Concurrent, Parallel Garbage Collection in Linear Time

Steven R. Brandt
Center for Computation and Technology, School of Electrical Engineering and Computer Science, Louisiana State University, Baton Rouge, LA 70803, USA
sbrandt@cct.lsu.edu

Hari Krishnan
Center for Computation and Technology, School of Electrical Engineering and Computer Science, Louisiana State University, Baton Rouge, LA 70803, USA
hkrish4@tigers.lsu.edu

Gokarna Sharma
School of Electrical Engineering and Computer Science, Louisiana State University, Baton Rouge, LA 70803, USA
gokarna@csc.lsu.edu

Costas Busch
School of Electrical Engineering and Computer Science, Louisiana State University, Baton Rouge, LA 70803, USA
busch@csc.lsu.edu

Abstract

This paper presents a new concurrent garbage collection algorithm based on two types of reference, *strong* and *weak*, to link the graph of objects. Strong references connect the roots to all the nodes in the graph but do not contain cycles. Weak references may, however, contain cycles.

Advantages of this system include: (1) reduced processing, non-trivial garbage collection work is only required when the last strong reference is lost; (2) fewer memory traces to delete objects, a garbage cycle only needs to be traversed twice to be deleted; (3) fewer memory traces to retain objects, since the collector can often prove objects are reachable without fully tracing support cycles to which the objects belong; (4) concurrency, it can run in parallel with a live system without "stopping the world;" (5) parallel, because collection operations in different parts of the memory can proceed at the same time.

Previous variants of this technique required exponential cleanup time [27, 31], but our algorithm is linear in total time, i.e. any changes in the graph take only $\mathcal{O}(N)$ time steps, where N is the number of edges in the affected subgraph (e.g. the subgraph whose strong support is affected by the operations).

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors–Memory management (garbage collection)

General Terms Algorithms, Performance, Experimentation, Languages, Design, Termination

Keywords Garbage Collection, Parallel programming theory and models, Compilers and runtime systems, Software for productivity parallel programming, Parallel algorithms, Concurrent data structures

1. Introduction

Garbage collection is an important productivity feature in many languages, eliminating a thorny set of coding errors which can be created by explicit memory management [2, 3, 5, 10, 15, 23, 24]. Despite the many advances in garbage collection, there are problem areas which have difficulty benefiting, such as distributed or real-time systems; see [4, 28, 33]. Even in more mundane settings, a certain liveness in the cleanup of objects may be required by an application for which the garbage collector provides little help, e.g. managing a small pool of database connections.

The garbage collection system we propose is based on a scheme originally proposed by Brownbridge [8]. Brownbridge proposed the use of two types of pointers: *strong* and *weak*[1]. Strong pointers are required to connect from the *roots* (i.e. references from the stack or global memory) to all nodes in the graph, and contain no cycles. A path of strong links (i.e. pointers) from the roots guarantees that an object should be in memory. Weak pointers are available to close cycles and provide the ability to connect nodes in arbitrary ways.

Brownbridge's proposal was vulnerable to premature collection [31], and subsequent attempts to improve it introduced poor performance (at least exponential cleanup time in the worst-case) [27]; details in Section 1.1. Brownbridge's core idea, however, of using two types of reference counts was sound: maintaining this pair of reference counts allows the system to remember a set of acyclic paths through memory so that the system can minimize collection operations. For example, Roy et al. [30] used Brownbridge's idea to optimize scanning in databases. In this work we will show how the above problems with the Brownbridge collection scheme can be repaired by the inclusion of a third type of counter, which we call a *phantom count*. This modified system has a number of advantages.

Typical hybrid reference count and collection systems, e.g. [2, 3, 5, 19, 22], which use a reference counting collector combined with a tracing collector or cycle collector, must perform nontrivial work whenever a reference count is decreased and does not reach zero. The modified Brownbridge system, with three types of reference counts, must perform nontrivial work only when the strong reference count reaches zero and the weak reference count is still positive, a significant reduction [8, 15].

Many garbage collectors in the literature employ a generational technique, for example the generational collector proposed in [29], taking advantage of the observation that younger objects are more likely to need garbage collecting than older objects. Because the strong/weak reference system tends to make the links in long-lived paths through memory strong, old objects connected to these paths are unlikely to become the focus of the garbage collector's work.

In other conceptions of weak pointers, if a node is reachable by a strong pointer and a weak pointer and the strong pointer is removed,

[1] These references are unrelated to the weak, soft, and phantom reference objects available in Java under Reference class.

ISMM'14, June 12, 2014, Edinburgh, United Kingdom.

http://dx.doi.org/10.1145/2602988.2602990

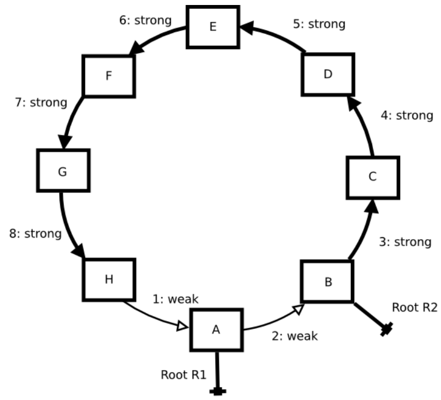

Figure 1. When root R1 is removed, the garbage collection algorithm only needs to trace link 2 to prove that object A does not need to be collected.

the node is garbage, and the node is deleted. In Brownbridge's conception of weak pointers, the weak pointer is turned into a strong pointer in an intelligent way.

In many situations, the collector can update the pointer types and prove an object is still live without tracing through memory cycles the object may participate in. For example, when root $R1$ is removed from the graph shown in Fig. 1, the collector only needs to convert link 1 to strong and examine link 2 to prove that object A does not need to be collected. What's more, because the strength of the pointer depends on a combination of states in the source and target, the transformation from strong to weak can be carried out without the need for back-pointers (i.e. links that can be followed from target to source as well as source to target).

This combination of effects, added to the fact that our collector can run in parallel with a live system, may prove useful in soft real-time systems, and may make the use of "finalizers" on objects more practical. These kinds of advantages should also be significant in a distributed setting, where traversal of links in the collector is a more expensive operation. Moreover, when a collection operation is local, it has the opportunity to remain local, and cleanup can proceed without any kind of global synchronization operation.

The contribution of this paper is threefold. Our algorithm never prematurely deletes any reachable object, operates in linear in time regardless of the structure (any deletion of edges and roots takes only $\mathcal{O}(N)$ time steps, where N is the number of edges in the affected subgraph), and works concurrently with the application, without the need for system-wide pauses, i.e. "stopping the world." This is in contrast to Brownbridge's original algorithm and its variants [8, 27, 31] which can not handle concurrency issues that arise in modern multiprocessor architectures [30].

Our algorithm does, however, add a third kind of pointer, a *phantom pointer*, which identifies a temporary state that is neither strong nor weak. The addition of the space overhead offsets the time overhead in Brownbridge's work.

1.1 Most Related Work: Premature Collection, Non-termination, and Exponential Cleanup Time

Before giving details of our algorithm in Section 2 – which in turn is a modification of Brownbridge's algorithm and its variants [8, 27, 31] – we first describe the problems with previous variants. In doing so, we follow the description given in the excellent garbage collection book due to Jones and Lins [15].

