2016

Garbage Collection for General Graphs

Hari Krishnan
Louisiana State University and Agricultural and Mechanical College, hari1106@gmail.com

GARBAGE COLLECTION FOR GENERAL GRAPHS

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The School of Electrical Engineering & Computer Science
The Division of Computer Science and Engineering

by
Hari Krishnan
B.Tech., Anna University, 2010
August 2016

Dedicated to my parents, Lakshmi and Krishnan.

Acknowledgments

I am very grateful to my supervisor Dr. Steven R. Brandt for giving the opportunity to fulfill my dream of pursuing Ph.D in computer science. His advice, ideas and feedback helped me understand various concepts and get better as a student. Without his continuous support and guidance, I wouldn't have come so far in my studies and research. Our discussions were very effective and motivating.

I would also like to express my sincere gratitude to Dr. Costas Busch. His expertise in the area, insights, motivation to direct the research in theoretical way, and advices helped me to be a better researcher. I would like to thank Dr. Gerald Baumgartner for his constructive feedbacks, advices, and interesting conversations about my work.

I would also like to thank Dr. Gokarna Sharma for useful feedbacks, constant help and collaborations. I would like to thank my family for their constant moral support. It is my pleasure to thank my cousin Usha who introduced me to computers and taught me more about very early age. I would also like to thank my friends Sudip Biswas and Dennis Castleberry, who motivated me when required and helped me to focus on my graduate school at all times.

Table of Contents

List of Figures

Abstract

Garbage collection is moving from being a utility to a requirement of every modern programming language. With multi-core and distributed systems, most programs written recently are heavily multi-threaded and distributed. Distributed and multi-threaded programs are called concurrent programs. Manual memory management is cumbersome and difficult in concurrent programs. Concurrent programming is characterized by multiple independent processes/threads, communication between processes/threads, and uncertainty in the order of concurrent operations.

The uncertainty in the order of operations makes manual memory management of concurrent programs difficult. A popular alternative to garbage collection in concurrent programs is to use smart pointers. Smart pointers can collect all garbage only if developer identifies cycles being created in the reference graph. Smart pointer usage does not guarantee protection from memory leaks unless cycle can be detected as process/thread create them. General garbage collectors, on the other hand, can avoid memory leaks, dangling pointers, and double deletion problems in any programming environment without help from the programmer.

Concurrent programming is used in shared memory and distributed memory systems. State of the art shared memory systems use a single concurrent garbage collector thread that processes the reference graph. Distributed memory systems have very few complete garbage collection algorithms and those that exist use global barriers, are centralized and do not scale well. This thesis focuses on designing garbage collection algorithms for shared memory and distributed memory systems that satisfy the following properties: concurrent, parallel, scalable, localized (decentralized), low pause time, high promptness, no global synchronization, safe, complete, and operates in linear time.

Chapter 1
Introduction

1.1 Garbage Collection

A processor, an algorithm (software), and memory are the trio of computation. Memory is organized by programming languages and developers to control the life of data. In this thesis, we abstract all data allocated in memory as objects. In general, we define an object as any block of memory that saves information and may contain address of other objects. Most programming languages support dynamic memory allocation. Programming languages divide available memory into three types: stack, static and heap memory. Static memory contains all the global variables used by the program. Stack memory is used to manage static (fixed size) allocation of objects whose scope is well defined. Heap memory is used to manage dynamically allocated objects whose scope cannot be determined at compile time.

The amount of static memory used is computed at the compilation phase and objects allocated in static memory is never deleted. Stack memory is used to manage function calls and the allocation of objects in a function's scope. So the size of the stack memory required for the execution cannot be computed at compilation time. Stack memory is filled from the lower address space of memory and heap memory is filled from the higher address space of memory. When two kinds of memory meet, the program usually crashes.

Stack memory is used to store all objects that are actively in use or will be used in the near future. Heap memory is used to store all objects whose scope is unknown to the program. Memory management is the term used to describe management of the heap. When developers explicitly delete objects that are no longer in use, it is called manual memory management.

There are three issues that happen when the heap is manually managed. They are dangling pointers, double-free bugs, and memory leaks. All of the three issues are deadly and harmful. Some of them produce wrong output, deletes memory that is in use, and also inefficiently organizes the heap memory. Apart from all the above-mentioned problems, it is extremely difficult to manually manage memory for a certain class of programs called concurrent programs. Automatic memory management avoids all the above-mentioned problems. This thesis focuses on designing efficient algorithms for automatic memory management. The process of automatically managing memory is termed as **Garbage Collection (GC)**.

1.1.1 How GC's work

The term garbage refers to the objects allocated in the heap that are not used anymore. All objects allocated in memory can be referenced by their address. An object x is said to hold a reference to object y when x contains the address of y. An object is said to be in use if it is reachable through a chain of references starting from stack or static memory. The objects in stack and static memory that are considered for defining the whether an object is in use are referred to as **roots**. When an object is no longer in use, it is called a garbage object. Dead is an alternative adjective used to describe a garbage object. Garbage collection is the process of collecting all dead objects in the heap memory.

There are two major techniques to identify garbage objects. They are tracing and reference counting. Tracing involves marking all the objects in the heap that are in use by going through all the chains of references from the roots. Once the marking is done, all unmarked objects in the heap are considered to be garbage and deleted by the garbage collection process. Reference counting attaches a counter to each object that counts the number of incoming references. When the counter reaches zero, the object is garbage. All available garbage collection algorithms are some combination of these two techniques.

1.2 Motivation and Objectives

Developers are using programming languages with automatic memory management for several benefits including no dangling pointers, no double-free bugs, no memory leaks, high productivity, and less code to write [13].

Dangling Pointers: Dangling pointers occur when objects are deleted, but the addresses pointing to those objects are not cleared. These references are very dangerous as one does not know what the reference points to. If the reference points to some other object, then computation might yield an undesired output or may crash because of a mismatch in type information with new object allocated in the same address.

Double-free bugs: Double free bugs is an issue that happens when an object allocated in the heap is deleted at some point in time and programmer deletes the object again at the same address. It is very risky and can lead to dangling pointers in some cases. If no new objects are created at the same address, then the delete might crash the program. In other extreme case where a new object is allocated at the same address, it deletes the object and creates dangling pointers for all the objects that hold a reference to the deleted object.

Memory leaks: Memory leaks occur when objects are not deleted, but the address pointing to those objects are cleared. The unused objects are not deleted and they consume memory which could be reused for other objects. This inefficient organization of heap can affect the runtime of an application heavily. This affects the memory allocator as there will be less memory available to allocate and program crashes when there is no more memory available to allocate memory for more objects.

The advantages of automatic memory management is important for successful execution of any application. This thesis focuses on concurrent programs and automatic memory management requirement in concurrent programs.

Concurrent program is a term used to describe multiple threads / processes running simultaneously to perform some computation. There are two different environments where concurrent programs are used: shared memory and distributed memory systems. Shared memory systems contain multiple processors / multi-core processors sharing a common memory and execute the computation through threads / processes. Threads communicate using shared memory and may use atomic instructions or traditional locks to execute some critical sections of the program. In a shared memory environment, concurrent programmers use smart pointers to manually manage memory. Smart pointers are a manual reference counting technique implemented by the application programming interface. Distributed memory systems contain multiple processors with dedicated memory and processors are separated by some physical distance and connected via a network. In distributed memory systems, processes execute on each processor and communicate by sending messages across the network to perform the computation. Both of the environments contain programs that run concurrently and require some form of communication to perform the desired computation. The environments are distinguished by communcation technique and memory use. Both of the environments uses automatic memory management to help developers build applications with the above-mentioned benefits.

When concurrent programs are written by developers, manual memory management is extremely difficult and error-prone. When multiple threads access a common object, dangling pointers are a common scenario due to the incomplete information about ownership. Double-free bugs can occur just as frequently as the dangling pointers. Smart pointers are the defacto standard to avoid memory management issues in concurrent programs. These smart pointers do not guarantee complete garbage collection. The use of cyclic objects in heap memory requires extreme care in a manually managed concurrent program. It is carefully avoided

by all programmers usually to make the program memory leak free. The above reasons explain difficulties of automatic memory management in the concurrent programming environment and the necessity for high-quality automatic memory management in concurrent programming environments.

Beyond solving the garbage collection problem in a concurrent programming environment, the objectives of this thesis are to design garbage collectors for shared and distributed memory system that satisfy the following properties :

1. Concurrent garbage collectors (Less pause time)
2. Multi-collector garbage collection (Parallel)
3. No global synchronization (High Throughput)
4. Locality-based garbage collection
5. Scalable
6. Prompt
7. Safety
8. Complete

There are no garbage collection algorithms available for shared and distributed memory systems that satisfy all the above-mentioned properties. This thesis focuses on designing a novel garbage collector with significant improvements. Apart from garbage collection, the ideas presented in the thesis will be of use to solve other theoretical problems in distributed computing like dynamic reachability, breaking cycles, detecting cycles, data aggregation in a sensor network, broken network detection in a sensor networks.

Concurrent: Concurrent garbage collectors work simultaneously with the application. The application will not experience any pauses during execution to scan memory. They have negligible to zero pause time. There are a handful of collectors that satisfy this property.

Multi-collector Garbage Collection : Multi-collector garbage collectors are collectors that have multiple independent garbage collector threads / processes that work independently on the heap to identify garbage.

These collectors utilize multiple processors when compute resources are being underutilized. In shared memory systems, when multiple processors are underutilized, this property helps to utilize processor cycles better. Apart from the processor utilization, the throughput of garbage collectors can be increased by multi-collector garbage collectors. These multi-collector garbage collector threads / processes communicate among themselves through memory in shared memory systems and through messages in distributed memory systems.

Global Synchronization: Multi-collector garbage collectors are common in distributed memory systems. When multi-collector garbage collectors are used, conventional solutions require a global barrier among all the collectors to communicate and share the information to identify garbage objects. With no global synchronization, the throughput of garbage collectors will be high and garbage collector algorithm scales well with this property.

Locality-based garbage collection: Detecting garbage is a global predicate. Traditionally, popular methods of garbage collection involve computing garbage as the global predicate. This requires scanning the entire heap for precise evaluation of the predicate. If garbage object can be detected locally based on the small set of objects, then the garbage detection process can save a lot of time spent by not tracing all objects in the heap.

Scalable: Shared memory and distributed memory systems does not have a scalable garbage collectors up until now. In shared memory systems, scalability is an issue due to most collectors designed are primarily single collector concurrent algorithm. In distributed system, all available garbage collectors require a form of global synchronization to detect garbage. To meet future demands of scalable garbage collection, collectors must be multi-collector locality-based with no global synchronization.

Promptness: With global garbage collection, the cost of scanning entire heap often is very expensive. To reduce the high price for global scanning of the heap, garbage collection is initiated when the threshold is met. The promptness property helps to keep heap memory free of floating garbage in a timely manner. This property helps to avoid global scanning and also quick availability of free memory for allocation.

