` is dominated by the `collect` phase. This is presumed to be largely caused by `nboyer` allocating many short-lived objects with children/cochildren that have many referrers (even with Mark-and-Sweep, about 45% of `nboyer`'s execution is spent in the collector). These objects must be removed from their children/cochildren's list of referrers when deallocated, which is a linear operation. Otherwise, these lists of referrers would contain dangling pointers.

The second outlier, `paraffins`, spends a larger portion of execution in the `adopt` phase. To reduce the overhead of adoption, objects' referrers could be kept in order of rank, either by using a priority queue or keeping the linked-list sorted. This could accelerate adoption (only the first coparent would have to be considered in Algorithm 5), but at the risk of introducing further overhead when managing referrers. Furthermore, `paraffins` suggests that, while the simple reranking heuristic used in this implementation works well in many cases, there are programs that could benefit from a fine-tuned heuristic, such as using runtime type information or program analysis to guide the decision of which objects should be explored by `rerank`.

6 Limitations and Future Work

This section discusses some limitations of the algorithm and its implementation, as well as possible future directions to address them.

6.1 Multithreading Support

The Arborescent garbage collector presented in this paper assumes that a program's execution is entirely single-threaded. Recent developments in adapting similar graph data structures to their concurrent equivalents, particularly in the context of dynamic connectivity [11], suggest that adapting the algorithm to support multithreaded programs is feasible although the additional cost of doing so in the context of garbage collection remains unclear.

6.2 Optimizations

The results shown in Figure 10 present performance baseline, as the implementation closely mirrors the algorithm described in Section 3 for tracking object reachability. This leaves room for performance improvements through further optimization.

For instance, in most programming languages, some objects are known to never contain cycles. Examples in Scheme include booleans, numbers, numeric vectors, and symbols. This allows using runtime type information and static analysis to detect some acyclic objects. For these, a reference count is sufficient to track reachability. Similarly, immutable data might deserve some special treatment since any cycle needs to go through at least one mutable object. This points toward a hybrid implementation of the algorithm where some objects have a reference count instead of a full set of referrers.

Because the Arborescent garbage collector tracks more information than a reference counting collector, it is always possible to compute the reference count of an object by counting its referrers. As such, reference counting optimizations such as coalescing [15], reuse of unreachable objects [24, 28] and subsumption [13] should also be applicable in this context. Even deferred collection [2] should be applicable, although it would defeat the immediacy benefit.

As discussed in Section 5, the choice of reranking heuristic (Algorithm 6) has a significant impact on performance. This is because, along with the adoption heuristic (Algorithm 5), it allows for the edge removal procedure to avoid the worst-case scenario where a large portion of the heap must be scanned to reconnect two spanning trees. This paper presented a relatively simple reranking heuristic, showing that decent performance is attainable without excessive fine-tuning. In practice, fine-tuning heuristics to the semantics of a programming language may be beneficial. Moreover, the adoption heuristic presented in this paper always chooses the first found valid adopter, which might not be optimal. Drawing inspiration from generational garbage collection [16], a potentially better heuristic may attempt to find a long-lived adopter, which is less likely to be itself deallocated.

6.3 Finalizers and Interoperability with C++ and Rust

The Arborescent GC could be extended with finalizers that get called immediately when an object becomes unreachable.

Doing so for languages with automatic memory management can introduce delicate issues linked to a lack of clarity around the semantics of reachability, which is often not fully specified by the language. This can leave execution of finalizers unpredictable even with the algorithm presented in this paper, since it can depend on optimization choices in the compiler. See [4] for an in-depth discussion of the problems linked to *when* a finalizer can be called and *what* is allowed to run inside of it. Research that clarifies these semantics could leverage the potential of immediate finalizers such as the instant release of resources (file handles, network sockets, and locks).

For languages with semi-managed memory, the Arborescent garbage collector's perfect information about objects' reachability could enable effective interoperability with their memory management system. For example, C++ and Rust clearly define when destructors (or finalizers) are called: often at the end of scopes for C++, or when their lifetime ends for Rust. At those points, the Arborescent GC knows which objects are unreachable, allowing for their reclamation and finalization by the garbage collector. This could enable the creation of libraries like the `Rc` type for Rust or smart pointers for C++ that can seamlessly handle cycles. Here again, some delicate issues of semantics might need to be addressed, such as ordering of finalizers within cycles.

