TOWARD SCALABLE, CONCURRENT, PARALLEL GARBAGE COLLECTION

Hari Krishnan

Thesis Proposal

Committee Members :

Konstantin Busch (advisor)

Steven R Brandt (co-advisor)

Gerald Baumgartner

1 Introduction

Garbage collection (GC) is very useful for software development. It helps the developers to avoid explict deallocation and also avoid dangling pointers. There has been constant complaint that the GC reduces the perfomance of the application. But the recent research shows that there is infact much benefit in using it. With the advent of multicore, the processor utilization is of the high concern and application post multi-core era has been focusing on building concurrent applications. Garbage Collections are designed to be concurrent and parallel. The distintion between concurrent and parallel is if the application runs simultaneously with the garbage collection it is termed as concurrent collector while there are more than one collector thread runs during the collection, it is called as parallel collector. There has been lot of focus on concurrent collectors lately.

In HPC community, applications are mostly written in languages with no or less managed runtime systems. It is just time that all applications that are written for the huge clusters to change to managed runtime systems. The HPC applications can benefit all the advantages of the garbage collection. Predominantly HPC codes are written for the two kind of the memory architectures. The first being the shared memory systems, where the memory is common for all the processing element. Other being distributed memory system where each processing element has its own share of memory. Bacon et al described the theory of garbage collection in very lucid way. The proposed hyphothesis is based on the Bacon et al observation on the different kinds of garbage collector. The two main GC techniques are mark-sweep (MS) and reference counting (RC). All the available GC algorithm follows one of these approach to identify garbage. In the abstract way, MS traces all the live heap objects and then deletes the untraced but allocated heap objects. On the other hand, RC traces all the dead/potential dead objects and then deletes the traced objects.

The hyphothesis of this research is from the above abstarct idea. If MS collector can determine the garbage by traversing the live heap objects, then in a huge memory systems with more live objects the collector should perform worse than RC collectors. In reality, MS is most preffered GC than RC. It is constantly proved that RC performs worse than MS. RC implementations have write barriers which require additional instruction to update the reference count of the heap objects on every change in the object graph. In a highly optimized GC system with huge memory, RC based systems will perform better than MS. The research steer towards RC based garbage collection.

2 Literature Review

McCarthy invented the concept of Garbage Collection in 1959 [26]. His works focused on indirect method of collecting garbage which is now called mark-sweep or tracing collector. While Mark-sweep was the only method that was available at the early 1960's, Collins designed a new method called reference counting, which is more direct method to detect garbage [9]. Dijkstra et al [10] describes the correctness condition of tricolor tracing based collector. This work redefined the way the collectors should be designed and how one can prove the properties of the collectors. Haddon et al [15] realized the fragmentation issues in the mark-sweep algorithm and designed a mark-compact algorithm. Fenichel et al [11] and Cheney [7] provided new kind of algorithm based on copying the live objects which was called copying colletors. Based on Foderaro and Fateman [12] observation, 98% of the objects collection has been allocated after the last collection. Several such observations were made and reported in [36, 33]. Appel designed a simple generation garbage collection [1].

While most of the literature up until now considered a indirect method of collecting garbage through tracing, the other form of collection is counting the references. While counting the references seems obvious way to solve the problem, McBeth's [**?**] article on the inability to collect cyclic garbage structures by reference counting method made the method less practical.

Bobrow's [5] solution to collect cylic garbage based on reference counting method was the first hybrid collector which uses both tracing and counting the references. Hughes [17, 18]

identified the issues in the Bobrow's algorithm. It was proven by McBeth [25] in early sixties that reference counting collectors were unable to handle cyclic structures; several attempts to fix this problem appeared subsequently, e.g. [14, 5, 24]. We give details in Section **??**. In contrast to the approach followed in [14, 5, 24] and several others, Brownbridge [6] 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) [20]. 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 [32] showed that Brownbridge's algorithm [6] 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 [32] 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. [29] 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 [20]. 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.