Safety: The safety property is the most crucial part of garbage collection. This avoids any dangling pointers and double free bugs. The property guarantees that every garbage collector will delete only garbage objects. While most garbage collector in literature satisfies this property, there are some that cannot satisfy this property. This thesis requires garbage collector to be safe in the multi-collector environment which is challenging given that there is no global synchronization. In a distributed memory system, this property is very difficult to satisfy as the garbage collector cannot get the global snapshot of a heap at any point in time. A locality-based algorithm works by using information obtained only from neighbors. Evaluating a global predicate using local information is challenging by all means. This thesis guarantees the safety of collectors designed for both environments.

Completeness: Completeness guarantees that a garbage collector will collect all garbage objects in the heap, thereby removing all memory leaks in the heap. Reference counting garbage collectors are well known for incomplete garbage collection due to their inability to collect cyclic garbage. This property is challenging as the locality-based collectors usually use reference counting. Collectors designed for both environments use reference counting and tracing techniques to solve the problem and guarantees the garbage collectors are complete.

1.3 Contributions

The contributions of this thesis are listed below :

1. Novel hybrid reference counting and tracing techniques are used to collect any garbage including cyclic garbage.

2. First known concurrent multi-collector shared memory garbage collector.

3. Well known issues in Brownbridge garbage collectors are fixed and complete and a safe garbage collector is designed.

4. Theoretically, our shared memory garbage collector detects garbage in fewer traversals than other state of the art concurrent reference counted garbage collector.

5. This thesis presents the first known locality-based concurrent multi-collector scalable distributed garbage collector with no global synchronization. The thesis contains a complete proof of the garbage collector.

6. The distributed garbage collector introduces a novel weight based approach to convert the graph into a directed acyclic graph and thereby detects cycles faster.

7. The designed garbage collector algorithms finishes garbage collection in linear time.

1.4 Dissertation Organization

The preliminary ideas required to understand the thesis are presented in chapter 2. Chapter 2 introduces the abstract version of the garbage collection problem, a brief literature review of garbage collectors in general, and in detail, it explains Brownbridge garbage collector and explains the failures of Brownbridge garbage collector. The understanding of Brownbridge is essential for this thesis as our algorithms correct and extend the approach. Chapter 3 contains a brief introduction of shared memory garbage collectors and related works of the particular class of garbage collectors used in shared memory systems. Shared memory single collector and multi-collector algorithms are described in chapter 3. Apart from describing the algorithm, chapter also contains the proofs of safety and completeness with simulated experimental results. The chapter also proves the linearity in the number of operations to detect garbage. Chapter 4 explains the overview of distributed garbage collection, existing algorithms and their issues. Single collector and multi-collector distributed garbage collector algorithms are explained in abstract and a novel technique is introduced to prove termination of garbage collection, safety, completeness and time complexity using the isolation property. The chapter also includes the pseudocode of the algorithm. The distributed garbage collector with scalability and other guarantees mentioned above is a theoretical leap. Chapter 5 captures the overall contributions of the thesis in broad view with possible future work.

Chapter 2
Preliminaries

2.1 Abstract Graph Model

Basic Reference Graph: We model the relationship among various objects and references in memory through a directed graph $G = (V, E)$, which we call a *reference graph*. The graph G has a special node R, which we call the *root*. Node R represents global and stack pointer variables, and thus does not have any incoming edges. Each node in G is assumed to contain a unique ID. All adjacent nodes to a given node in G are called $neighbors$, and denoted by Γ. The $in\text{-}neighbors$ of a node $x \in G$ can be defined as the set of nodes whose outgoing edges are incident on x, represented by $\Gamma_{in}(x)$. The $out\text{-}neighbors$ of x can be defined as the set of nodes whose incoming edges originate on x, represented by $\Gamma_{out}(x)$. Note that each node $x \in G$ does not know $\Gamma_{in}(x)$ at any point in time.

Garbage Collection Problem: All nodes in G can be classified as either *live* (i.e., not garbage) or *dead* (i.e., garbage) based on a property called $reachability$. Live and dead nodes can be defined as below:

$Reachable(y, x) = x \in \Gamma_{\text{out}}(y) \vee (x \in \Gamma_{\text{out}}(z) \mid Reachable(y, z))$

$Live(x) = Reachable(R, x)$

$Dead(x) = \neg Live(x)$

We allow the live portion of G, denoted as G', to be mutated while the algorithm is running, and we refer to the source of these mutations as the *Adversary (Mutator).* The Adversary can create nodes and attach them to G', create new edges between existing nodes of G', or delete edges from G'. Moreover, the Adversary can perform multiple events (creation and deletion of edges) simultaneously. The Adversary, however, can never mutate the dead portion of the graph $G'' = G \backslash G'$.

Axiom 1 (Immutable Dead Node)**.** The Adversary cannot mutate a dead node.

Axiom 2 (Node Creation)**.** All nodes are live when they are created by the Adversary.

From Axiom 3, it follows that a node that becomes dead will never become live again.

Each node experiencing deletion of an incoming edge has to determine whether it is still live. If a node detects that it is dead then it must delete itself from the graph G.

Definition 2.1.1 (Garbage Collection Problem)**.** Identify the dead nodes in the reference graph G and delete them.

2.2 Literature Review

McCarthy invented the concept of Garbage Collection in 1959 [48]. His works focused on an indirect method of collecting garbage which is called Mark Sweep or tracing collector. Collins designed a new method called reference counting, which is a more direct method to detect garbage [16]. At the time when garbage collectors were designed, several of them could be used in practice due to bugs in the algorithms. Dijkstra et al [17] modeled a tricolor tracing garbage collector that provided correctness proofs. This work redefined the way the collectors should be designed and how one can prove the properties of the collectors.

Haddon et al [23] realized the fragmentation issues in the Mark Sweep algorithm and designed a Mark Compact algorithm where objects are moved when traced and arranged in order to accumulate all the free space together. Fenichel et al [18] and Cheney [14] provided new kind of algorithm called a copying collector based on the idea of live objects. Copying collectors divide memory into two halves and the collector moves live objects from one section to the other section and cleans the other section. This method uses only half of the heap at any given time and requires copying the object which is very expensive. Based on Foderaro and Fateman [19] observation, 98% of the objects collected were allocated after the last collection. Several such observations were made and reported in [69, 62]. Based on the above observations, Appel designed a simple generation garbage collection [2]. Generational garbage collection utilizes idea from the all classical method mentioned above and provides a good throughput collector. It divides the memory into at least two generation. The old generation occupies a large portion of the heap and the young generation takes a small portion of the heap. It exploits the weak generational hypothesis which says young objects die soon. In spite of considerable overhead due to inter-generational pointers, promotions, and full heap collection once in a while, this garbage collection technique dominates alternatives in terms of performance and is widely used in many run-time systems.

While most of the methods considered until now are an indirect method of collecting garbage through tracing, the other form of collection is reference counting. McBeth's [46] article on the inability to collect cyclic garbage by reference counting made the method less practical. Bobrow's [9] solution to collect cyclic garbage based on reference counting was the first hybrid collector which used both tracing and counting the

references. Hughes [27, 28] identified the practical issues in the Bobrow's algorithm. several attempts to fix the issues in the algorithm appeared subsequently, e.g. [21, 9, 39]. In contrast to the approach followed in [21, 9, 39] and several others, Brownbridge [12] proposed, in 1985, an algorithm to tackle the problem of reclaiming cyclic structures using strong and weak pointers [30]. This algorithm relied on maintaining two invariants: (a) there are no cycles in strong pointers and (b) all items in the object reference graph must be strongly reachable from the roots.

A couple of years after the publication, Salkild [61] showed that Brownbridge's algorithm [12] 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 [61] 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. [53] 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 marks: 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 [30]. 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 collection systems, e.g. [5, 36, 3, 6, 39], 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. Trial deletion approach was studied by Christopher [15] which tries to collect cycles by identifying groups of self-sustaining objects. Lins [37] used a cyclic buffer to reduce repeated scanning of the same nodes in their Mark Scan algorithm for cyclic reference counting. Moreover, in [38], Lins improved his algorithm from [37] 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 [36]. Recent work in reference counting algorithms, e.g. [6, 36, 5, 3], try to reduce concurrent operations and increase the efficiency of reference counting collectors. However, as mentioned earlier, reference counting garbage collectors cannot collect cycles [46]. Therefore, concurrent reference counting collectors [6, 36, 5, 3, 51, 39] 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 [51] combines the sliding view reference counting concurrent collector of [36] with the cycle collector of [5]. Recently, Frampton provides a detailed study of cycle collection in his Ph.D. thesis [20].

Apple's ARC memory management system makes a distinction between "strong" and "weak" pointers, similar to what Brownbridge describes. 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. [58]. Huelsbergen and Winterbottom [25] proposed an incremental algorithm for the concurrent garbage collection that is a variant of Mark Sweep collection scheme first proposed in [49]. Furthermore, garbage collection is also considered for several other systems, namely real-time systems and asynchronous distributed systems, e.g. [55, 67]. Concurrent collectors are gaining popularity. The concurrent collector described in Bacon and Rajan [5] can be considered to be one of the most 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 the cycle when the cycle is reachable and eleven cycle traversals when the cycle is garbage.

Distributed systems require garbage collectors to reclaim memory. Distributed Garbage Collector(DGC) requires an algorithm to be designed in a different way than the other approaches. Most of the algorithm discussed above cannot be applicable to the distributed setting. Bevan[8] proposed the use of Weighted

Reference Counting as an efficient solution to DGC. Each reference has two weights: a partial weight and a total weight. Every reference creation halves the partial weight. The drawback of this method is an application cannot have more than 32 or 64 references based on the architecture. Piquer[54] suggests an original solution to the previous approach. Both of the above-mentioned methods use straight reference counting, so they cannot reclaim cyclic structures and they are not resilient to message failures. Shapiro et al [64] provided an acyclic truly distributed garbage collector. Although this method is not cyclic distributed garbage collector, the work lays a foundation for a model of distributed garbage collection. A standard approach to a distributed tracing collector is to combine independent local, per-space collectors, with a global inter-space collector. The main problem with distributed tracing is to synchronize the distributed mark phase with independent sweep phase[56]. Hughes designed a distributed tracing garbage collector with a time-stamp instead of mark bits[26]. The method requires the global GC to process nodes only when the application stops its execution or requests a stop when collecting. This is not a practical solution for many systems. Several solutions based on a centralized collector and moving objects to one space are proposed[44, 42, 40, 33, 65]. Recently, a mobile actor-based distributed garbage collector was designed[68]. Mobile actor DGCs cannot be easily implemented as an actor needs access to a remote site's root set and actors move with huge memory payload. All of the distributed garbage collection solutions have one or more of the problems: inability to detect cycles, moving objects to one site for cycle detection, weak heuristics to compute cycles, no fault-tolerance, centralized solution, stopping the world, not scalable, and failure to be resilient to message failures.

2.3 Cycle detection using Strong-Weak

Brownbridge classifies the edges in the reference graph into strong and weak. The classification is based on the invariant that there is no cycle of strong edges. So every cycle must contain at least one weak edge. The notion of the strong edge is connected to the liveness of the node. So the existence of a strong incoming edge for a node indicates that the node is live.