It was proven by McBeth [23] in early sixties that reference counting collectors were unable to handle cyclic structures; several attempts to fix this problem appeared subsequently, e.g. [7, 12, 22]. We give details in Section 1.2. In contrast to the approach followed in [7, 12, 22] and several others, Brownbridge [8] proposed, in 1985, a strong/weak pointer algorithm to tackle the problem of reclaiming cyclic data structures by distinguishing cycle closing pointers (weak pointers) from other references (strong pointers) [15]. This algorithm relied on maintaining two invariants: (a) there are no cycles in strong pointers and (b) all items in the graph must be strongly reachable from the roots.

Some years after the publication, Salkild [31] showed that Brownbridge's algorithm [8] could reclaim objects prematurely in some configurations, e.g. a double cycle. If the last strong pointer (or link) to an object in one cycle but not the other was lost, Brownbridge's method would incorrectly claim nodes from the cycle. Salkild [31] corrected this problem by proposing that if the last strong link was removed from an object which still had weak pointers, a collection process should re-start from that node. While this approach eliminated the premature deletion problem, it introduced a potential non-termination problem.

Subsequently, Pepels et al. [27] proposed a new algorithm based on Brownbridge-Salkild's algorithm and solved the problem of non-termination by using a marking scheme. In their algorithm, they used two kinds of mark: one to prevent an infinite number of searches, and the other to guarantee termination of each search. Although correct and terminating, Pepels et al.'s algorithm is far more complex than Brownbridge-Salkild's algorithm and in some cyclic structures the cleanup cost complexity becomes at least exponential in the worst-case [15]. This is due to the fact that when cycles occur, whole state space searches from each node in the cyclic graph must be initiated, possibly many times. After Pepels et al.'s algorithm, we are not aware of any other work on reducing the cleanup cost or complexity of the Brownbridge algorithm. Moreover, there is no concurrent collection technique using this approach which can be applicable for the garbage collection in modern multiprocessors.

The algorithm we present in this paper removes all the limitations described above. Our algorithm does not perform searches as such. Instead, whenever a node loses its last strong reference and still has weak references, it marks all affected links as phantom. When this process is complete for a subgraph, the system recovers the affected subgraph by converting phantom links to either strong or weak. Because this process is a transformation from weak or strong to phantom, and from phantom to weak or strong, it has at most two steps and is, therefore, manifestly linear in the number of links, i.e. it has a complexity of only $\mathcal{O}(N)$ time steps, where N is the number of edges in the affected subgraph. Moreover, in contrast to Brownbridge's algorithm, our algorithm is concurrent and is suitable for multiprocessors.

1.2 Other Related Work

Garbage collection is an automatic memory management technique which is considered to be an important tool for developing fast as well as reliable software. Garbage collection has been studied extensively in computer science for more than five decades, e.g., [2, 3, 5, 8, 15, 23, 27, 31]. Reference counting is a widely-used form of garbage collection whereby each object has a count of the number of references to it; garbage is identified by having a reference count of zero [2]. Reference counting approaches were first developed for LISP by Collins [10]. Improved variations were proposed in several subsequent papers, e.g. [12, 14, 15, 19, 22]. We direct readers to Shahriyar et al. [32] for the valuable overview of the current state of reference counting collectors.

It was noticed by McBeth [23] in early sixties that reference counting collectors were unable to handle cyclic structures. After that several reference counting collectors were developed, e.g. [7, 12, 20, 21]. The algorithm in Friedman [12] dealt with recovering cyclic data in immutable structures, whereas Bobrow's algorithm [7] can reclaim all cyclic structures but relies on the explicit information provided by the programmer. Trial deletion approach was studied by Christopher [9] which tries to collect cycles by identifying groups of self-sustaining objects. Lins [20] used a cyclic buffer to reduce repeated scanning of the same nodes in their mark-scan algorithm for cyclic reference counting. Moreover, in [21], Lins improved his algorithm from [20] by eliminating the scan operation through the use of a Jump-stack data structure.

With the advancement of multiprocessor architectures, reference counting garbage collectors have become popular because they do not require all application threads to be stopped before the garbage collection algorithm can run [19]. Recent work in reference counting algorithms, e.g. [2, 3, 5, 19], try to reduce concurrent operations and increase the efficiency of reference counting collectors. Since our collector is a reference counting collector, it can potentially benefit from the same types of optimizations discussed here. We leave that, however, to a future work.

However, as mentioned earlier, reference counting garbage collectors cannot collect cycles [23]. Therefore, concurrent reference counting collectors [2, 3, 5, 19, 22, 26] use other techniques, e.g. they supplement the reference counter with a tracing collector or a cycle detector, together with their concurrent reference counting algorithm. For example, the reference counting collector proposed in [26] combines the sliding view reference counting concurrent collector of [19] with the cycle collector of [2]. Our collector has some similarity with these, in that our $Phantomization$ process may traverse many nodes. It should, however, trace fewer nodes and do so less frequently. Recently, Frampton provides a detailed study of cycle collection in his PhD thesis [11].

Herein we have tried to cover a sampling of garbage collectors that are most relevant to our work.

Apple's ARC memory management system makes a distinction between "strong" and "weak" pointers, similar to what we describe here. In the ARC memory system, however, the type of each pointer must be specifically designated by the programmer, and this type will not change during the program's execution. If the programmer gets the type wrong, it is possible for ARC to have strong cycles as well as prematurely deleted objects. With our system, the pointer type is automatic and can change during the execution. Our system protects against these possibilities, at the cost of lower efficiency.

There exist other concurrent techniques optimized for both uniprocessors as well as multiprocessors. Generational concurrent garbage collectors were also studied, e.g. [29]. Huelsbergen and Winterbottom [13] proposed an incremental algorithm for the concurrent garbage collection that is a variant of mark-and-sweep collection scheme first proposed in [24]. Furthermore, garbage collection is also considered for several other systems, namely real-time systems and asynchronous distributed systems, e.g. [28, 33].

Concurrent collectors are gaining popularity. The concurrent collector described in Bacon and Rajan [2] can be considered to be one of the more efficient reference counting concurrent collectors. The algorithm uses two counters per object, one for the actual reference count and other for the cyclic reference count. Apart from the number of the counters used, the cycle detection strategy requires a minimum of two traversals of cycle when the cycle is reachable and eleven cycle traversals when the cycle is garbage.

1.3 Paper Organization

The rest of the paper is organized as follows. We present our strong/weak/phantom pointer based concurrent garbage collector in Section 2 with some examples. In Section 4, we sketch proofs of its correctness and complexity properties. In Section 5, we give some experimental results. We conclude the paper with future research directions in Section 6 and a short discussion in Section 7. Detailed algorithms may be found in the appendix.

2. Algorithm

In this section, we present our concurrent garbage collection algorithm. Each object in the heap contains three reference counts: the first two are the strong and weak, the third is the phantom count. Each object also contains a bit named `which` (Brownbridge [8] called it the "strength-bit") to identify which of the first two counters is used to keep track of strong references, as well as a boolean called `phantomized` to keep track of whether the node is phantomized. Outgoing links (i.e., pointers) to other objects must also contain (1) a `which` bit to identify which reference counter on the target object they increment, and (2) a `phantom` boolean to identify whether they have been phantomized. This data structure for each object can be seen in the example given in Fig. 2.