Typical hybrid reference count and collection systems, e.g. [3, 21, 2, 4, 24], 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 [6, 20]. 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., [25, 6, 32, 29, 3, 2, 4, 20]. 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 [3]. Reference counting approaches were first developed for LISP by Collins [**?**]. Improved variations were proposed in several subsequent papers, e.g. [14, 19, 20, 24, 21]. We direct readers to Shahriyar et al. [34] for the valuable overview of the current state of reference counting collectors.

It was noticed by McBeth [25] in early sixties that reference counting collectors were unable to handle cyclic structures. After that several reference counting collectors were developed, e.g. [14, 5, 22, 23]. The algorithm in Friedman [14] dealt with recovering cyclic data in immutable structures, whereas Bobrow's algorithm [5] can reclaim all cyclic structures but relies on the explicit information provided by the programmer. Trial deletion approach was studied by Christopher [8] which tries to collect cycles by identifying groups of self-sustaining objects. Lins [22] used a cyclic buffer to reduce repeated scanning of the same nodes in their mark-scan algorithm for cyclic reference counting. Moreover, in [23], Lins improved his algorithm from [22] 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 [21]. Recent work in reference counting algorithms, e.g. [4, 21, 3, 2], 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 [25]. Therefore, concurrent reference counting collectors [4, 21, 3, 2, 28, 24] 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 [28] combines the sliding view reference counting concurrent collector of [21] with the cycle collector of [3]. 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 [13].

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. [31]. Huelsbergen and Winterbottom [16] proposed an incremental algorithm for the concurrent garbage collection that is a variant of mark-and-sweep collection scheme first proposed in [27]. Furthermore, garbage collection is also considered for several other systems, namely real-time systems and asynchronous distributed systems, e.g. [30, 35].

Concurrent collectors are gaining popularity. The concurrent collector described in Bacon and Rajan [3] 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.

3 Preliminary Work

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 [6] 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. 1.

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 2 in Section 3.8. 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.

3.1 Example: A Simple Cycle

In Fig. 1 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.

3.2 Example: A Doubly-Linked List

The doubly linked list depicted in Fig. 2 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. 1, which is necessary for a more complex figure.