7 Conclusion

This paper described a reachability algorithm for garbage collection that supports synchronous memory reclamation, including cyclic structures. At its core, the algorithm inscribes a spanning forest in the reference graph of a program and explores the reference graph to repair the spanning forest whenever an edge is deleted. This approach is not new, but previous implementations have proven to be prohibitively expensive unless cycle reclamation is deferred to an asynchronous task. This issue is fixed by introducing a weak notion of rank in the reference graph. This additional information allows a more efficient exploration of the graph on an edge deletion.

The resulting garbage collector was implemented in the Ribbit Scheme compiler. While still slower than other state-of-the-art tracing garbage collectors, it offers decent performance without the need to defer cycle collection to an asynchronous process, making it suitable for applications that require a reliable, deterministic behaviour.

Acknowledgment

This work was supported by the *Natural Sciences and Engineering Research Council of Canada* and the *Fonds de recherche du Québec*[3].

References

[1] 2025. R7RS Benchmarks. https://github.com/ecraven/r7rs-benchmarks.

[2] David F. Bacon and V. T. Rajan. 2001. Concurrent Cycle Collection in Reference Counted Systems. In *ECOOP 2001 - Object-Oriented Programming, 15th European Conference, Budapest, Hungary, June 18-22, 2001, Proceedings (Lecture Notes in Computer Science, Vol. 2072)*, Jørgen Lindskov Knudsen (Ed.). Springer, 207–235. doi:10.1007/3-540-45337-7_12

[3] Joshua Bloch. 2008. *Creating and Destroying Objects* (2nd ed.). Addison-Wesley Professional, Chapter 2.

[3] https://doi.org/10.69777/366699 & https://doi.org/10.69777/352044

[4] Hans-J. Boehm. 2003. Destructors, finalizers, and synchronization. *SIGPLAN Not.* 38, 1 (Jan. 2003), 262–272. doi:10.1145/640128.604153

[5] Steven R. Brandt, Hari Krishnan, Costas Busch, and Gokarna Sharma. 2018. Distributed garbage collection for general graphs. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on Memory Management, ISMM 2018, Philadelphia, PA, USA, June 18, 2018*, Hannes Payer and Jennifer B. Sartor (Eds.). ACM, 29–44. doi:10.1145/3210563.3210572

[6] Steven R. Brandt, Hari Krishnan, Gokarna Sharma, and Costas Busch. 2014. Concurrent, parallel garbage collection in linear time. In *International Symposium on Memory Management, ISMM '14, Edinburgh, United Kingdom, June 12, 2014*, David Grove and Samuel Z. Guyer (Eds.). ACM, 47–58. doi:10.1145/2602988.2602990

[7] David R. Brownbridge. 1985. Cyclic Reference Counting for Combinator Machines. In *Functional Programming Languages and Computer Architecture, FPCA 1985, Nancy, France, September 16-19, 1985, Proceedings (Lecture Notes in Computer Science, Vol. 201)*, Jean-Pierre Jouannaud (Ed.). Springer, 273–288. doi:10.1007/3-540-15975-4_42

[8] George E. Collins. 1960. A method for overlapping and erasure of lists. *Commun. ACM* 3, 12 (Dec. 1960), 655–657. doi:10.1145/367487.367501

[9] Intel Corporation. 2025. *Intel® 64 and IA-32 Architectures Software Developer's Manual - Volume 3.* Chapter 19.17: Time-Stamp Counter, 19–42 to 19–44. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

[10] Shimon Even and Yossi Shiloach. 1981. An On-Line Edge-Deletion Problem. *J. ACM* 28, 1 (1981), 1–4. doi:10.1145/322234.322235

[11] Alexander Fedorov, Nikita Koval, and Dan Alistarh. 2021. A Scalable Concurrent Algorithm for Dynamic Connectivity. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures* (Virtual Event, USA) *(SPAA '21)*. Association for Computing Machinery, New York, NY, USA, 208–220. doi:10.1145/3409964.3461810

[12] Saverio Giallorenzo and Francesco Goretti. 2025. Breadth-first Cycle Collection Reference Counting: Theory and a Rust Smart Pointer Implementation. In *ACM digital library*. Catania (IT), Italy. doi:10.1145/3672608.3707785