Strong edges form a directed acyclic graph in which every node is reachable from R. The rest of the edges are classified as weak. This classification is not simple to compute as they themselves might require complete scanning of the graph. To avoid complete scanning, some heuristic approaches are required. The identification of cycle creating edges are the bottleneck of the classification problem. The edges are labeled weak if it is created to a node that already exists in the memory. This heuristic guarantee that all cycles

contain at least one weak edge. But a node with weak incoming edge does not mean it is part of a cycle. The heuristic helps the mutator to save time in classifying the edges. But the collectors require more information about the topology of the subgraph to identify whether the subgraph is indeed garbage.

The heuristic idea mentioned above prevents the creation of strong cycles. The other technique required by the Brownbridge algorithm is the ability to change the strong and weak edges whenever required by the nodes. To solve this issue, Brownbridge used two boolean values, one for each edge in the source and one for each node. For an edge to be strong, both the boolean values have to be same. So if a node wants to change the type of its edges, a node changes the corresponding boolean value. A node will change all the incoming edge types when it changes its boolean value. By changing the boolean value for an outgoing edge, a node changes the type of the edge. The idea of toggling the edge types is an important piece of the Brownbridge algorithm.

2.4 Brownbridge Garbage Collector

Brownbridge's algorithm works for a system where the Adversary stops the world for the garbage collection process. Each node contains two counters each corresponding to the type of incoming edge. When the graph mutations are serialized, any addition / deletion of edge increments / decrements the corresponding counters in the target node. An addition event never makes a node garbage. A deletion event might make a node garbage. If the edge deleted is weak, no further actions are taken except decrementing the counter. Only a last strong incoming edge deletion to a node performs additional work. When the last strong incoming edge is deleted and the counter is decremented, the node is considered garbage if there are no weak incoming edges. If a node contains weak incoming edges after last strong incoming edge deletion, then the algorithm performs a partial tracing to identify if it is cyclic garbage. Tracing starts by converting all the weak incoming edges of the node into strong by toggling the boolean value. Once the boolean value is changed, the node swaps the counters and identifies all the incoming edges as strong. To determine whether the subgraph is cyclic garbage, the node weakens all its outgoing edges. The node marks itself that it has performed a transformation. By weakening the outgoing edges, we mean converting all the outgoing edges of a node into weak edges. Some nodes might lose all strong incoming edges due to this process. This propagates the weakening process. When a marked node receives a weakening message it will decide it is garbage and deletes itself. Every

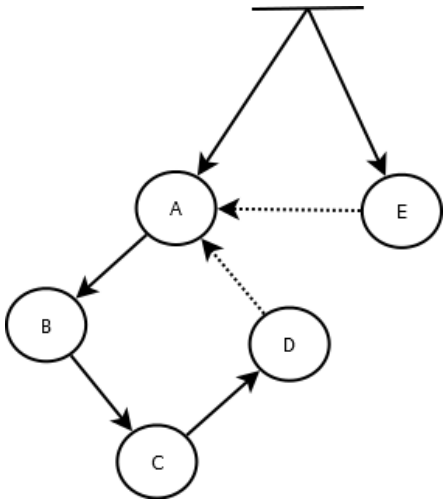

FIGURE 2.1. When root R deletes strong edge to A, through partial tracing A will have strong edge at the end.

weakening call propagates by switching the weak incoming edges into strong. This method determines if there are any external references pointing to the node that lost the strong incoming edge.

Figure 2.1 shows an example graph where the Brownbridge technique will identify the external support to a node and restore the graph. In figure 2.1, if the strong edge to A is deleted, A will convert all incoming weak edges into strong and start weakening the outgoing edges. When B receives a weaken message, it weakens the outgoing edge and then converts all weak incoming edges to strong. The process continues until it reaches A again. When the process reaches A, A realizes now it has an additional strong incoming edge. So the graph is recovered now. If there is no edge from E to A, then the whole subgraph A, B, C and D will be deleted.

2.5 Pitfalls of Brownbridge Garbage Collection

The idea behind the Brownbridge method is to identify external support for the node that lost the last strong incoming edge. If the external strong support to the subgraph comes from a node that did not loose its last strong incoming edge, then the method fails to identify the support and prematurely deletes nodes.

In figure 2.2, when the root deletes the edge to A, A starts partial tracing. When the partial tracing reaches back to A, D has additional strong support. But the algorithm only makes a decision based on A's strong incoming edge. So the A, B, C, D subgraph will be deleted prematurely.

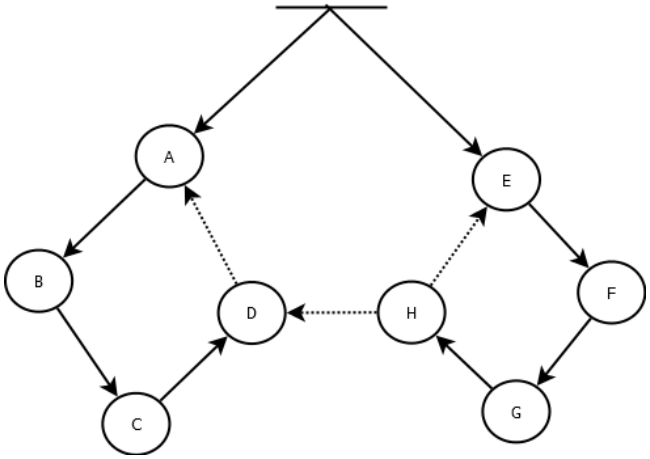

FIGURE 2.2. When root R deletes strong edge to A, through partial tracing A will fail to identify liveness.

Chapter 3
Shared Memory Multi-processor Garbage Collection

3.1 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 [46, 5, 3, 6, 30, 49, 16]. Despite the many advances in garbage collection, there are problem areas which have difficulty benefiting, such as distributed or real-time systems; see [4, 55, 67]. 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 [12]. 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 [61], and subsequent attempts to improve it introduced poor performance (at least exponential cleanup time in the worst-case) [53]; details in Section 3.1.1. 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. [60] 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. [5, 36, 3, 6, 39], 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

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

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 [12, 30].

Many garbage collectors in the literature employ a generational technique, for example the generational collector proposed in [58], 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, 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. 3.1, 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 [12, 61, 53] which can not handle concurrency issues that arise in modern multiprocessor architectures [60].

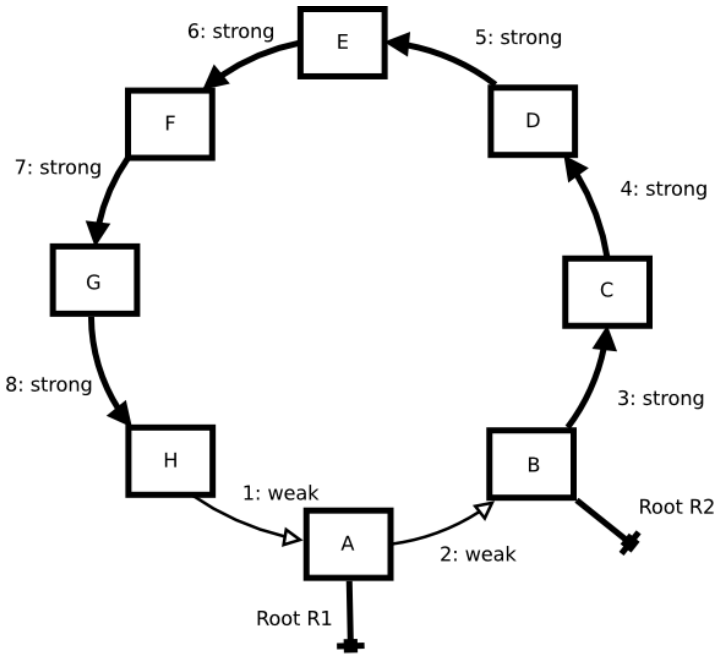

FIGURE 3.1. When root R1 is removed, the garbage collection algorithm only needs to trace link 2 to prove that object A does not need to be collected.

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.

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

Before giving details of our algorithm in Section 3.2 – which in turn is a modification of Brownbridge's algorithm and its variants [12, 61, 53] – 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 [30].

It was proven by McBeth [46] in early sixties that reference counting collectors were unable to handle cyclic structures; several attempts to fix this problem appeared subsequently, e.g. [21, 9, 39]. We give details in Section 3.1.2. In contrast to the approach followed in [21, 9, 39] and several others, Brownbridge [12] 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) [30]. 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 [61] showed that Brownbridge's algorithm [12] 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 [61] 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. [53] 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 [30]. 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.

3.1.2 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., [46, 12, 61, 53, 5, 3, 6, 30]. 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 [5]. Reference counting approaches were first

developed for LISP by Collins [16]. Improved variations were proposed in several subsequent papers, e.g. [21, 29, 30, 39, 36]. We direct readers to Shahriyar et al. [63] for the valuable overview of the current state of reference counting collectors.

It was noticed by McBeth [46] in early sixties that reference counting collectors were unable to handle cyclic structures. After that several reference counting collectors were developed, e.g. [21, 9, 37, 38]. The algorithm in Friedman [21] dealt with recovering cyclic data in immutable structures, whereas Bobrow's algorithm [9] can reclaim all cyclic structures but relies on the explicit information provided by the programmer. Trial deletion approach was studied by Christopher [15] which tries to collect cycles by identifying groups of self-sustaining objects. Lins [37] used a cyclic buffer to reduce repeated scanning of the same nodes in their mark-scan algorithm for cyclic reference counting. Moreover, in [38], Lins improved his algorithm from [37] 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 [36]. Recent work in reference counting algorithms, e.g. [6, 36, 5, 3], 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 [46]. Therefore, concurrent reference counting collectors [6, 36, 5, 3, 51, 39] 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 [51] combines the sliding view reference counting concurrent collector of [36] with the cycle collector of [5]. 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 [20].

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. [58]. Huelsbergen and Winterbottom [25] proposed an incremental algorithm for the concurrent garbage collection that is a variant of mark-and-sweep collection scheme first proposed in [49]. Furthermore, garbage collection is also considered for several other systems, namely real-time systems and asynchronous distributed systems, e.g. [55, 67].

Concurrent collectors are gaining popularity. The concurrent collector described in Bacon and Rajan [5] 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.

3.1.3 Paper Organization

The rest of the paper is organized as follows. We present our strong/weak/phantom pointer based concurrent garbage collector in Section 3.2 with some examples. In Section 3.4, we sketch proofs of its correctness and complexity properties. In Section 3.5, we give some experimental results. We conclude the paper with future research directions in Section 3.6 and a short discussion in Section 3.7. Detailed algorithms may be found in the appendix.

3.2 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 [12] 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. 3.2.

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 2 in Section 3.4. 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.

3.2.1 Example: A Simple Cycle

In Fig. 3.2 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.

3.2.2 Example: A Doubly-Linked List

The doubly linked list depicted in Fig. 3.3 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. 3.2, which is necessary for a more complex figure.

3.2.3 Example: Rebalancing A Doubly-Linked List

Fig. 3.4 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.