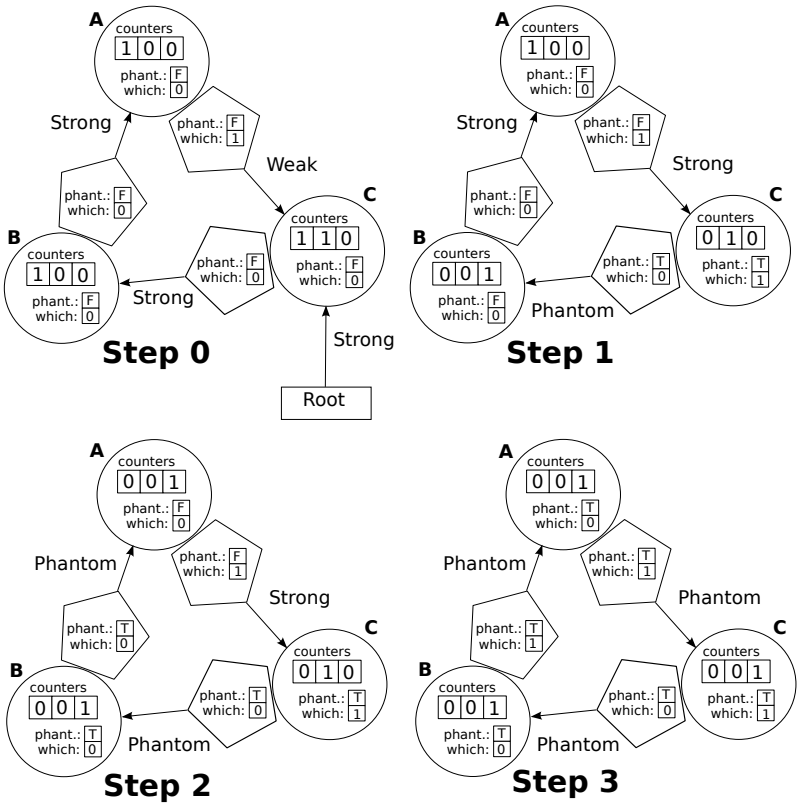

Figure 1: Reclaiming a cycle with three objects

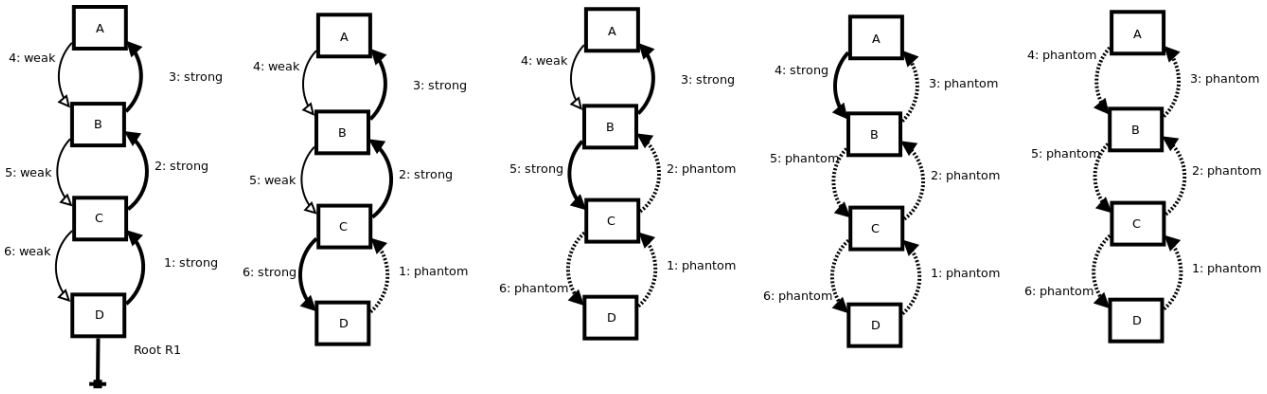

Figure 2: Doubly-linked list

3.3 Example: Rebalancing A Doubly-Linked List

Fig. 3 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.

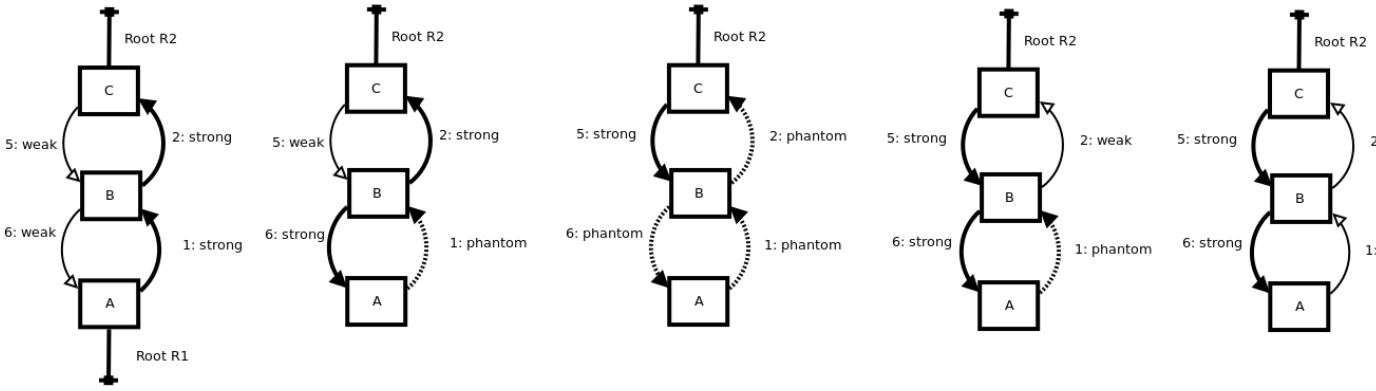

Figure 3: Rebalancing a doubly-linked list

3.4 Example: Recovering Without Detecting a Cycle

In Fig. **??** 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.

3.5 Concurrency Issues

This section provides details of the implementation.

3.6 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.

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.

3.7 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

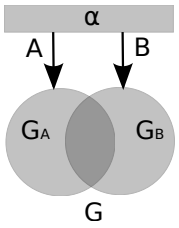

Figure 4: Graph model

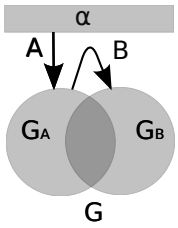

Figure 5: Subgraph model

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.