[13] Pramod G. Joisha. 2006. Compiler optimizations for nondeferred reference: counting garbage collection. In *Proceedings of the 5th International Symposium on Memory Management* (Ottawa, Ontario, Canada) *(ISMM '06)*. Association for Computing Machinery, New York, NY, USA, 150–161. doi:10.1145/1133956.1133976

[14] Jaehwang Jung, Jeonghyeon Kim, Matthew J. Parkinson, and Jeehoon Kang. 2024. Concurrent Immediate Reference Counting. *Proc. ACM Program. Lang.* 8, PLDI, Article 153 (June 2024), 24 pages. doi:10.1145/3656383

[15] Yossi Levanoni and Erez Petrank. 2006. An on-the-fly reference-counting garbage collector for java. *ACM Trans. Program. Lang. Syst.* 28, 1 (Jan. 2006), 1–69. doi:10.1145/1111596.1111597

[16] Henry Lieberman and Carl Hewitt. 1983. A real-time garbage collector based on the lifetimes of objects. *Commun. ACM* 26, 6 (June 1983), 419–429. doi:10.1145/358141.358147

[17] Chin-Yang Lin and Ting-Wei Hou. 2007. A simple and efficient algorithm for cycle collection. *ACM SIGPLAN Notices* 42, 3 (2007), 7–13. doi:10.1145/1273039.1273041

[18] Rafael Dueire Lins. 1992. Cyclic Reference Counting with Lazy Mark-Scan. *Inf. Process. Lett.* 44, 4 (1992), 215–220. doi:10.1016/0020-0190(92)90088-D

[19] Léonard Oest O'Leary and Marc Feeley. 2023. A Compact and Extensible Portable Scheme VM. In *Companion Proceedings of the 7th International Conference on the Art, Science, and Engineering of Programming* (Tokyo, Japan) *(Programming '23)*. Association for Computing Machinery, New York, NY, USA, 3–6. doi:10.1145/3594671.3594672

[20] Léonard Oest O'Leary, Mathis Laroche, and Marc Feeley. 2023. A R4RS Compliant REPL in 7 KB. *CoRR* abs/2310.13589 (2023). doi:10.48550/ARXIV.2310.13589

[21] Oracle Corporation. 2025. Object (Java SE 17 & JDK 17). https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Object.html#finalize()

[22] José M. Piquer. 1991. Indirect Reference Counting: A Distributed Garbage Collection Algorithm. In *PARLE '91: Parallel Architectures and Languages Europe, Volume I: Parallel Architectures and Algorithms, Eindhoven, The Netherlands, June 10-13, 1991, Proceedings (Lecture Notes in Computer Science, Vol. 505)*, Emile H. L. Aarts, Jan van Leeuwen, and Martin Rem (Eds.). Springer, 150–165. doi:10.1007/BFB0035102

[23] Python Software Foundation. 2025. Data Model - Python 3.13. https://docs.python.org/3/reference/datamodel.html#object.__del__

[24] Alex Reinking, Ningning Xie, Leonardo de Moura, and Daan Leijen. 2021. Perceus: garbage free reference counting with reuse. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 96–111. doi:10.1145/3453483.3454032

[25] Ruby Core Team. 2025. module ObjectSpace - Documentation for Ruby 3.5. https://docs.ruby-lang.org/en/master/ObjectSpace.html#method-c-define_finalizer

[26] Michael Schöttner, Ralph Göckelmann, Stefan Frenz, Markus Fakler, and Peter Schulthess. 2006. Incremental Distributed Garbage Collection Using Reverse Reference Tracking. In *Euro-Par 2006, Parallel Processing, 12th International Euro-Par Conference, Dresden, Germany, August 28 - September 1, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 4128)*, Wolfgang E. Nagel, Wolfgang V. Walter, and Wolfgang Lehner (Eds.). Springer, 571–581. doi:10.1007/11823285_59

[27] Matthew Sotoudeh. 2025. Pathological Cases for a Class of Reachability-Based Garbage Collectors. *Proc. ACM Program. Lang.* 9, OOPSLA1, Article 96 (April 2025), 28 pages. doi:10.1145/3720430

[28] Sebastian Ullrich and Leonardo de Moura. 2021. Counting immutable beans: reference counting optimized for purely functional programming. In *Proceedings of the 31st Symposium on Implementation and Application of Functional Languages* (Singapore, Singapore) *(IFL '19)*. Association for Computing Machinery, New York, NY, USA, Article 3, 12 pages. doi:10.1145/3412932.3412935

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