Local creation of links only allows the creation of strong references when no cycle creation is possible. Consider the creation of a link from a source object S to a target object T. The link will be created strong if (i) the only strong links to S are from roots i.e. there is no object C with a strong link to S; (ii) object T has no outgoing links i.e. it is newly created and its outgoing links are not initialized; and (iii) object T is phantomized, and S is not. All self-references are weak. Any other link is created phantom or weak.

To create a strong link, the `which` bit on the link must match the value of the `which` bit on the target object. A weak link is created by setting the `which` bit on the reference to the complement of the value of the `which` bit on the target.

When the strong reference count on any object reaches zero, the garbage collection process begins. If the object's weak reference count is zero, the object is immediately reclaimed. If the weak count is positive, then a a sequence of three phases is initiated: $Phantomization$, $Recovery$, and $CleanUp$. In $Phantomization$, the object toggles its `which` bit, turning its incoming weak reference counts to strong ones, and phantomizes its outgoing links.

Phantomizing a link transfers a reference count (either strong or weak), to the phantom count on the target object. If this causes the object to lose its last strong reference, then the object may also phantomize, i.e. toggle its `which` bit (if that will cause it to gain strong references), and phantomizes all its outgoing links. This process may spread to a large number of target objects.

All objects touched in the process of a phantomization that were able to recover their strong references by toggling their `which` bit are remembered and put in a "recovery list". When phantomization is finished, $Recovery$ begins, starting with all objects in the recovery list.

To perform a recovery, the system looks at each object in the recovery list, checking to see whether it still has a positive strong reference count. If it does, it sets the `phantomized` boolean to false, and rebuilds its outgoing links, turning phantoms to strong or weak according to the rules above. If a phantom link is rebuilt and the target object regains its first strong reference as a result, the target object sets its `phantomized` boolean to false and attempts to recover its outgoing phantom links (if any). The recovery continues to rebuild outgoing links until it terminates.

Finally, after the recovery is complete, $CleanUp$ begins. The recovery list is revisited a second time. Any objects that still have no strong references are deleted.

Note that all three of these phases, $Phantomization$, $Recovery$, and $CleanUp$ are, by their definitions, linear in the number of links; we prove this formally in Theorem 4.2 in Section 4. Links can undergo only one state change in each of these phases: strong or weak to phantom during $Phantomization$, phantom to strong or weak during $Recovery$, and phantom to deleted in $CleanUp$.

We now present some examples to show how our algorithm performs collection in several real word scenarios.

2.1 Example: A Simple Cycle

In Fig. 2 we see a cyclic graph with three nodes. This figure shows the counters, bits, and boolean values in full detail to make it clear how these values are used within the algorithm. Objects are represented with circles, links have a pentagon with state information at their start and an arrow at their end.

In Step 0, the cycle is supported by a root, a reference from stack or global space. In Step 1, the root reference is removed, decrementing the strong reference by one, and beginning a $Phantomization$. Object C toggles its which pointer and phantomizes its outgoing links. Note that toggling the which pointer causes the link from A to C to become strong, but nothing needs to change on A to make this happen.

In Step 2, object B also toggles its which bit, and phantomizes its outgoing links. Likewise, in Step 3, object A phantomizes, and the $Phantomization$ phase completes.

$Recovery$ will attempt to unphantomize objects A, B, and C. None of them, however, have any strong support, and so none of them recover.

$Cleanup$ happens next, and all objects are reclaimed.

2.2 Example: A Doubly-Linked List

The doubly linked list depicted in Fig. 3 is a classic example for garbage collection systems. The structure consists of 6 links, and the collector marks all the links as phantoms in 8 steps.

This figure contains much less detail than Fig. 2, which is necessary for a more complex figure.

2.3 Example: Rebalancing A Doubly-Linked List

Fig. 4 represents a worst case scenario for our algorithm. As a result of losing root $R1$, the strong links are pointing in exactly the wrong direction to provide support across an entire chain of double links. During $Phantomization$, each of the objects in the list must convert its links to phantoms, but nothing is deleted. $Phantomization$ is complete in the third figure from the left, and $Recovery$ begins. The fourth step in the figure, when link 6 is converted from phantom to weak marks the first phase of the recovery.

2.4 Example: Recovering Without Detecting a Cycle

In Fig. 1 we see the situation where the collector recovers from the loss of a strong link without searching the entire cycle. When root $R1$ is removed, node A becomes phantomized. It turns its incoming link (link 1) to strong, and phantomizes its outgoing link (link 2), but then the phantomization process ends. Recovery is successful, because A has strong support, and it rebuilds its outgoing link as weak. At this point, collection operations are finished.

Unlike the doubly-linked list example above, this case describes an optimal situation for our garbage collection system.

3. Concurrency Issues

This section provides details of the implementation.

3.1 The Single-Threaded Collector

There are several methods by which the collector may be allowed to interact concurrently with a live system. The first, and most straightforward implementation, is to use a single garbage collection thread to manage nontrivial collection operations. This technique has the advantage of limiting the amount of computational power the garbage collector may use to perform its work.

For the collection process to work, phantomization must run to completion before recovery is attempted, and recovery must run to completion before cleanup can occur. To preserve this ordering in a live system, whenever an operation would remove the last strong link to an object with weak or phantom references, the link is instead transferred to the collector, enabling it to perform phantomization at an appropriate time.

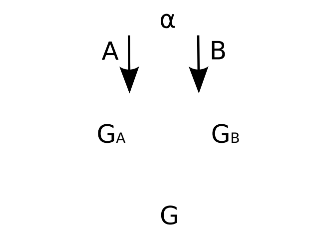

Figure 5. Graph model

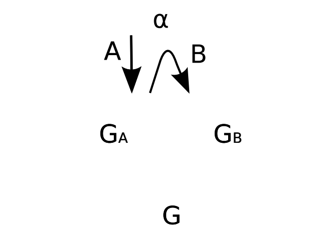

Figure 6. Subgraph model

After the strong link is processed, the garbage collector needs to create a phantom link to hold onto the object while it performs its processing, to ensure the collector itself doesn't try to use a deleted object.

Another point of synchronization is the creation of new links. If the source of the link is a phantomized node, the link is created in the phantomized state.

With these relatively straightforward changes, the single-threaded garbage collector may interact freely with a live system.

3.2 The Multi-Threaded Collector

The second, and more difficult method, is to allow the collector to use multiple threads. In this method, independent collector threads can start and run in disjoint areas of memory. In order to prevent conflicts from their interaction, we use a simple technique: whenever a link connecting two collector threads is phantomized, or when a phantom link is created by the live system connecting subgraphs under analysis by different collector threads, the threads merge. A merge is accomplished by one thread transferring its remaining work to the other and exiting. To make this possible, each object needs to carry a reference to the collection threads and ensure that this reference is removed when collection operations are complete. While the addition of a pointer may appear to be a significant increase in memory overhead, it should be noted that the pointer need not point directly to the collector, but to an intermediate object which can carry the phantom counter, as well as other information if desired.