3.8 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 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. 4), $\alpha = \{A, B\}$, 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. 5.

Theorem 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. This invariant should be maintained through out the graph for any cycles for the algorithm. This property ensures the 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 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. 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. 4 (or, equivalently, Fig. 5). 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. 4 (or Fig. 5), 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 3 (Safety)**.** *Every node collected by our algorithm is indeed garbage and no nodes reachable by roots are collected.*

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

Proof. 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 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 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.

3.9 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. 6 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. 7 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. 8, 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.

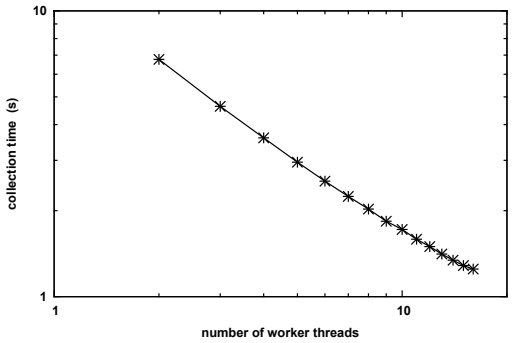

Figure 6: A large number of independent rings are collected by various number of worker threads. Collection speed drops linearly with the number of cores used.

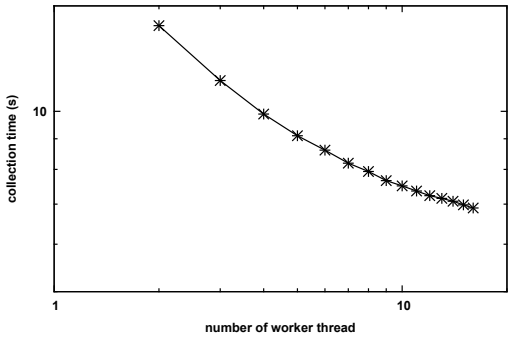

Figure 7: A chain of linked cycles is created in memory. The connections are severed, then the roots are removed. Multiple collector threads are created and operations partially overlap.

4 Proposed Work

In HPC community, applications are written for both the shared memory and distributed memory systems. The preliminary work uses the centralized queue based solution. This work can be applied to shared memory systems and distributed memory systems. Although the solution is applicable to both of the systems, the distributed memory would benefit more from completely decentralized solution. So the work is focused more towards the decentralized/localized solution to the distributed garabge collection.

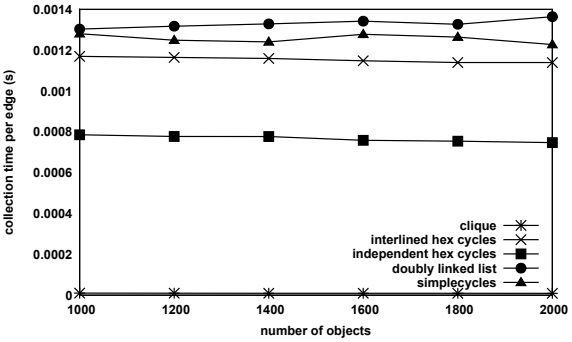

Figure 8: Graphs of different types are created at various sizes in memory, including cliques, chains of cycles, large cycles, and large doubly linked lists. Regardless of the type of object, collection time per object remains constant, verifying the linearity of the underlying collection mechanism.

Fundamental thesis of the work is any tracing based algorithm traces the entire memory. But hybrid approaches like our algorithm only traces the affected subgraph. So in practical use, the graph traversed to collect garbage is just infinitesimal of the total graph. So the throughput of the garbage collector should be better in case of our approach. We want to extend the concept of strong and weak to the distributed garbage collection. Various approaches up until now available for the distributed garbage collectors are not complete. They are not practical solution for various reason. Our proposed work is to build a reliable, fault-tolerant, linear garbage collector in the distributed memory system. Apart from the designing the algorithm, we would also like to evaluate the performance of the the previously described algorithm in a virtual machine and also try to implement the solution to be discovered in a scientific framework to compare with the other algorithms.