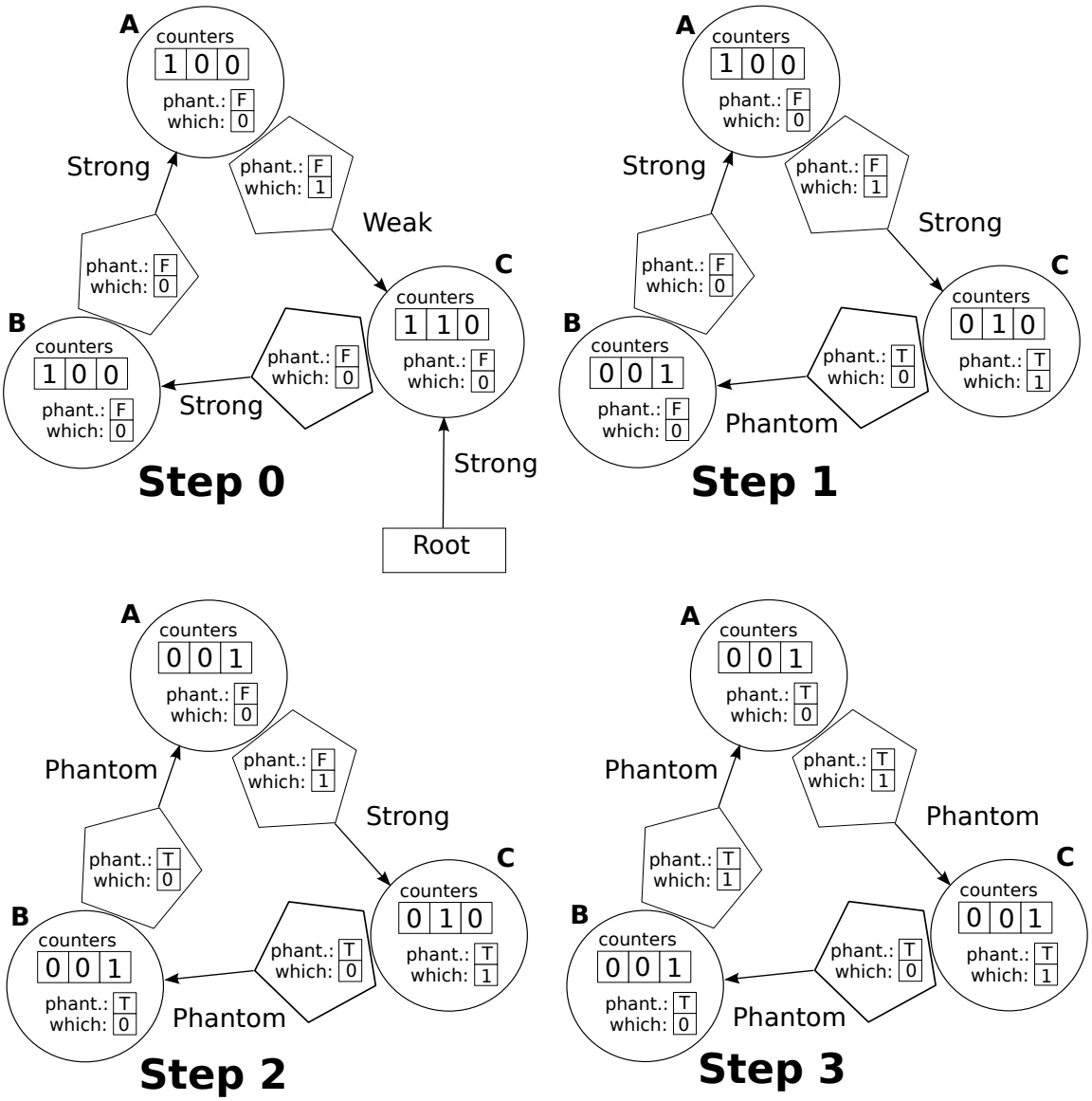

FIGURE 3.2. Reclaiming a cycle with three objects

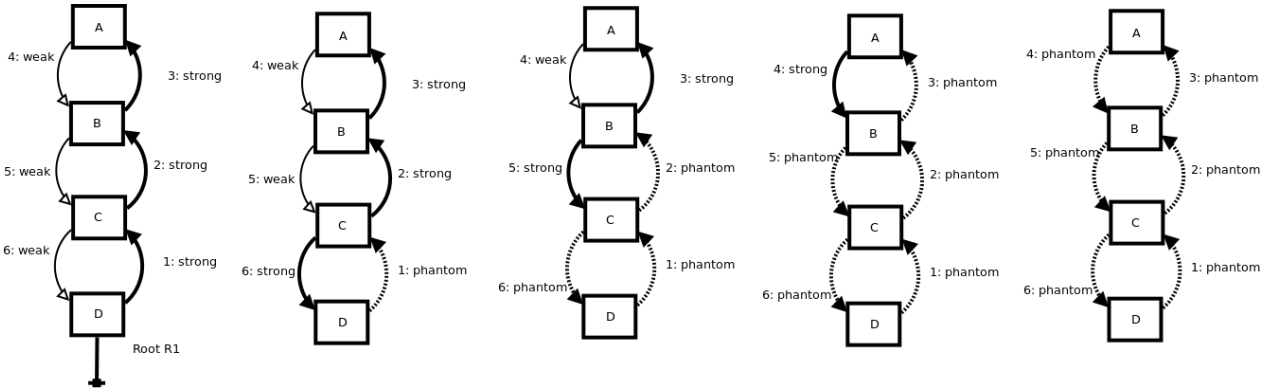

FIGURE 3.3. Doubly-linked list

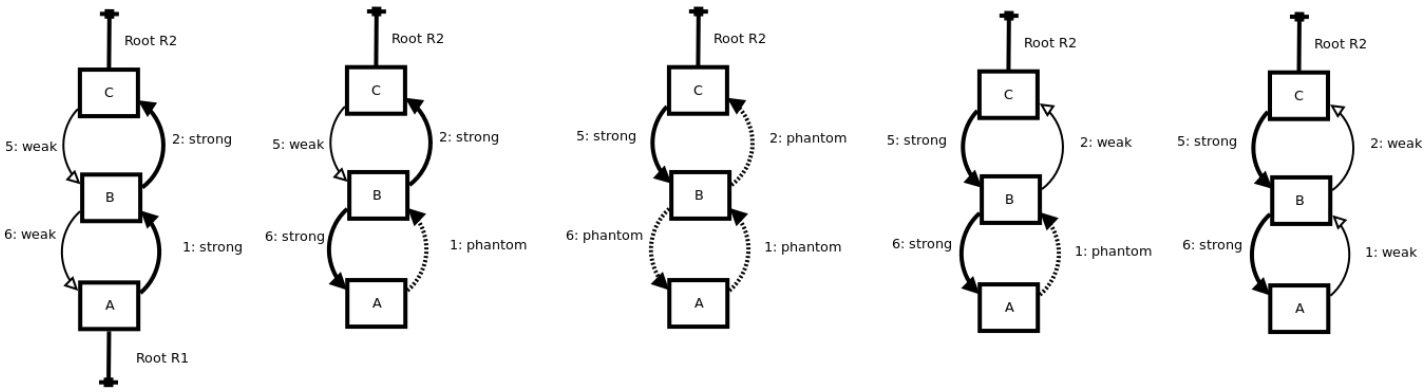

FIGURE 3.4. Rebalancing a doubly-linked list

$Phantomization$ 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.

3.2.4 Example: Recovering Without Detecting a Cycle

In Fig. 3.1 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.

3.3 Concurrency Issues

This section provides details of the implementation.

3.3.1 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.

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.

3.3.2 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.

3.4 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

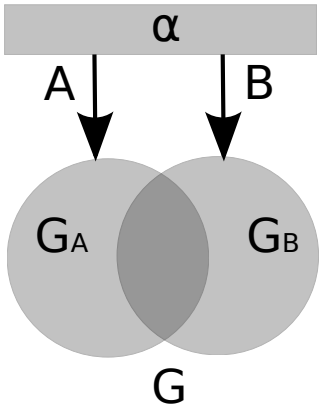

FIGURE 3.5. Graph model

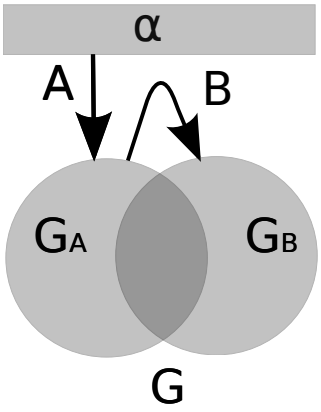

FIGURE 3.6. Subgraph model

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. 3.5), $\alpha = \{A, B\}$, 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. 3.6.

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 (sketch). 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 (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. 3.5 (or, equivalently, Fig. 3.6). 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. 3.5 (or Fig. 3.6), 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 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 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 (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 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.

3.5 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. 3.7 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. 3.8 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. 3.9, 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.

3.6 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

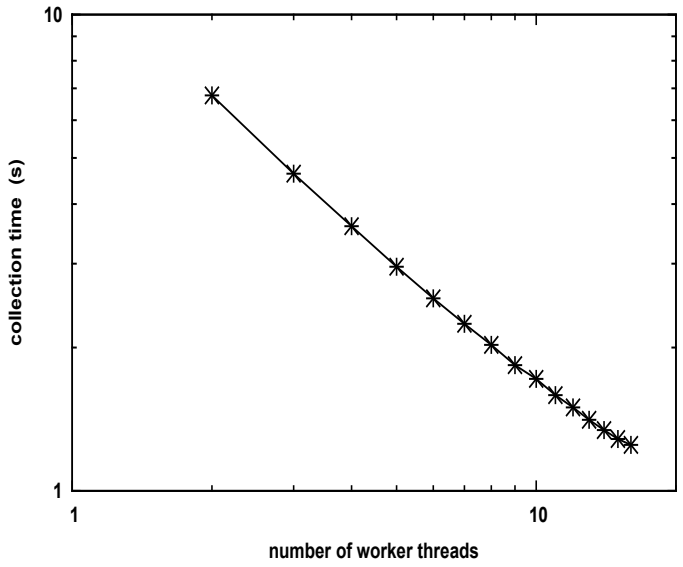

FIGURE 3.7. A large number of independent rings are collected by various number of worker threads. Collection speed drops linearly with the number of cores used.

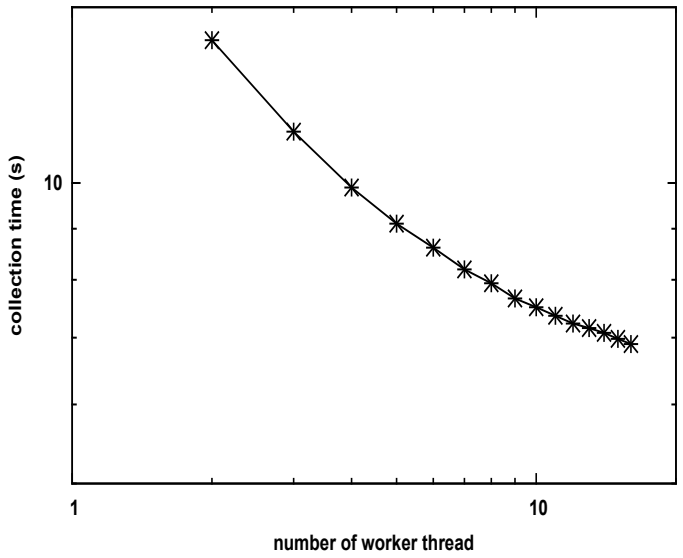

FIGURE 3.8. A chain of linked cycles is created in memory. The connections are severed, then the roots are removed. Multiple collector threads are created and operations partially overlap.