An implementation of this parallelization strategy is given in pseudocode in the appendix.

4. Correctness and Algorithm Complexity

The garbage collection problem can be modeled as a directed graph problem in which the graph has a special set of edges (i.e. links) called $roots$ that come from nowhere. These edges determine if a node in the graph is reachable or not. A node X is said to be $reachable$ if there is a path from any root to a node X directly or transitively. Thus, the garbage collection problem can be described as removing all nodes in the graph that are not reachable from any $roots$.

Our algorithm uses three phases to perform garbage collection. The three phases are $Phantomize$, $Recover$ and $CleanUp$. The $Phantomization$ phase is a kind of search that marks (i.e. phantomizes) nodes which have lost strong support. The $Recovery$ phase unmarks the nodes, reconnecting the affected subgraph to strong links. If $Recovery$ fails to rebuild links, the $CleanUp$ phase deletes them. The algorithm progresses through all three phases in the order (1. $Phantomize$, 2. $Recover$ and 3. $CleanUp$) and transitions only when there are no more operations left in the current phase. Our algorithm is concurrent because the garbage collection

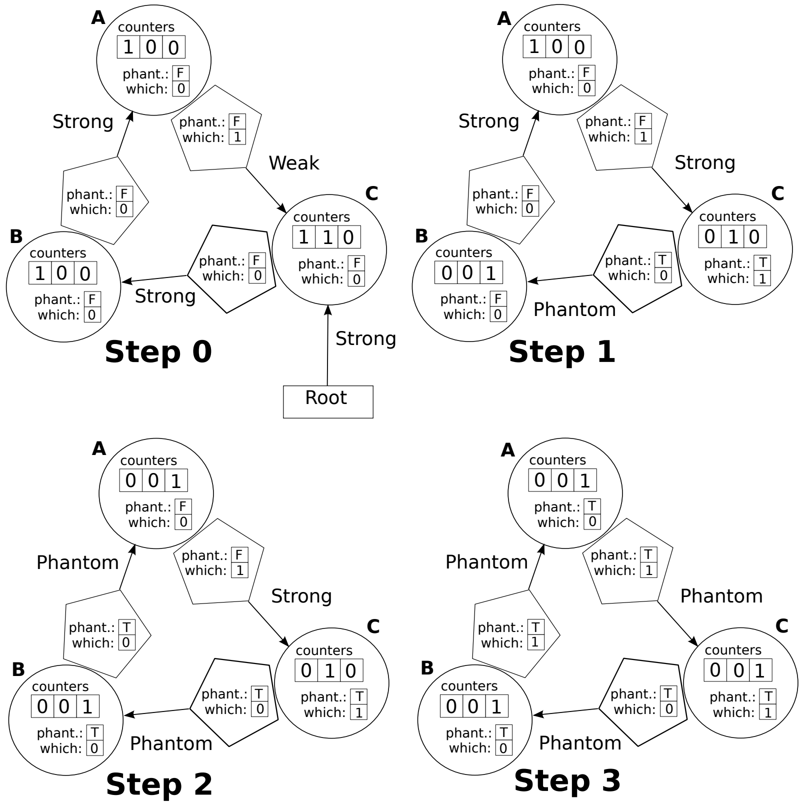

Figure 2. Reclaiming a cycle with three objects

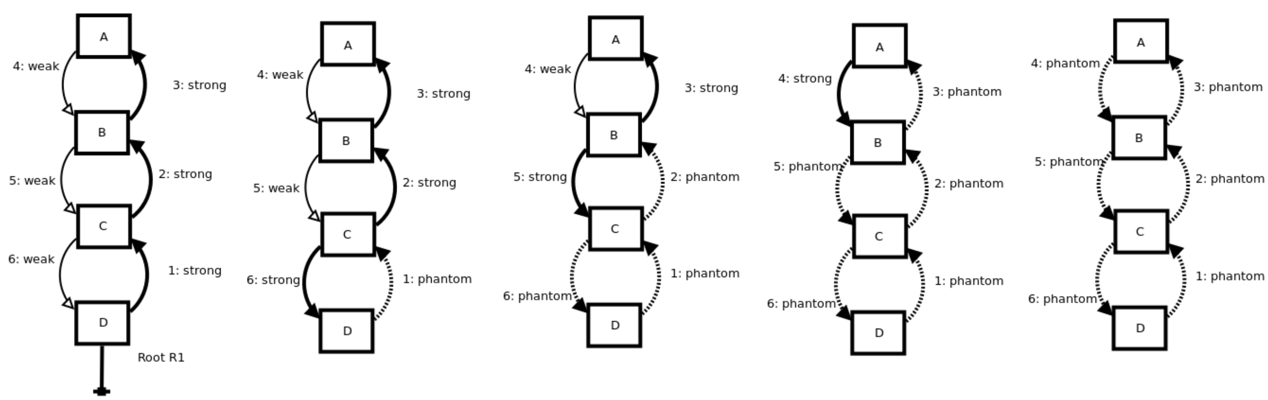

Figure 3. Doubly-linked list

on different subgraphs can proceed independently until, and unless they meet.

If G is the given graph and α is the root set prior to any deletions (see Fig. 5), $\alpha = \{A, B\}$, and $\alpha = \{A\}$ after deletions, then G will become G_A, the nodes reachable from A. Thus, $G = G_A \cup G_B$ initially, and the garbage due to the loss of B will be Γ_B.

$$\Gamma_B = G_B - (G_A \cap G_B).$$

During phantomization, all nodes in Γ_B and some nodes in $G_A \cap G_B$ will be marked. During recovery, the nodes in $G_A \cap G_B$ will all be unmarked. Hence, after the $Recovery$ phase, all nodes in $G_A \cap G_B$ will be strongly connected to A. The final phase $CleanUp$ discards the marked memory, Γ_B.

The above discussion holds equally well if instead of being a root, B is the only strong link connecting subgraph G_A to subgraph G_B. See Fig. 6.

THEOREM 4.1 (Cycle Invariant). *No strong cycles are possible, and all cycles formed in the graph should have at least one weak or phantom edge in the cyclic path.*

Proof. (sketch) This invariant should be maintained through out the graph for any cycles for the algorithm. This property ensures the