5 Methodology and Evaluation

I am on the process of implementing the garbage collector in the OpenJDK project. The plan to evaulate the garbage collector algorithm for the shared memory is to implement in the HotSpot Virtual Machine in OpenJDK project and measure the performances in terms of various parameters to prove the hyphothesis is right.

6 Conclusion

This proposal is aimed at designing a RC based collector that is applicable to shared and distributed memory systems efficiently. The contributions would be reviving the old RC technique to collect cyclic garbage, design a localized garbage collection algorithm that perform efficiently than the available distributed collector. If possible provide fault-tolerance guarantees by designing the algorithm with self-stabilization property.

References

[1] Andrew W. Appel. Simple generational garbage collection and fast allocation. *Softw., Pract. Exper.*, 19(2):171–183, 1989.

[2] David F. Bacon, Clement R. Attanasio, Han B. Lee, V. T. Rajan, and Stephen Smith. Java without the coffee breaks: a nonintrusive multiprocessor garbage collector. *SIGPLAN Not.*, 36(5):92–103, 2001.

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

[4] Katherine Barabash, Ori Ben-Yitzhak, Irit Goft, Elliot K. Kolodner, Victor Leikehman, Yoav Ossia, Avi Owshanko, and Erez Petrank. A parallel, incremental, mostly concurrent garbage collector for servers. *ACM Trans. Program. Lang. Syst.*, 27(6):1097–1146, 2005.

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

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

[7] C. J. Cheney. A non-recursive list compacting algorithm. *Communications of the ACM*, 13(11):677–8, November 1970.

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

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

[10] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Commun. ACM*, 21(11):966–975, November 1978.

[11] Robert R. Fenichel and Jerome C. Yochelson. A Lisp garbage collector for virtual memory computer systems. *Communications of the ACM*, 12(11):611–612, November 1969.

[12] John K. Foderaro and Richard J. Fateman. Characterization of VAX Macsyma. In *1981 ACM Symposium on Symbolic and Algebraic Computation*, pages 14–19, Berkeley, CA, 1981.

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

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

[15] B. K. Haddon and W. M. Waite. A compaction procedure for variable-length storage elements. *The Computer Journal*, 10(2):162–165, 1967.

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

[17] R. John M. Hughes. Reference counting with circular structures in virtual memory applicative systems. Internal paper, Programming Research Group, Oxford, 1983.

[18] R. John M. Hughes. Managing reduction graphs with reference counts. Departmental Research Report CSC/87/R2, University of Glasgow, March 1987.

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

[20] Richard Jones and Rafael Lins. *Garbage collection: algorithms for automatic dynamic memory management*. John Wiley & Sons, Inc., New York, NY, USA, 1996.

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

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

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

[24] Rafael Dueire Lins. Cyclic reference counting. *Inf. Process. Lett.*, 109(1):71 – 78, 2008.

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

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

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

[28] Harel Paz, David F. Bacon, Elliot K. Kolodner, Erez Petrank, and V. T. Rajan. An efficient on-the-fly cycle collection. *ACM Trans. Program. Lang. Syst.*, 29(4), 2007.

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

[30] Filip Pizlo, Erez Petrank, and Bjarne Steensgaard. A study of concurrent real-time garbage collectors. *SIGPLAN Not.*, 43(6):33–44, 2008.

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

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

[33] Patrick M. Sansom and Simon L. Peyton Jones. Generational garbage collection for Haskell. pages 106–116.

[34] Rifat Shahriyar, Stephen M. Blackburn, and Daniel Frampton. Down for the count? getting reference counting back in the ring. In *ISMM*, pages 73–84, 2012.

[35] Luis Veiga and Paulo Ferreira. Asynchronous complete distributed garbage collection. In *IPDPS*, pages 24.1–24.10, 2005.

[36] Benjamin G. Zorn. *Comparative Performance Evaluation of Garbage Collection Algorithms.* PhD thesis, March 1989. Technical Report UCB/CSD 89/544.

A Appendix

A.1 Multi-Threaded Collector Algorithm

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.