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

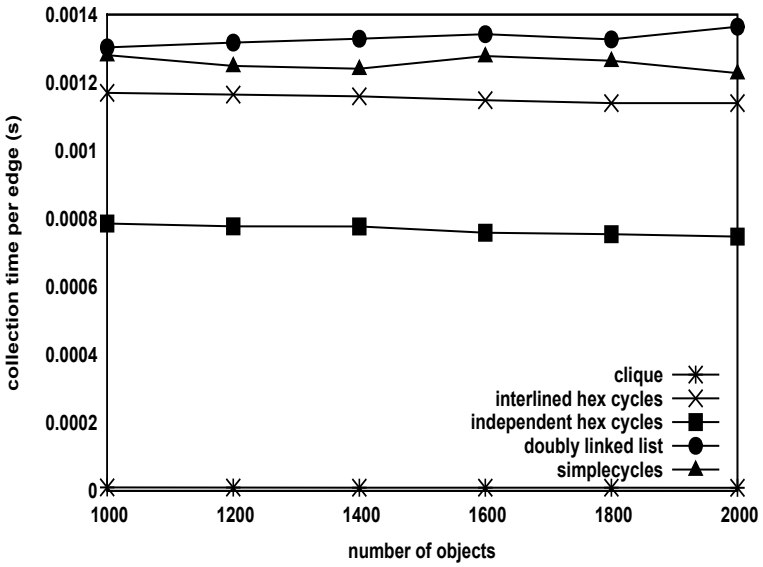

FIGURE 3.9. Graphs of different types are created at various sizes in memory, including cliques, chains of cycles, large cycles, and large doubly linked lists. Regardless of the type of object, collection time per object remains constant, verifying the linearity of the underlying collection mechanism.

profiling tools that identify garbage collection hot spots within a program, and give the programmer the option to take steps (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++ [32], HPX [31], Swarm/ETI [34], UNITAH [7], The Global Arrays Toolkit [50], 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.

3.7 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. [58]; 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.

3.8 Appendix

3.8.1 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 [10].

Algorithm 1 LinkSet creates a new link, and decides the type of the link to create.

```
 1: procedure LINKSET(LINK,NODE)
 2:     lock (link.Source,node)
 3:     if node == NULL then
 4:         LinkFree(link)
 5:         link.Target = NULL
 6:     end if
 7:     if link.Target == node then
 8:         Return
 9:     end if
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:     else if link.Source == node then
17:         link.Which = 1 - node.Which
18:         node.Count[link.Which]++
19:     else if 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:     end if
26:     if oldLink != NULL then
27:         LinkFree(oldLink)
28:     end if
29:     unlock()
30: end procedure
```

Algorithm 2 LinkFree

```
1:  procedure LINKFREE(LINK)
2:      lock(link.Source,link.Target)
3:      if link.Target == NULL then
4:          Return
5:      end if
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 link.Target.Which == link.Which then
11:             if link.Target.Count[1-link.Target.Which] == 0 And link.Target.PhantomCount == 0 then
12:                 Delete(node)
13:                 link.Target = NULL
14:             else
15:                 if link.Target.Collector == NULL then
16:                     link.Target.Collector = new Collector()
17:                 end if
18:                 AddToCollector(link.Target)
19:             end if
20:         end if
21:     end if
22:     unlock()
23: end procedure
```

Algorithm 3 AddToCollector

```
1:  procedure ADDTOCOLLECTOR(NODE)
2:      while True do
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:         end if
12:         unlock()
13:     end while
14: end procedure
```

Algorithm 4 PhantomizeNode

```
1: procedure PHANTOMIZENODE(NODE,COLLECTOR)
2:     lock(node)
3:     while collector.Forward != NULL do
4:         collector = collector.Forward
5:     end while
6:     node.Collector = collector
7:     node.Count[node.Which]- -
8:     phantomize = False
9:     if node.Count[node.Which] > 0 then
10:        Return
11:    else
12:        if node.Count[1-node.Which] > 0 then
13:            node.Which = 1-node.Which
14:        end if
15:        if Not node.Phantomized then
16:            node.Phantomized = True
17:            node.PhantomizationComplete = False
18:            phantomize = True
19:        end if
20:    end if
21:    links = NULL
22:    if phantomize then
23:        links = copy(node.Links)
24:    end if
25:    unlock()
26:    for each outgoing link in links do
27:        PhantomizeLink(link)
28:    end for
29:    lock(node)
30:    node.PhantomizationComplete = True
31:    unlock()
32: end procedure
```

Algorithm 5 Collector.Main

```
1: procedure COLLECTOR.MAIN()
2:     while True do
3:         WaitFor(Collector.RefCount == 0 Or Work to do)
4:         if Collector.RefCount == 0 And No work to do then
5:             Break
6:         end if
7:         while Collector.MergedList.size() > 0 do
8:             node = Collector.MergedList.pop()
9:             Collector.RecoveryList.append(node)
10:        end while
11:        while Collector.CollectionList.size() > 0 do
12:            node = Collector.CollectionList.pop()
13:            PhantomizeNode(node,Collector)
14:            Collector.RecoveryList.append(node)
15:        end while
16:        while Collector.RecoveryList.size() > 0 do
17:            node = Collector.RecoveryList.pop()
18:            RecoverNode(node)
19:            Collector.CleanList.append(node)
20:        end while
21:        while Collector.RebuildList.size() > 0 do
22:            node = Collector.RebuildList.pop()
23:            RecoverNode(node)
24:        end while
25:        while Collector.CleanList.size() > 0 do
26:            node = Collector.CleanList.pop()
27:            CleanNode(node)
28:        end while
29:    end while
30: end procedure
```

Algorithm 6 PhantomizeLink

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

Algorithm 7 DecPhantom

```
1:  procedure DECPHANTOM(NODE)
2:      lock(node)
3:      node.PhantomCount- -
4:      if node.PhantomCount == 0 then
5:          if node.Count[node.Which]== 0 And node.Count[1-node.Which] == 0 then
6:              Delete(node)
7:          else
8:              node.Collector = NULL
9:          end if
10:     end if
11:     unlock()
12: end procedure
```

Algorithm 8 RecoverNode

```
1:  procedure RECOVERNODE(NODE)
2:      lock(node)
3:      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:      end if
9:      unlock()
10:     for each link in links do
11:         Rebuild(link)
12:     end for
13: end procedure
```

Algorithm 9 Rebuild

```
1:  procedure 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:          else if link.Target.Phantomized then
7:              link.Which = link.Target.Which
8:          else if count(link.Target.Links) == 0 then
9:              link.Which = link.Target.Which
10:         else
11:             link.Which = 1-link.Target.Which
12:         end if
13:         link.Target.Count[link.Which]++
14:         link.Target.PhantomCount- -
15:         if link.Target.PhantomCount == 0 then
16:             link.Target.Collector = NULL
17:         end if
18:         link.Phantomized = False
19:         Add link.Target to Collector.RecoveryList
20:     end if
21:     unlock()
22: end procedure
```

Algorithm 10 CleanNode

```
1: procedure CLEANNODE(NODE)
2:     lock(node)
3:     die = False
4:     if node.Count[node.Which]== 0 And node.Count[1-node.Which]== 0 then
5:         die = True
6:     end if
7:     unlock()
8:     if die then
9:         for each link in node do
10:            LinkFree(link)
11:        end for
12:    end if
13:    DecPhantom(node)
14: end procedure
```

Algorithm 11 Delete

```
1: procedure DELETE(NODE)
2:     for each link in node do
3:         LinkFree(link)
4:     end for
5:     freeMem(node)
6: end procedure
```

Algorithm 12 MergeCollectors

```
1:  procedure MERGECOLLECTORS(SOURCE,TARGET)
2:      s = source.Collector
3:      t = target.Collector
4:      done = False
5:      if  s == NULL And t != NULL  then
6:          lock(source)
7:          source.Collector = t
8:          unlock()
9:          Return
10:     end if
11:     if  s != NULL And t == NULL  then
12:         lock(target)
13:         target.Collector = s
14:         unlock()
15:         Return
16:     end if
17:     if  s == NULL Or s == NULL  then
18:         Return
19:     end if
20:     while Not done  do
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:         else if  t.Forward == s and s.Forward == NULL  then
27:             target.Collector = t
28:             source.Collector = t
29:             done = True
30:         else if  t.Forward != NULL  then
31:             t = t.Forward
32:         else if  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:         end if
44:         unlock()
45:     end while
46: end procedure
```

Chapter 4
Distributed Memory Multi-collector Garbage Collection

4.1 Introduction

Garbage-collected languages are widely used in distributed systems, including big-data applications in the cloud [41, 22]. Languages in this space include Java, Scala, Python, C#, PHP, etc., and platforms include Hadoop, Spark, Zookeeper, DryadLINQ, etc. In addition, cloud services such as Microsoft Azure and Google AppEngine, and companies such as Twitter and Facebook all make significant use of garbage-collected languages in their code infrastructure [41]. Garbage collection is seen as a significant benefit by the developers of these applications and platforms because it eliminates a large class of programming errors, which translates into higher developer productivity.

Garbage collection is also an issue in networked object stores, which share many properties with distributed systems. Like such systems, they cannot use algorithms that require scanning the entire heap. In any situation in which traces can go across storage boundaries, i.e. from node to node, node to disk, etc., garbage collectors that need to trace the heap becomes impractical. What is needed is something distributed, decentralized, scalable, that can run without "stopping the world," and has good time and space complexity.

There are two main types of garbage collectors, namely Tracing and Reference Counting (RC). Tracing collectors track all the reachable objects from the root (in the reference graph) and delete all the unreachable objects. RC collectors count the number of objects pointing to a given block of data at any point in time. When a reference count is decremented and becomes zero, its object is garbage, otherwise it *might* be garbage and some tracing is necessary (note that pure RC collectors do not trace and will not collect cycles [47]). Unfortunately, cycles among distributed objects are frequent [59]. Previous attempts at distributed garbage collection, e.g. [24, 33, 35, 45, 43, 66], suffer from the need for centralization, global barriers, the migration of objects, or have inefficient time and space complexity guarantees.

Contributions: We present a hybrid reference counting algorithm for garbage collection in distributed systems that works on the asynchronous model of communication with reliable FIFO channel. Our algorithm collects both non-cyclic and cyclic garbage in distributed systems of arbitrary topology by advancing on

the three reference count collector which only works for a single machine [11] based on the Brownbridge system [12] [1]. The advantage for such systems is that they can frequently determine that a reference count decrement does not create garbage without the need for tracing.