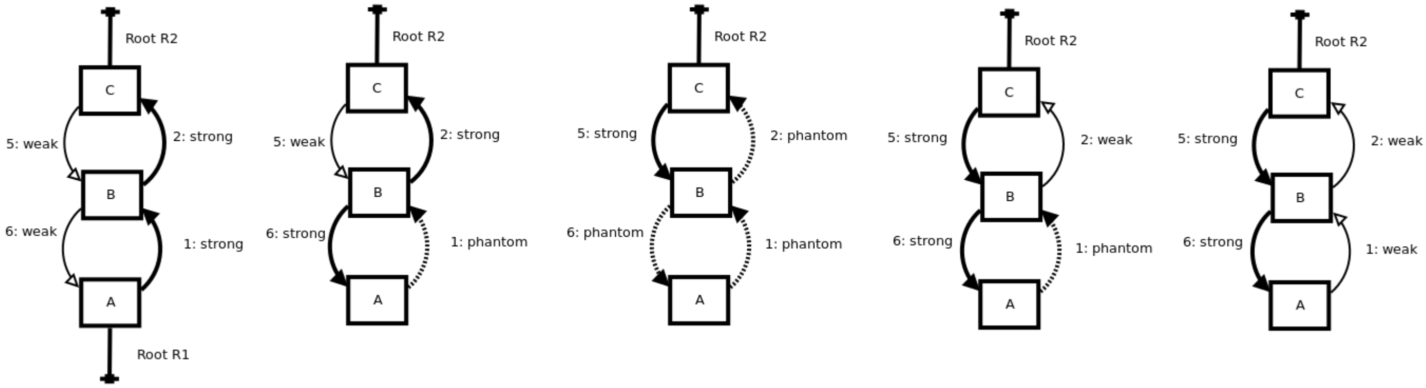

Figure 4. Rebalancing a doubly-linked list

correctness of the algorithm and the definition of the stable graph. In the rules below, the edge being created is E, the source node (if applicable) is called S, the target node is T. The edge creation rules state:

0. Roots always count as strong edges to nodes.
1. Edge E can be strong if S has no incoming strong edges from other nodes, i.e. if there is no node C with a strong edge to S, then E can be created strong. Thus, S is a source of strong edges.
2. A new edge E can be strong if node T has no outgoing non-phantom edges, and $S \neq T$. Thus, T is a terminus of strong edges.
3. A new edge E is strong if T is phantomized and S is not. Thus, T is a terminus of strong edges.
4. A new edge E is weak otherwise.
5. Edge E can only be converted to strong if T is phantomized.
6. A new edge E must be phantom if the node S is phantomized.

Any change described by the above rules regarding strong edges results in one of the nodes becoming a source or terminus of strong edges. Hence, no strong cycles are possible. □

THEOREM 4.2 (Termination). *Any mutations to a stable graph G will take $\mathcal{O}(N)$ time steps to form a new stable graph G', where N is number of edges in the affected subgraph.*

Proof. (sketch) By *stable graph* we mean a graph in which all nodes are strongly connected from the roots and no phantom links or phantomized nodes are present. Mutations which enlarge the graph, e.g. adding a root, or edge are constant time operations since they update the counters and outgoing edge list in a node. Mutations which diminish the graph, e.g. deleting roots, or edges potentially begin a process of $Phantomization$, which may spread to any number of nodes in G.

To prove the algorithm is linear we have to prove that each of the three phases in the algorithm is linear in time. Without loss of generality, consider the graph in Fig. 5 (or, equivalently, Fig. 6). In this graph there are two sets of root links A and B leading into graph G. The graph has three components $G_A, G_B, G_A \cap G_B$. So,

$$G_A \cap G_B \subset G_A$$

and

$$G_A \cap G_B \subset G_B,$$

where π_A and π_B are only reachable by A and B, such that

$$\pi_A = G_A - G_A \cap G_B,$$

$$\pi_B = G_B - G_A \cap G_B.$$

$Phantomization$ starts when a node attempts to convert its weak links to strong, and marks a path along which strong links are lost. $Phantomization$ stops when no additional nodes in the affected subgraph lose strong support. In Fig. 5 (or Fig. 6), the marking process will touch at least π_B, and at most, all of G_B. The marking step affects both nodes and edges in G_B and ensures that graph is not traversed twice. Thus, $Phantomization$ will take at most $\mathcal{O}(N)$ steps to complete where N is the number of edges in G_B.

$Recovery$ traverses all nodes in G_B identified during $Phantomization$. If the node is marked and has a strong count, it unmarks the node and rebuilds its outgoing edges, making them strong or weak according to the rules above. The nodes reached by outgoing links are, in turn, $Recovered$ as well. Since $Recovery$ involves the unmarking of nodes, it is attempted for every node and edge identified during phantomization, and can happen only once, and can take at most $\mathcal{O}(N)$ steps to complete.

Once the $Recovery$ operations are over, then $CleanUp$ traverses the nodes in the recovery list. For each node that is still marked as phantomized, the node's outgoing links are deleted. At the end of this process, all remaining nodes will have zero references and can be deleted. Because this operation is a single traversal of the remaining list, it too is manifestly linear. □

THEOREM 4.3 (Safety). *Every node collected by our algorithm is indeed garbage and no nodes reachable by roots are collected.*

Proof. (sketch) Garbage is defined as a graph not connected to any roots. If the garbage graph contains no cycles, then it must have at least one node with all zero reference counts. However, at the point it reached all zero reference counts, the node would have been collected, leaving a smaller acyclic garbage graph. Because the smaller garbage graph is also acyclic, it must lose yet another node. So acyclic graphs will be collected.

If a garbage graph contains cycles, it cannot contain strong cycles by Theorem 4.1. Thus, there must be a first node in the chain of strong links. However, at the point where a node lost its last strong link, it would have either been collected or phantomized, and so it can not endure. Since there no first link in the chain of strong links can endure, no chain of strong links can endure in a garbage graph. Likewise, any node having only weak incoming links will phantomize. Thus, all nodes in a garbage graph containing cycles must eventually be phantomized.

If such a state is realized, $Recovery$ will occur and fail, and $Cleanup$ will delete the garbage graph.

Alternatively, we show that an object reachable from the roots will not be collected. Suppose V^C is a node and there is an acyclic chain of nodes $Root \rightarrow ... \rightarrow V^A \rightarrow V^B \rightarrow ... \rightarrow V^C$. Let V^A be a node that is reachable from a root, either directly or by some chain of references. If one of the nodes in the chain, V^B, is connected to V^A and supported only by weak references, then at the moment V^B lost its last strong link it would have phantomized and converted any incoming weak link from V^B to strong. If V^B was connected by a phantom link from V^A, then V^B is on the recovery list and will be rebuilt in $Recovery$. This logic can be repeated for nodes from V^B onwards, and so V^C will eventually be reconnected by strong links, and will not be deleted. □

THEOREM 4.4 (Liveness). *For a graph of finite size, our algorithm eventually collects all unreachable nodes.*

Proof. (sketch) We say that a garbage collection algorithm is $live$ if it eventually collects all unreachable objects, i.e. all unreachable objects are collected and never left in the memory.

The only stable state for the graph in our garbage collection is one in which all nodes are connected by strong links, because any time the last strong link is lost a chain of events is initiated which either recovers the links or deletes the garbage. Deletion of a last strong link can result in immediate collection or $Phantomization$. Once $Phantomization$ begins, it will proceed to completion and be followed $Recovery$ and $Cleanup$. These operations will either delete garbage, or rebuild the strong links. See Theorem 4.3. □

Note that the live system may create objects faster than the $Phantomization$ phase can process. In this case, the $Phantomization$ phase will not terminate. However, in Theorem 4.4 when we say the graph be "of finite size" we also count nodes that are unreachable but as yet uncollected, which enables us to bound the number of nodes that are being added while the $Phantomization$ is in progress. On a practical level, it is possible for garbage to be created too rapidly to process and the application could terminate with an out-of-memory error.