Algorithm 1: LinkSet

```
   // LinkSet creates a new link, and decides the type of the link to create.
 1 LinkSet(link,node):
 2   lock (link.Source,node)
 3   if node == NULL then
 4     LinkFree(link)
 5     link.Target = NULL
 6   EndIf
 7   if link.Target == node then
 8     Return
 9   EndIf
10   oldLink = copy(link)
11   link.Target = node
12   If link.Source.Phantomized then
13     MergeCollectors(link.Source, link.Target)
14     link.PhantomCount++
15     link.Phantomized = True
16   ElseIf link.Source == node then
17     link.Which = 1 - node.Which
18     node.Count[link.Which]++
19   ElseIf node.Links not initialized then
20     node.Count[link.Which]++
21     link.Which = node.Which
22   Else
23     link.which = 1 - node.Which
24     node.Count[link.Which]++
25   EndIf
26   If oldLink != NULL
27     LinkFree(oldLink)
28   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.
 1 LinkFree(link):
 2   lock(link.Source,link.Target)
 3   If link.Target == NULL then
 4     Return
 5   EndIf
 6   If link.Phantomized then
 7     DecPhantom(link.Target)
 8   Else
 9     link.Target.Count[link.Which]--
10     If link.Target.Count[link.Which] == 0 And
11          link.Target.Which == link.Which then
12       If link.Target.Count[1-link.Target.Which] == 0 And
13            link.Target.PhantomCount == 0 then
14         Delete(node)
15         link.Target = NULL
16       Else
17         If link.Target.Collector == NULL then
18           link.Target.Collector = new Collector()
19         EndIf
20         AddToCollector(link.Target)
21       EndIf
22     EndIf
23   EndIf
24   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.
 1 AddToCollector(node):
 2   While True
 3     lock(node,node.Collector)
 4     If node.Collector.Forward != NULL then
 5       node.Collector = node.Collector.Forward
 6     Else
 7       node.Count[node.Which]++
 8       node.PhantomCount++
 9       node.Collector.CollectionList.append(node)
10       Break
11     EndIf
12     unlock()
13   EndWhile
```

Algorithm 4: PhantomizeNode

```
   // The collector takes away the strong link it made in AddToCollector().
 1 PhantomizeNode(node,collector):
 2   lock(node)
 3   While collector.Forward != NULL
 4     collector = collector.Forward
 5   EndWhile
 6   node.Collector = collector
 7   node.Count[node.Which]--
 8     // Prevent deletion while the
 9     // node is managed by the Collector
10   Let phantomize = False
11   If node.Count[node.Which] > 0 then
12     Return
13   Else
14     If node.Count[1-node.Which] > 0 then
15       node.Which = 1-node.Which
16     EndIf
17     If Not node.Phantomized then
18       node.Phantomized = True
19       node.PhantomizationComplete = False
20       phantomize = True
21     EndIf
22   EndIf
23   Let links = NULL
24   If phantomize then
25     links = copy(node.Links)
26   EndIf
27   unlock()
28   ForEach outgoing link in links
29     PhantomizeLink(link)
30   EndFor
31   lock(node)
32   node.PhantomizationComplete = True
33   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.
 1 Collector.Main():
 2   While True
 3     WaitFor(Collector.RefCount == 0 Or Work to do)
 4     If Collector.RefCount == 0 And No work to do then
 5       Break
 6     EndIf
 7     While Collector.MergedList.size() > 0
 8       Let node = Collector.MergedList.pop()
 9       Collector.RecoveryList.append(node)
10     EndWhile
11     While Collector.CollectionList.size() > 0
12       Let node = Collector.CollectionList.pop()
13       PhantomizeNode(node,Collector)
14       Collector.RecoveryList.append(node)
15     EndWhile
16     While Collector.RecoveryList.size() > 0
17       Let node = Collector.RecoveryList.pop()
18       RecoverNode(node)
19       Collector.CleanList.append(node)
20     EndWhile
21     While Collector.RebuildList.size() > 0
22       Let node = Collector.RebuildList.pop()
23       RecoverNode(node)
24     EndWhile
25     While Collector.CleanList.size() > 0
26       Let node = Collector.CleanList.pop()
27       CleanNode(node)
28     EndWhile
29   EndWhile
```