Our proposed algorithm is scalable because it collects garbage in $O(E)$ time using only $O(\log n)$ bits memory per node, where E is the number of edges in the affected subgraph (of the reference graph) and n is the number of nodes of the reference graph. Moreover, in contrast to previous work, our algorithm does not need centralization, global barriers, or the migration of objects. Apart from the benefits mentioned above, our algorithm handles concurrent mutation (addition and deletion of edges and nodes in the reference graph) and provides liveness and safety guarantees by maintaining a property called *isolation*. Theorems 5 and 6 prove that when a collection process works alone (i.e. in isolation), it is guaranteed to collect garbage and not to prematurely delete. Theorem 7 proves that when multiple collection processes interact, our synchronization mechanisms allow each of them to act as if they were working alone, and Theorems 9 and 10 prove the correctness in this setting. To the best of our knowledge, this is the first algorithm for garbage collection in distributed systems that simultaneously achieves such guarantees.

This algorithm falls into the category of self-stabilization algorithms, because it maintains the invariants (1) all nodes strongly connected, (2) no strong cycles, and (3) no floating garbage.

Related Work: Prior work on distributed garbage collection is vast; we discuss here the papers that are closely related to our work. The marking algorithm proposed by Hudak [24] requires a global barrier. All local garbage collectors coordinate to start the marking phase. Once the marking phase is over in all the sites, then the sweeping phase continues. Along with the marking and sweeping overhead, there are consistency issues in tracing based collectors [57].

Ladin and Liskov [33] compute reachability of objects in a highly available centralized service. This algorithm is logically centralized but physically replicated, hence achieving high availability and fault-tolerance. All objects and tables are backed up in stable storage. Clocks are synchronized and message delivery delay is bounded. These assumptions enable the centralized service to build a consistent view of the distributed system. The centralized service registers all the inter-space references and detects garbage using a standard tracing algorithm. This algorithm has scalability issues due to the centralized service. A

[1] Note, the original algorithm of Brownbridge suffered from premature collection, and subsequent attempts to remedy this problem needed exponential time in certain scenarios [61, 53]. We addressed those limitations in [11].

heuristic-based algorithm by Le Fessant [35] uses the minimal number of inter-node references from a root to identify "garbage suspects" and then verifies the suspects. The algorithm propagates marks from the references to all the reachable objects. A cycle is detected when a remote object receives only its own mark. The algorithm needs tight coupling between local collectors and time complexity for the collection of cycles is not analyzed.

Collecting distributed garbage cycles by backtracking is proposed by Maheshwari and Liskov [45]. This algorithm first hypothesizes that objects are dead and then tries to backtrack and prove whether that is true. The algorithm needs to use an additional datastructure to store all inter-site references. An alternative approach of distributed garbage collection by migrating objects have been proposed by Maheshwari and Liskov [43]. Both of the algorithms use heuristics to detect garbage. The former one uses more memory, the latter one increases the overhead by moving the objects between the sites. Recently, Veiga et al. [66] proposed an algorithm that uses asynchronous local snapshots to identify global garbage. Cycle detection messages (CDM) traverse the reference graph and gain information about the graph. A drawback of this approach is that the algorithm doesn't work with the $\mathcal{CONGEST}$ model. Any change to the snapshots has to be updated by local mutators, forcing current global garbage collection to quit. For a thorough understanding of the literature, we recommend reading [57, 1].

Paper Outline: Section 4.2 describes the garbage collection problem, constraints involved in the problem, and the model for which the algorithm is presented. Section 4.3 explains the single collector version of the algorithm and provides correctness, time, and space guarantees. Section 4.4 extends the results of Section 4.3 to the multiple collector scenario. Finally, Section 4.5 concludes the paper with a short discussion. Some of the proofs and the pseudocode for the algorithms may be found in Appendix 4.6.

4.2 Model and Preliminaries

Distributed System: We consider a distributed system of nodes where each node operates independently and communicates with other nodes through message passing. The underlying topology of the system is assumed to be arbitrary but connected. Each node has a queue to receive messages, and in response to a message a node can only read and write its own state and send additional messages. We further assume that the nodes can communicate over a reliable FIFO channel and that messages are never lost, duplicated, or

delivered out of order. These properties make our system compatible with the $\mathcal{CONGEST}$ asynchronous network model with no failures [52].

Basic Reference Graph: We model the relationship among various objects and pointers in memory through a directed graph $G = (V, E)$, which we call a *reference graph*. The graph G has a special node R, which we call the *root*. Node R represents global and stack pointer variables, and thus does not have any incoming edges. Each node in G is assumed to contain a unique ID. All adjacent nodes to a given node in G are called $neighbors$, and denoted by Γ. The $in\text{-}neighbors$ of a node $x \in G$ can be defined as the set of nodes whose outgoing edges are incident on x, represented by $\Gamma_{in}(x)$. The $out\text{-}neighbors$ of x can be defined as the set of nodes whose incoming edges originate on x, represented by $\Gamma_{out}(x)$. Note that each node $x \in G$ does not know $\Gamma_{in}(x)$ at any point in time.

Distributed Garbage Collection Problem: All nodes in G can be classified as either *live* (i.e., not garbage) or *dead* (i.e., garbage) based on a property called $reachability$. Live and dead nodes can be defined as below:

$Reachable(y, x) = x \in \Gamma_{\text{out}}(y) \vee (x \in \Gamma_{\text{out}}(z) \mid Reachable(y, z))$

$Live(x) = Reachable(R, x)$

$Dead(x) = \neg Live(x)$

We allow the live portion of G, denoted as G', to be mutated while the algorithm is running, and we refer to the source of these mutations as the *Adversary.* The Adversary can create nodes and attach them to G', create new edges between existing nodes of G', or delete edges from G'. Moreover, the Adversary can perform multiple events (creation and deletion of edges) simultaneously. The Adversary, however, can never mutate the dead portion of the graph $G'' = G \backslash G'$.

Axiom 3 (Immutable Dead Node)**.** The Adversary cannot mutate a dead node.

Axiom 4 (Node Creation)**.** All nodes are live when they are created by the Adversary.

From Axiom 3, it follows that a node that becomes dead will never become live again.

Each node experiencing deletion of an incoming edge has to determine whether it is still live. If a node detects that it is dead then it must delete itself from the graph G.

Definition 4.2.1 (Distributed Garbage Collection Problem)**.** Identify the dead nodes in the reference graph G and delete them.

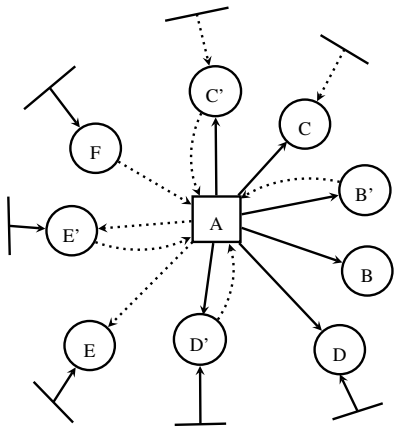

FIGURE 4.1. We depict all the ways in which an initiator node, A, can be connected to the graph. Circles represent sets of nodes. Dotted lines represent one or more non-strong paths. Solid lines represent one or more strong paths. A T-shaped end-point indicates the root, R. If C′, D′, E′ and F are empty sets, A is garbage, otherwise it is not.

Classification of Edges in G: We classify the edges in G as $weak$ or $strong$ according to the Brownbridge method [12]. This classification does not need preprocessing of G and can be done directly at the time of creation of G. Chains of strong edges connect the root R to all the nodes G and contain no cycles. The weak edges are required for all cycle-creating edges in G. The advantage to the Brownbridge method is that it permits less frequent tracing of the outgoing edges. Without a distinction between strong and weak edges, tracing is required after every deletion that does not bring the reference count to zero. In the Brownbridge method, if a strong edge remains after the deletion of any edge, the node is live and no tracing is necessary. Fig. 4.1 illustrates an example reference graph G with strong (solid) and weak (dotted) edges.

Weight-based Edge Classification for the Reference Graph G: We assign *weights* to the nodes as a means of classifying the edges in G. Each node of G maintains two attributes $weight$ and $max\text{-}weight$, as well as counts of strong and weak incoming edges. The weight of the root, R, is zero at all times. For an edge to be strong, the weight of the source must be less than the weight of the target. The $max\text{-}weight$ of a node is defined as the maximum weight of the node's $in\text{-}neighbors$. Messages sent by nodes contain the $weight$ of the sender so that the $out\text{-}neighbors$ can keep track of their strong count, weak count, and $max\text{-}weight$. When a node is created, its initial weight is given by adding one to the weight of the first node to reference it.

Lemma 1 (Strong Cycle Invariant)**.** No cycle of strong edges will be formed by weight-based edge classification.

Proof. By construction, for any node y which is reachable from a node x by strong edges, y must have a larger weight than x. Therefore, if x is reachable from itself through a cycle, and if that cycle is strong, x must have a greater weight than itself, which is impossible. □

Terminology: A $path$ is an ordered set of edges such that the destination of each edge is the origin of the next. A path is called a $strong$ path if it consists exclusively of strong edges, otherwise it is called $non\text{-}strong$ path. We say two nodes are $related$ if a path exists between them. A node, x, is a member of the $dependent\ set$ of A if all strong paths leading to x originate on A (i.e. B, B', C, and C' in Fig. 4.1).

Consider a node, A, that has just lost its last strong edge. We model this situation with an induced subgraph with A removed (i.e. $G - A$), then the $purely\ dependent\ set$ of A consists of the nodes that have no path from R (i.e. B, and B' in Fig. 4.1). A node, y, is a member of the $supporting\ set$ of A if there is a path from R to y, and from y to A (i.e. C', D', E', and F in Fig. 4.1). A node, z, is a member of the $independent\ set$ of A if z is related to A and there is at least one strong path from R to z (i.e. D, D', E, E', and F in Fig. 4.1). A node, x, is a member of the $build\ set$ of A if it is both a member of the supporting set and the independent set. Alternatively, A node, x is said to be in the $build\ set$ if a strong path from R to x exists, but also a path from x to A exists (i.e. D', E' and F in Fig. 4.1). A node, x, is a member of the $recovery\ set$ of A if it is both a member of the supporting set and the dependent set (i.e. C' in Fig. 4.1). A node, x, is said to be in the $affected\ set$ if there exists a path, p, from A to x, such that no node found along p (other than x itself) has a strong path from R (i.e. everything except F in Fig. 4.1).

4.3 Single Collector Algorithm (SCA)

We discuss here the single collection version of our algorithm; the multi-collection version will be discussed in Section 4.4. For the single collector version, we assume that every mutation in G is serialized and that the Adversary does not create or delete edges in G during a *collection process* (i.e. the sequence of messages and state changes needed to determine the liveness of a node). When the last strong edge to a node, x, in G is deleted, but $\Gamma_{\text{in}}(x)$ is not empty, x becomes an *initiator*. The initiator node starts a set of graph traversals which we call phases: *phantomization*, *recovery*, *building*, and *deletion*. We classify the latter three phases as *correction* phases. Fig. 4.2 provides a flow diagram of the phases of an initiator.