5. Experimental Results

To verify our work, we modeled the graph problem described by our garbage collector in Java using fine-grained locks. Our implementation simulates the mutator and collector behavior that would occur in a production environment. Our mutator threads create, modify, and delete edges and nodes, and the collector threads react as necessary. This prototype shows how a real system should behave, and how it scales up with threads.

We also developed various test cases to verify the correctness of the garbage collector implementation. Our test cases involve a large cycle in which the root node is constantly moved to the next node in the chain (a "spinning wheel"), a doubly linked list with a root node that is constantly shifting, a clique structure, and various tests involving a sequence of hexagonal cycles connected in a chain.

In Fig. 7 we collected a large number of hexagonal rings in parallel. This operation should complete in time inversely proportional to the number of threads in use, as each ring is collected independently. The expected behavior is observed.

In Fig. 8 we performed the same test, but to a set of connected rings. The collection threads merge, but not immediately, so the collection time goes down with the number of threads used, but not proportionally because the collection threads only operate in parallel part of the time.

In Fig. 9, we perform tests to see whether our garbage collector is linear. We considered a clique, two different hexagonal cycles (one is interlinked and other separate), a doubly-linked list, and simple cycles, and measured the collection time per object by varying the size of the graph and fixing the collector threads to two all times. The results confirmed that our collector in indeed linear in time.

Our tests are performed on two 2.6 GHz 8-Core Sandy Bridge Xeon Processors (i.e. on 16 cores) running Redhat Linux 6 64-bit operating system.

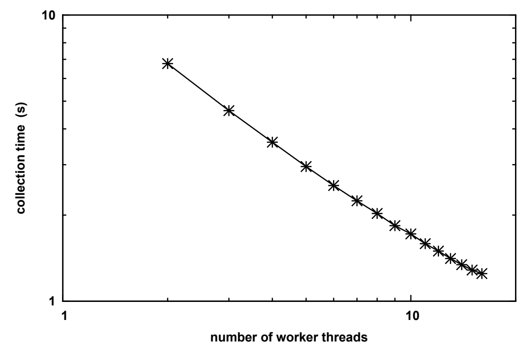

Figure 7. A large number of independent rings are collected by various number of worker threads. Collection speed drops linearly with the number of cores used.

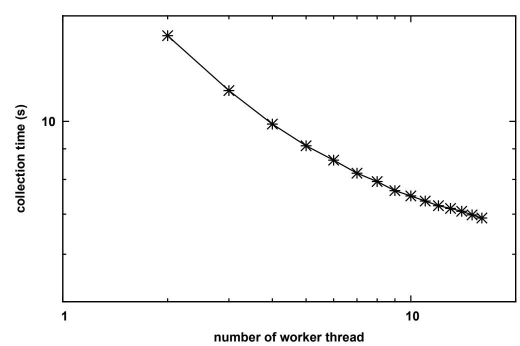

Figure 8. A chain of linked cycles is created in memory. The connections are severed, then the roots are removed. Multiple collector threads are created and operations partially overlap.

6. Future Work

There are numerous avenues to explore within this modified Brownbridge framework.

First there are the details of the concurrent operation. While we have explored the use of merging collector threads upon phantomization, we have not made use of parallelism within a single collection thread's operations. Doing so may or may not be desirable, depending on the requirements for liveness and the compute needs of the application.

Our implementation uses fine-grained locking, but an approach using atomic variables or atomic transactions should be possible.

Because the situations that lead to nontrivial work are algorithmic, it should be possible for compilers to optimize instructions to create or remove links to avoid work. Likewise, it should be possible to build profiling tools that identify garbage collection hot spots within a program, and give the programmer the option to take steps

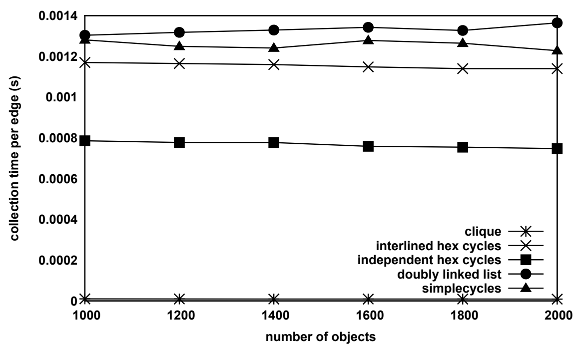

Figure 9. Graphs of different types are created at various sizes in memory, including cliques, chains of cycles, large cycles, and large doubly linked lists. Regardless of the type of object, collection time per object remains constant, verifying the linearity of the underlying collection mechanism.

(e.g. re-arrange link structure) to minimize the garbage collector's work.

We also intend to build a high performance distributed version in C++, for use in High Performance Computing (HPC) systems. HPC is entering a new phase in which more complex simulations are being carried out, covering multiple orders of time and space resolution, and merging the time evolution of multiple kinds of calculations within a single calculation. Parallelization of such systems is nontrivial, and increasingly researchers are moving to dynamic parallelism managed by task queues and active messages. Numerous frameworks exist, or are being constructed for this purpose (Charm++ [17], HPX [16], Swarm/ETI [18], UNITAH [6], The Global Arrays Toolkit [25], etc.)

Until this point, these systems have either required programmers to manage memory themselves, or use a form of simple reference counting. None of the systems above offer an option for global garbage collection capable of reclaiming cycles. Because garbage collection tends to be a complex, global operation which traces links throughout memory, it has not been considered viable in high performance computing to date. However, if the trend toward increasingly complex multi-scale multi-physics simulations continues, it may only be a time before garbage collection becomes a requirement.

7. Conclusion

We have described a garbage collector based on strong and weak reference counts, proven its validity, and illustrated its potential advantages over existing systems, including:

1. It can run at the same time as a live system, using multiple threads if that is desired, without needing to "stop the world."
2. It has a reduced need to operate on memory compared to collectors because it only performs nontrivial work when the last strong link is removed;
3. It has a reduced need to trace links in performing its operations, specifically:
   (a) When objects do not need to be collected, it is often able to prove this without completely tracing cycles to which they belong which have support;
   (b) It remembers stable paths through memory and thus avoids doing work on old objects, a benefit similar to that derived from generational collectors, e.g. [29]; deletions are occurring;
   (c) When objects do need to be collected, the collector only needs to trace the cycle twice;
   (d) The collector operation is local;
   (e) The collector does not need or use back-pointers;

These advantages should make our collector useful for distributed systems, where traces that cross node boundaries are likely to be extremely expensive.

Disadvantages include:

1. An increased memory overhead: three counters and a pointer are required for each object;
2. An additional cost of pointer creation/mutation;
3. The lack of a fault tolerance protocol;
4. Little effort has, as yet, been devoted to optimization of any implementation. Further reduction of memory and computational overheads may yet be achieved with variations on this algorithm.

While there are undoubtedly cases for which the increased overheads are unacceptable, there are just as undoubtedly cases where the potential performance gains make it acceptable.

For distributed applications, a fault tolerance protocol may be of high importance, depending especially on the reliability of the application components. We expect, however, that variants of this protocol and synchronization strategies associated with it may be discovered to assist with these problems.