Algorithm 6: PhantomizeLink

```
 1 PhantomizeLink(link):
 2   lock(link.Source,link.Target)
 3   If link.Target == NULL then
 4     unlock()
 5     Return
 6   EndIf
 7   If link.Phantomized then
 8     unlock()
 9     Return
10   EndIf
11   link.Target.PhantomCount++
12   link.Phantomized = True
13   linkFree(link)
14   MergeCollectors(link.Source, link.Target)
15   unlock()
```

Algorithm 7: DecPhantom

```
   // DecPhantom is responsible for removing any reference to the collector.
 1 DecPhantom(node):
 2   lock(node)
 3   node.PhantomCount- -
 4   If node.PhantomCount == 0 then
 5     If node.Count[node.Which]== 0 And
 6           node.Count[1-node.Which] == 0 then
 7       Delete(node)
 8     Else
 9       node.Collector = NULL
10     EndIf
11   EndIf
12   unlock()
```

Algorithm 8: RecoverNode

```
 1 RecoverNode(node):
 2   lock(node)
 3   Let links = NULL
 4   If node.Count[node.Which] > 0 then
 5     WaitFor(node.PhantomizationComplete == True)
 6     node.Phantomized = False
 7     links = copy(node.Links)
 8   EndIf
 9   unlock()
10   ForEach link in links
11     Rebuild(link)
12   EndFor
```

Algorithm 9: Rebuild

```
 1 Rebuild(link):
 2   lock(link.Source,link.Target)
 3   If link.Phantomized then
 4     If link.Target == link.Source then
 5       link.Which = 1- link.Target.Which
 6     ElseIf link.Target.Phantomized then
 7       link.Which = link.Target.Which
 8     ElseIf count(link.Target.Links) == 0 then
 9       link.Which = link.Target.Which
10     Else
11       link.Which = 1-link.Target.Which
12     EndIf
13     link.Target.Count[link.Which]++
14     link.Target.PhantomCount- -
15     If link.Target.PhantomCount == 0 then
16       link.Target.Collector = NULL
17     EndIf
18     link.Phantomized = False
19     Add link.Target to Collector.RecoveryList
20   EndIf
21   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.
 1 CleanNode(node):
 2   lock(node)
 3   Let die = False
 4   If node.Count[node.Which]== 0 And
 5         node.Count[1-node.Which]== 0 then
 6     die = True
 7   EndIf
 8   unlock()
 9   If die then
10     ForEach link in node
11       LinkFree(link)
12     EndFor
13   EndIf
14   DecPhantom(node)
```

Algorithm 11: Delete

```
1 Delete(node):
2     ForEach link in node
3       LinkFree(link)
4     EndFor
5     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.
 1 MergeCollectors(source,target):
 2   Let s = source.Collector
 3   Let t = target.Collector
 4   Let done = False
 5   If s == NULL And t != NULL then
 6     lock(source)
 7     source.Collector = t
 8     unlock()
 9     Return
10   EndIf
11   If s != NULL And t == NULL then
12     lock(target)
13     target.Collector = s
14     unlock()
15     Return
16   EndIf
17   If s == NULL Or s == NULL then
18     Return
19   EndIf
20   While Not done
21     lock(s,t,target,source)
22     If s.Forward == t and t.Forward == NULL then
23       target.Collector = s
24       source.Collector = s
25       done = True
26     ElseIf t.Forward == s and s.Forward == NULL then
27       target.Collector = t
28       source.Collector = t
29       done = True
30     ElseIf t.Forward != NULL then
31       t = t.Forward
32     ElseIf s.Forward != NULL then
33       s = s.Forward
34     Else
35       Transfer s.CollectionList to t.CollectionList
36       Transfer s.MergedList to t.MergedList
37       Transfer s.RecoveryList to t.MergedList
38       Transfer s.RebuildList to t.RebuildList
39       Transfer s.CleanList to t.MergedList
40       target.Collector = t
41       source.Collector = t
42       done = True
43     EndIf
44     unlock()
45   EndWhile
```