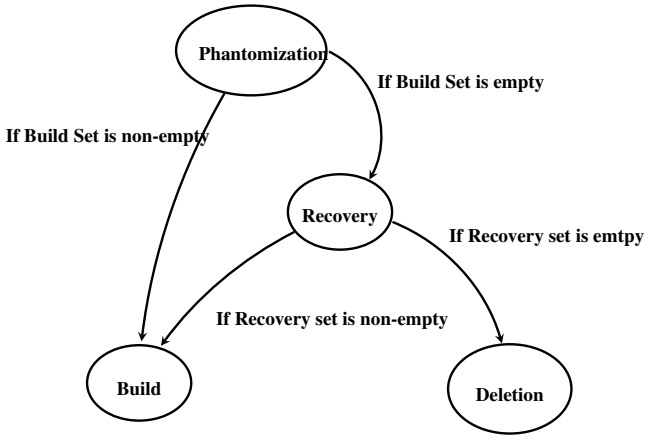

FIGURE 4.2. The above figure depicts the phase transitions performed by initiator in the algorithm.

As illustrated in Fig. 4.1, a node $A \in G$ is dead if and only if its supporting set is empty. If A is discovered to be dead, then its purely dependent set is also dead. Even when the supporting set does not intersect the affected set (i.e. when C', D' and E' are empty), and thus nodes in the build set do not receive any messages, our algorithm will still detect whether the supporting set is empty. Appendix 4.6.2 contains the detailed pseudocode of the single collector algorithm. We describe the phantomization and correction (recovery, building, and deletion) phases separately below.

4.3.1 Phantomization Phase

For a node of G to become a phantom node, the node must (1) have just lost its last strong edge, and (2) not already be a phantom node. Each node has a flag that identifies it as a phantom node. The initiator node is, therefore, always a phantom node, and phantomization always begins with the initiator. When a node w becomes a phantom node, it notifies $\Gamma_{\text{out}}(w)$ of this fact with *phantomize* messages. From that point on, all the outgoing edges from w will be considered to be *phantom edges*, i.e. neither strong nor weak but a transient and indeterminate state. If a node, u, in $\Gamma_{\text{out}}(w)$ loses its last strong edge as a result of receiving a *phantomize* message, u also becomes phantom node, but not an initiator. Phantomization will thus mark all nodes in the dependent set. All the nodes that receive the phantomize message are called *affected nodes*.

Since the algorithm proceeds in phases, we need to wait for phantomization to complete before recovery or building can begin. For this reason, for every *phantomize* message sent from a node, x, to a node, y, a *return* message must be received by x from y . If y does not phantomize as a result of receiving the message from x, the *return* is sent back to x immediately. If the *phantomize* message causes y to phantomize, then y waits until it has received *return* back from all nodes in $\Gamma_{\text{out}}(y)$ before replying to x with *return*. For this purpose, each phantom node maintains a single backward-pointing edge and a counter to track the number of

return messages it is waiting for. While y is waiting for *return* messages, it is said to be *phantomizing*. Once all *return* messages are received, the phantom node is said to be *phantomized*.

Phantomization is, therefore, essentially, a breadth-first search rooted at the initiator. The traversal contains two steps. In the forward step, messages originate from the initiator and propagate to the affected nodes. After they reach all the affected nodes, *return* messages will propagate backward to the initiator. The reverse step is a simple termination detection process which uses a spanning tree in the subgraph comprised of the affected nodes (i.e., the single backward-pointing edge forms the spanning tree).

The edges of any phantom node are said to be *phantom edges*. Each node has a counter for its incoming phantom edges (in addition to its strong and weak counter). Every node in the affected subgraph will have a positive phantom count at the end of the phantomization phase. The initiator then enters to the correction phase.

In addition to sending phantomize messages to its $out\text{-}neighbors$, when a node satisfies the above conditions and contains only weak incoming edges, the node converts all the weak incoming edges into strong incoming edges. The process of converting all the weak incoming edges into strong edges is called *toggling*. Toggling is achieved by updating the weight of the node to the maximum weight of its in-neighbors plus one ($weight \leftarrow max\text{-}weight + 1$).

4.3.2 Correction Phase

The initiator starts either the *recovery* or *building* phase depending on the build set. If the build set is empty, the initiator enters the recovery phase; if it is not empty, it enters the building phase. In the building phase, the affected subgraph is traversed, phantom edges are converted to strong and weak edges, and the weights of the nodes are adjusted. In the recovery phase, the affected subgraph is traced until the recovery set is found (i.e. phantom nodes that have strong incoming edges). If and when this set is found, building phase begins. If not the initiator deletes itself and all the purely dependent nodes. We describe each phase in detail below.

Building Phase: If the initiator has any strong incoming edges after the phantomization phase, then the build set is not empty. In response, the initiator updates its phase to *building* and sends *build* messages to its $out\text{-}neighbors$, which convert phantom edges to strong or weak edges.

If a node, y, sends a *build* message to a node, x, that is phantom node, the $max\text{-}weight$ of x is set to the weight of y ($x.max\text{-}weight \leftarrow y.weight$), and the weight of x is set to the weight of y plus one ($x.weight \leftarrow y.weight + 1$). The node x then decrements the phantom count, increments the strong count, and propagates the *build* message to its $out\text{-}neighbors$.

If a node, y, sends a *build* message to a node, x, that is not phantom node, it updates the $max\text{-}weight$ of x if necessary, decrements the phantom count and increments either the strong or the weak count, depending on whether $y.weight < x.weight$.

After a node, x, builds its last phantom edge and replies to its parent in the spanning tree with *return* (if it is the initiator node, it does not do this), x is then returned to a *healthy* state (i.e. all flags set to false, etc.).

Lemma 2 (Not Phantomized)**.** After phantomization, nodes in the build set are not phantom nodes.

Proof. Appendix 4.6.4 contains the proof. □

Lemma 3 (Build set)**.** After phantomization, if the initiator has strong incoming edges, then the initiator is not dead.

Proof. Appendix 4.6.4 contains the proof. □

Like the phantomization phase, the building phase has a forward and reverse step and creates a spanning tree by resetting the parent attribute of the node in the same way as phantomization (i.e. when it receives the first *build* message). When all its $out\text{-}neighbors$ reply with *return*, it sends *return* to its parent. A node that receives a *build* message, but is already building, builds the phantom edge and sends the *return* message to the sender.

Recovery Phase: Recovery is the search for the recovery set. If found, a build of the subgraph of nodes that are reachable from recovery set of nodes will begin. It is possible that the recovery phase can build edges to the initiator and thereby rebuild the entire affected set.

Lemma 4 (Phantomized)**.** After phantomization, nodes in the recovery set are phantom nodes.

Proof. Appendix 4.6.4 contains the proof. □

Lemma 5 (Recovery set)**.** After phantomization, if a phantom node contains at least one strong incoming edge, it belongs to the recovery set.

Proof. Appendix 4.6.4 contains the proof. □

After the phantomization phase is complete, all phantom nodes that contain strong incoming edges are members of the recovery set. Once a set of phantomized nodes with strong edges are detected, the building process described above can begin from those nodes.

When a phantom node with zero strong count receives a *recovery* message it simply updates its state to *recovering* and propagates the *recovery* message to $out\text{-}neighbors$. As in phantomization, when it receives all *return* messages back, it changes its state to *recovered*. Also, as in phantomization and building, a spanning tree and *return* messages will be used to track the progress of recovery. If a node receives a *recovery* message and that node has incoming strong edges, it immediately starts building instead of recovering.

Note that it is possible for a build message to reach a node, x, that is still in the forward step of the recovery process. To accommodate this, when x receives all its *return* messages back from $\Gamma_{\text{out}}(x)$, x checks to see if it now belongs to the recovery set (i.e. if it has positive strong count). If it still does not belong to the recovery set, then the return message is sent back to the parent as usual. If it is part of recovery set, the node starts the build process. Any building node that sends the *return* message to its parent is not part of the phantomized subgraph and becomes healthy, i.e. it is no longer part of any collection process.

Deletion Phase: If the recovery phase finishes and the initiator, x, still has no incoming strong edge, it issues *plague delete* messages to $\Gamma_{\text{out}}(x)$. As the name suggests, this message is contagious and deadly. Any node receiving it decrements its phantom count by one and (if the recipient has no incoming strong edges) it sends the *plague delete* message along its outgoing edges. Once a node has no edges (i.e. phantom, strong, and weak count are zero), it is deleted. Unlike the other phases, there is no *return* message for the *plague delete* message.

Lemma 6 (Time Complexity)**.** SCA finishes in O(Rad(i,G_a)) time, where G_a is the graph induced by affected nodes, i is the initiator, and Rad(i, G_a) is the radius of the graph from from i.

Proof. Appendix 4.6.4 contains the proof. □

Corollary 7. SCA finishes in O(E_a) time, since Rad(i, G_a) can be O(E_a).

Lemma 8 (Communication Complexity)**.** SCA sends O(E_a) messages, where E_a is the number of edges in the graph induced by affected nodes.

Proof. Appendix 4.6.4 contains the proof. □

Lemma 9 (Message Size Complexity)**.** SCA sends messages of O(log(n)) size, where n is the total number of nodes in the graph.

Proof. Appendix 4.6.4 contains the proof. □

Theorem 5. All dead nodes will be deleted at the end of the correction phases.

Proof. Appendix 4.6.4 contains the proof. □

Theorem 6. No live node will be deleted at the end of the correction phases.

Proof. Appendix 4.6.4 contains the proof.

□

4.4 Multi-Collector Algorithm (MCA)

We now discuss the case where mutations in G are not serialized and the Adversary is allowed to create and delete edges in G during the collection process. As a consequence, multiple deletion events might create collection processes with overlapping (induced) subgraphs, possibly trying to perform different phases of the algorithm at the same time. We assign unique identifiers to collection processes to break the symmetry among multiple collectors and a transaction approach to ordering the computations of collectors if collection operations conflict. By doing this, we allow each collection process to operate in *isolation*. Appendix 4.6.3 contains the detailed pseudocode of the multi-collector algorithm.

Definition 4.4.1 (Isolation)**.** A collection process is said to proceed in isolation if either (1) the affected subgraph does not mutate, or (2) mutations to the affected subgraph occur, but they do not affect the decisions of the initiator to delete or build, and do not affect the decisions of the recovery set to build by the correction phases.

Symmetry Breaking for Multiple Collectors: Each collection process has a unique id. If there are two or more (collection) processes acting on the same shared subgraph, the collection process with the higher id (higher priority) will acquire ownership of the nodes (of the shared subgraph). When a node receives a message from a higher priority process, it marks itself with that new higher id.

The phantomization phase of the multi-collector does not mark nodes with the collection process id, however, as the simultaneous efforts of multiple collection processes to phantomize do not interfere in a harmful way. Moreover, when an initiator node, I, receives a *phantomize* message from a higher priority collection process, I will unmark its collection id, and will, therefore, loses its status as an initiator.