In short, we feel that collectors based on a system of strong and weak references like the one we have described here have many potential advantages over existing systems and should provide a fertile ground for future research and language development.

Acknowledgments

We acknowledge the support of the following grants: The DoE XPRESS proposal (DE-SC0008714), and the NSF PXGL (1160602). We thank Frank Löffler and Hartmut Kaiser for helpful conversations.

References

[1] URL `https://github.com/stevenrbrandt/MultiThreadBrownbridge`.

[2] D. F. Bacon and V. T. Rajan. Concurrent cycle collection in reference counted systems. In *ECOOP*, pages 207–235, 2001.

[3] D. F. Bacon, C. R. Attanasio, H. B. Lee, V. T. Rajan, and S. Smith. Java without the coffee breaks: a nonintrusive multiprocessor garbage collector. *SIGPLAN Not.*, 36(5):92–103, 2001.

[4] D. F. Bacon, P. Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. *SIGPLAN Not.*, 38(1): 285–298, 2003.

[5] K. Barabash, O. Ben-Yitzhak, I. Goft, E. K. Kolodner, V. Leikehman, Y. Ossia, A. Owshanko, and E. Petrank. A parallel, incremental, mostly concurrent garbage collector for servers. *ACM Trans. Program. Lang. Syst.*, 27(6):1097–1146, 2005.

[6] M. Berzins, J. Luitjens, Q. Meng, T. Harman, C. A. Wight, and J. R. Peterson. Uintah: a scalable framework for hazard analysis. In *TG*, pages 3.1–3.8, 2010.

[7] D. G. Bobrow. Managing reentrant structures using reference counts. *ACM Trans. Program. Lang. Syst.*, 2(3):269–273, 1980.

[8] D. Brownbridge. Cyclic reference counting for combinator machines. In J.-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 273–288. 1985.

[9] T. W. Christopher. Reference count garbage collection. *Software: Practice and Experience*, 14(6):503–507, 1984.

[10] G. E. Collins. A method for overlapping and erasure of lists. *Commun. ACM*, 3(12):655–657, 1960.

[11] D. Frampton. *Garbage Collection and the Case for High-level Low-level Programming*. PhD thesis, Australian National University, June 2010.

[12] D. P. Friedman and D. S. Wise. Reference counting can manage the circular environments of mutual recursion. *Inf. Process. Lett.*, 8(1): 41–45, 1979.

[13] L. Huelsbergen and P. Winterbottom. Very concurrent mark-&-sweep garbage collection without fine-grain synchronization. *SIGPLAN Not.*, 34(3):166–175, 1998.

[14] R. Hughes. Managing reduction graphs with reference counts. Departmental Research Report CSC/87/R2, March 1987.

[15] R. Jones and R. Lins. *Garbage collection: algorithms for automatic dynamic memory management*. John Wiley & Sons, Inc., New York, NY, USA, 1996. ISBN 0-471-94148-4.

[16] H. Kaiser, M. Brodowicz, and T. Sterling. Parallex an advanced parallel execution model for scaling-impaired applications. In *ICPPW*, pages 394–401, 2009.

[17] L. V. Kale and S. Krishnan. *CHARM++: a portable concurrent object oriented system based on C++*, volume 28. 1993.

[18] C. Lauderdale and R. Khan. Towards a codelet-based runtime for exascale computing: position paper. In *EXADAPT*, pages 21–26, 2012.

[19] Y. Levanoni and E. Petrank. An on-the-fly reference-counting garbage collector for java. *ACM Trans. Program. Lang. Syst.*, 28(1):1–69, 2006.

[20] R. D. Lins. Cyclic reference counting with lazy mark-scan. *Inf. Process. Lett.*, 44(4):215–220, 1992.

[21] R. D. Lins. An efficient algorithm for cyclic reference counting. *Inf. Process. Lett.*, 83(3):145–150, 2002.

[22] R. D. Lins. Cyclic reference counting. *Inf. Process. Lett.*, 109(1):71 – 78, 2008.

[23] J. H. McBeth. Letters to the editor: on the reference counter method. *Commun. ACM*, 6(9):575, 1963.

[24] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, 1960.

[25] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global arrays: A portable shared-memory programming model for distributed memory computers. In *SC*, pages 340–349, 1994.

[26] H. Paz, D. F. Bacon, E. K. Kolodner, E. Petrank, and V. T. Rajan. An efficient on-the-fly cycle collection. *ACM Trans. Program. Lang. Syst.*, 29(4), 2007.

[27] E. Pepels, M. Plasmeijer, C. van Eekelen, and M. Eekelen. *A Cyclic Reference Counting Algorithm and Its Proof*. Internal report 88-10. Department of Informatics, Faculty of Science, University of Nijmegen, 1988.

[28] F. Pizlo, E. Petrank, and B. Steensgaard. A study of concurrent real-time garbage collectors. *SIGPLAN Not.*, 43(6):33–44, 2008.

[29] T. Printezis and D. Detlefs. A generational mostly-concurrent garbage collector. *SIGPLAN Not.*, 36(1):143–154, 2000.

[30] P. Roy, S. Seshadri, A. Silberschatz, S. Sudarshan, and S. Ashwin. Garbage collection in object-oriented databases using transactional cyclic reference counting. *The VLDB Journal*, 7(3):179–193, 1998.

[31] J. Salkild. Implementation and analysis of two reference counting algorithms. Master thesis, University College, London, 1987.

[32] R. Shahriyar, S. M. Blackburn, and D. Frampton. Down for the count? getting reference counting back in the ring. In *ISMM*, pages 73–84, 2012.

[33] L. Veiga and P. Ferreira. Asynchronous complete distributed garbage collection. In *IPDPS*, pages 24.1–24.10, 2005.

A. Appendix

A.1 Multi-Threaded Collector

Note that in the following sections, locks are always obtained in canonical order that avoids deadlocks. Unlock methods unlock the last set of items that were locked.

The lists on the collectors are thread-safe.

The collector object is itself managed by a simple reference count. Code for incrementing and decrementing this count is not explicitly present in the code below.

A reference implementation is also provided [1].

Algorithm 1: LinkSet

```
// LinkSet creates a new link, and decides the type
   of the link to create.
LinkSet(link,node):
  lock (link.Source,node)
  if node == NULL then
    LinkFree(link)
    link.Target = NULL
  EndIf
  if link.Target == node then
    Return
  EndIf
  oldLink = copy(link)
  link.Target = node
  If link.Source.Phantomized then
    MergeCollectors(link.Source, link.Target)
    link.PhantomCount++
    link.Phantomized = True
  ElseIf link.Source == node then
    link.Which = 1 - node.Which
    node.Count[link.Which]++
  ElseIf node.Links not initialized then
    node.Count[link.Which]++
    link.Which = node.Which
  Else
    link.which = 1 - node.Which
    node.Count[link.Which]++
  EndIf
  If oldLink != NULL
    LinkFree(oldLink)
  unlock()
```

Algorithm 2: LinkFree

```
// Freeing a link is usually just the decrement of a
   reference count, but if it is the last strong
   count, this could potentially start a
   Phantomization process.
LinkFree(link):
  lock(link.Source,link.Target)
  If link.Target == NULL then
    Return
  EndIf
  If link.Phantomized then
    DecPhantom(link.Target)
  Else
    link.Target.Count[link.Which]- -
    If link.Target.Count[link.Which] == 0 And
        link.Target.Which == link.Which then
      If link.Target.Count[1-link.Target.Which] == 0 And
          link.Target.PhantomCount == 0 then
        Delete(node)
        link.Target = NULL
      Else
        If link.Target.Collector == NULL then
          link.Target.Collector = new Collector()
        EndIf
        AddToCollector(link.Target)
      EndIf
    EndIf
  EndIf
  unlock()
```

Algorithm 3: AddToCollector

```
// Adding an object to the collector puts back the
   strong count, effectively transferring the source
   of the strong link to the collector. It also adds
   a phantom count, which helps prevent the clearing
   of the Collector field.
AddToCollector(node):
  While True
    lock(node,node.Collector)
    If node.Collector.Forward != NULL then
      node.Collector = node.Collector.Forward
    Else
      node.Count[node.Which]++
      node.PhantomCount++
      node.Collector.CollectionList.append(node)
      Break
    EndIf
    unlock()
  EndWhile
```

Algorithm 4: PhantomizeNode

```
// The collector takes away the strong link it made
   in AddToCollector().
PhantomizeNode(node,collector):
  lock(node)
  While collector.Forward != NULL
    collector = collector.Forward
  EndWhile
  node.Collector = collector
  node.Count[node.Which]- -
    // Prevent deletion while the
    // node is managed by the Collector
  Let phantomize = False
  If node.Count[node.Which] > 0 then
    Return
  Else
    If node.Count[1-node.Which] > 0 then
      node.Which = 1-node.Which
    EndIf
    If Not node.Phantomized then
      node.Phantomized = True
      node.PhantomizationComplete = False
      phantomize = True
    EndIf
  EndIf
  Let links = NULL
  If phantomize then
    links = copy(node.Links)
  EndIf
  unlock()
  ForEach outgoing link in links
    PhantomizeLink(link)
  EndFor
  lock(node)
  node.PhantomizationComplete = True
  unlock()
```

Algorithm 5: Collector.Main

```
// This method describes the work to be carried out
   by a garbage collection thread. Live objects
   pointing to this collector, or Forward pointers
   from other collectors contribute to the RefCount
   field on the Collector.
Collector.Main():
  While True
    WaitFor(Collector.RefCount == 0 Or Work to do)
    If Collector.RefCount == 0 And No work to do then
      Break
    EndIf
    While Collector.MergedList.size() > 0
      Let node = Collector.MergedList.pop()
      Collector.RecoveryList.append(node)
    EndWhile
    While Collector.CollectionList.size() > 0
      Let node = Collector.CollectionList.pop()
      PhantomizeNode(node,Collector)
      Collector.RecoveryList.append(node)
    EndWhile
    While Collector.RecoveryList.size() > 0
      Let node = Collector.RecoveryList.pop()
      RecoverNode(node)
      Collector.CleanList.append(node)
    EndWhile
    While Collector.RebuildList.size() > 0
      Let node = Collector.RebuildList.pop()
      RecoverNode(node)
    EndWhile
    While Collector.CleanList.size() > 0
      Let node = Collector.CleanList.pop()
      CleanNode(node)
    EndWhile
  EndWhile
```

Algorithm 6: PhantomizeLink

```
PhantomizeLink(link):
  lock(link.Source,link.Target)
  If link.Target == NULL then
    unlock()
    Return
  EndIf
  If link.Phantomized then
    unlock()
    Return
  EndIf
  link.Target.PhantomCount++
  link.Phantomized = True
  linkFree(link)
  MergeCollectors(link.Source, link.Target)
  unlock()
```

Algorithm 7: DecPhantom

```
// DecPhantom is responsible for removing any
   reference to the collector.
DecPhantom(node):
  lock(node)
  node.PhantomCount- -
  If node.PhantomCount == 0 then
    If node.Count[node.Which]== 0 And
        node.Count[1-node.Which] == 0 then
      Delete(node)
    Else
      node.Collector = NULL
    EndIf
  EndIf
  unlock()
```

Algorithm 8: RecoverNode

```
RecoverNode(node):
  lock(node)
  Let links = NULL
  If node.Count[node.Which] > 0 then
    WaitFor(node.PhantomizationComplete == True)
    node.Phantomized = False
    links = copy(node.Links)
  EndIf
  unlock()
  ForEach link in links
    Rebuild(link)
  EndFor
```

Algorithm 9: Rebuild

```
Rebuild(link):
  lock(link.Source,link.Target)
  If link.Phantomized then
    If link.Target == link.Source then
      link.Which = 1- link.Target.Which
    ElseIf link.Target.Phantomized then
      link.Which = link.Target.Which
    ElseIf count(link.Target.Links) == 0 then
      link.Which = link.Target.Which
    Else
      link.Which = 1-link.Target.Which
    EndIf
    link.Target.Count[link.Which]++
    link.Target.PhantomCount- -
    If link.Target.PhantomCount == 0 then
      link.Target.Collector = NULL
    EndIf
    link.Phantomized = False
    Add link.Target to Collector.RecoveryList
  EndIf
  unlock()
```

Algorithm 10: CleanNode

```
// After deleting all the outgoing links, decrement
   the phantom count by one (i.e. the reference held
   by the collector itself). When the last phantom
   count is gone, the object is cleaned up.
CleanNode(node):
  lock(node)
  Let die = False
  If node.Count[node.Which]== 0 And
      node.Count[1-node.Which]== 0 then
    die = True
  EndIf
  unlock()
  If die then
    ForEach link in node
      LinkFree(link)
    EndFor
  EndIf
  DecPhantom(node)
```

Algorithm 11: Delete

```
Delete(node):
    ForEach link in node
      LinkFree(link)
    EndFor
    freeMem(node)
```

Algorithm 12: MergeCollectors

```
// When two collector threads realize they are
   managing a common subset of objects, one defers
   to the other. The arguments, source and target,
   are both nodes.
MergeCollectors(source,target):
  Let s = source.Collector
  Let t = target.Collector
  Let done = False
  If s == NULL And t != NULL then
    lock(source)
    source.Collector = t
    unlock()
    Return
  EndIf
  If s != NULL And t == NULL then
    lock(target)
    target.Collector = s
    unlock()
    Return
  EndIf
  If s == NULL Or s == NULL then
    Return
  EndIf
  While Not done
    lock(s,t,target,source)
    If s.Forward == t and t.Forward == NULL then
      target.Collector = s
      source.Collector = s
      done = True
    ElseIf t.Forward == s and s.Forward == NULL then
      target.Collector = t
      source.Collector = t
      done = True
    ElseIf t.Forward != NULL then
      t = t.Forward
    ElseIf s.Forward != NULL then
      s = s.Forward
    Else
      Transfer s.CollectionList to t.CollectionList
      Transfer s.MergedList to t.MergedList
      Transfer s.RecoveryList to t.MergedList
      Transfer s.RebuildList to t.RebuildList
      Transfer s.CleanList to t.MergedList
      target.Collector = t
      source.Collector = t
      done = True
    EndIf
    unlock()
  EndWhile
```