Another graph, which we call the *collection process graph* and denote by C, is formed by treating each set of nodes with the same collection id as a single node. Edges in G are included in C if and only if they connect nodes that are part of different collection processes. To avoid ambiguity, we refer to nodes in C as *entities*.

Lemma 10. The collection process graph is eventually acyclic.

Proof. Appendix 4.6.4 contains the proof. □

Recovery Count: Consider the configuration of nodes in Fig. 4.3. Initially, we have strong paths connected to both N_6 and N_1 and after these paths are deleted we have two collection processes. Assume that these two collection processes, with priority I_6 and I_1, complete their phantomizations phases, and process I_6 begins its recovery phase. Process I_6 should identify node N_4 as belonging to the recovery set, but if process I_1 has not completed recovering and building, it will not. Therefore, the recovery phase for I_6 will complete and prematurely delete N_6.

We remedy this kind of conflict using *recovery count*. The recovery count tracks the number of *recovery* messages each node receives from its own process. A recovering node can neither send the *return* message nor start deleting until the recovery count is equal to the phantom count. The fourth edge type in the abstract refers to the recovery count. A recovery edge is also a phantom edge, and a phantom edge will be converted to a recovery edge when a recover message is sent for that edge with the same CID.

In the configuration discussed above, this would cause process I_6 to stall at node N_4 while it waits for the recovery count and phantom count to become equal. When process I_1 rebuilds the edge to N_4 as strong, this condition is met and process I_6 stops waiting and rebuilds.

We assume in the discussion above that I_6 has higher priority than I_1. If the priority were reversed, I_1 would take over building node N_4, but this introduces additional complexity which we will now address.

Lemma 11. If an entity A precedes an entity B topologically in the collection process graph, and A has a lower priority than B, entity A will complete before entity B and both will complete in isolation.

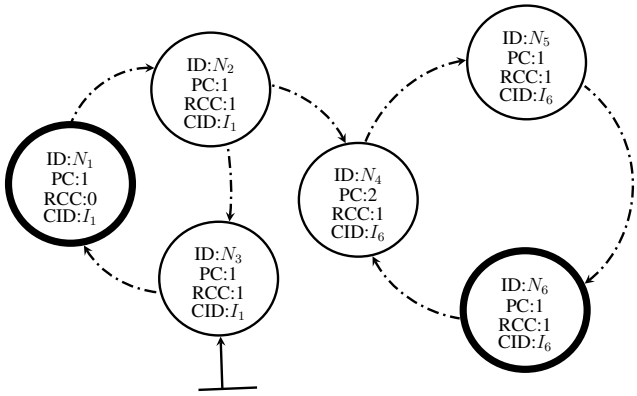

FIGURE 4.3. We depict two collection processes in the recovery phase. Each circle is a node. Node properties are: node id (ID), phantom count (PC), recovery count (RCC), collection id (CID). Bold borders denote initiators, and dot and dashed edges denote phantom edges.

Proof. Appendix 4.6.4 contains the proof. □

Transaction Approach: Consider a *recover* message, m, arriving at node N_4 from N_2 (Fig. 4.3). This time, assume that the collection id of I_1 is higher than I_6. If N_4 is in the recovered state, the collection id on x is updated with the collection id in m ($x.CID \leftarrow m.CID$). If, however, x is recovering, i.e. it is waiting for *return* messages from its $out\text{-}neighbors$, a *return* message, r, is immediately sent to N_4's parent, N_4's parent is set to point to N_2, and the *re-recover flag* on x is set. The re-recover flag will cause recovery to restart on x once the current recovery operation finishes. This will allow the higher priority collection to take over managing the subgraph.

In addition, a flag called *start recovery over (SRO)* is set on the *return* message. The SRO propagates back to N_6, where the recovery count will be reset and the entire recovery phase will start over, but with a slightly higher collection priority, I_6', which is still lower than I_1. The slightly higher priority (signified by the prime) doesn't change the relative priority compared to another independent collection process, it merely ensures that any increments to the recovery count come from the new, restarted recovery operation and not leftover messages from the old one (I_6). After this, the new recovery phase will not proceed until the phantom count is equal to the recovery count ($PC = RCC$), which ensures that the new recovery does not start until the original *recovery* phase is complete. It is easy to see that the new recovery triggered by the SRO flag is similar to the rollback of a transactional system.

When a node changes ownership, the node has to join the phase of the new collection. Each node has three redo flags: rephantomize, rebuild, and re-recover. These flags only take effect after all *return* messages have come back. This way the messages of old collection processes will never be floating in the system.

Lemma 12. If an entity A precedes an entity B in the collection process graph C with respect to a topological ordering, and A has higher priority than B, entity A will take ownership of entity B and A will proceed in isolation.

Proof. Because A has higher priority, it takes ownership of every node it touches and is thus unaware of B. So it proceeds in isolation. If B loses ownership of a recovered node, it will not affect isolation because a recovered node has received *recovery* messages on all its incoming edges ($RCC = PC$), and has already sent its return. If B loses a node that is building or recovering, B is forced to start over. In the new recovery or build phase, B will precede A in the topological order, and both will proceed in isolation. □

Theorem 7. All collection processes will complete in isolation.

Proof. By Lemma 10 we know that the collection process graph will eventually be topologically ordered, and the ordering will not violate isolation given by Lemma 12. Once ordered, the collection processes will then complete in order proven by Lemma 11. □

Corollary 8. In quiescent state (i.e. one in which the Adversary takes no further actions on G_a), with p active collection processes, our algorithm finishes in $O(Rad_{all})$ time where G_a is the affected subgraphs of all connected collection process and Rad_{all} is the sum of the radii of the affected subgraphs of all p initiators.

Handling the Mutations of The Adversary: Creation of edges and nodes by the Adversary poses no difficulty. If an edge is created that originates on a phantomized node, the edge is created as a phantom edge. If a new node, x, is created, and the first edge to the node comes from a node, y, then x is created in the same state as y (e.g., phantomized, recovered, etc.).

Lemma 13 (Creation Isolation)**.** Creation of edges and nodes by the Adversary does not violate isolation.

Proof. The addition of edges cannot affect the isolation of a graph because (1) addition of an edge cannot cause anything to become dead and (2) if an edge is created to or from any edge in a collection process, then the process was already live by Axiom 3. □

Deletion is comparatively difficult, though. If the Adversary deletes a phantom edge $x \to y$ from collection x_1, and $\Gamma_{\text{in}}(y)$ is not empty, it is not enough to reduce the phantom count on y by 1. In this case, y is made the initiator for a new collection process, y_2, such that y_2 has higher priority than x_1. If y is in the recovering state, it sends *return* with $SRO = true$, and re-recovers y. If x is not phantomized, the procedure is similar to the single collector algorithm.

Lemma 14 (Deletion Isolation)**.** Edge deletion by the Adversary does not violate isolation.

Proof. Appendix 4.6.4 contains the proof. □

Theorem 9 (Liveness)**.** Eventually, all dead nodes will be deleted.

Proof. By Axiom 3, we know the Adversary cannot create any edges to dead nodes. Theorem 5 proves that if a collection process works in isolation, all dead nodes will be deleted. Lemma 14 proves that the deletion of an edge in the affected subgraph does not affect isolation property. Lemma 7 proves that indeed all collection processes complete in isolation. Theorem 5 proves all collection processes in isolation delete dead nodes. Thus, eventually, all dead nodes will be deleted. □

Theorem 10 (Safety)**.** No live nodes will be deleted.

Proof. Theorem 6 proves that if a collection process works in isolation, no live nodes of G will be deleted. When the Adversary adds a strong edge in the affected subgraph, Lemma 13 proves that it does not violate the isolation property. Theorems 6 and 7 prove that no live nodes will be deleted. □

4.5 Conclusions

We have described a hybrid reference counting algorithm for garbage collection in distributed systems. The algorithm improves on the previous efforts by eliminating the need for centralization, global barriers, back references for every edge, and object migration. Moreover, it achieves efficient time and space complexity. Furthermore, the algorithm is stable against concurrent mutations in the reference graph. The main idea was to develop a technique to identify the supporting set (nodes that prevent a given node from being dead) and handle the synchronization of multiple collection processes. We believe that our techniques might be of independent interest in solving other fundamental problems that arise in distributed computing.

In future work, we hope to address how to provide an order in which the dead nodes will be cleaned up, permitting some kind of "destructor" to be run, and to address fault tolerance. In addition, we hope to

implement the proposed algorithm and compare its performance with previous algorithms using different benchmarks.

4.6 Appendices

4.6.1 Terminology

SC Strong Reference Coung

WC Weak Reference Count

PC Phantom Reference Count

RCC Recovery Count

CID Collector Identifier

SRO Start Recovery Over.

Parent Parent Pointer.

<**MSG TYPE, Receiver, CID, SRO**> Sends a message of mentioned type to receiver. Message types will one of the following: Phantomize(PH), Recovery(RC), Build(B), Return(R), Delete(D), Plague Delete(PD). All parameters are not necessary for some messages.

State State can be one of the following: Phantomizing, Phantomized, Recovering, Recovered, Building, Healthy.

Healthy SC $>$ 0 & PC $=$ 0 & RCC $=$ 0 & PhantomNodeFlag $=$ false & CID $=\perp$ & All return msgs received

Weakly Supported WC $>$ 0 & SC $=$ 0 & PC $=$ 0 & RCC $=$ 0 & PhantomNodeFlag $=$ false & CID $=\perp$ & waiting $=$ 0

Maybe Delete PC $=$ RCC & SC $=$ WC $=$ 0 & waiting $=$ 0 & CID $\neq\perp$ & PC $>$ 0

Simply Dead PC $=$ RCC & SC $=$ WC $=$ 0 & waiting $=$ 0 & PC $=$ 0

Garbage Simply Dead or (Maybe Delete & Initiator)

4.6.2 Single Collector Algorithm

Algorithm 13 Single Collector Algorithm

```
1:  procedure ONMSGRECEIVE
2:      if msg type = PH then
3:          Decrement SC/WC and Increment PC
4:          if state = Phantomizing then
5:              <R, msg.sender >
6:          else
7:              Parent = msg.sender
8:              if SC = 0 then
9:                  Toggle SC and WC
10:                 <PH, Γ_out >
11:                 state = Phantomizing & PhantomNodeFlag = true
12:             end if
13:             if No Out-neighbors or SC > 0 then
14:                 <R, Parent>
15:             end if
16:         end if
17:     else if msg type = RC then
18:         if state = Phantomized then
19:             Parent = msg.sender.
20:             if  SC > 0 then
21:                 state = Building
22:                 < B, Γ_out >
23:             else
24:                 state = Recovering
25:                 <RC, Γ_out >
26:             end if
27:             if No Out-neighbors then
28:                 <R, msg.sender>
29:             end if
30:         else
31:             < R, msg.sender >
32:         end if
33:     else if msg type = B then
34:         Increment SC/WC and decrement PC.
35:         if state = Phantomized or Recovered then
36:             state = Building & Parent = msg.sender
37:             <B, Γ_out >
38:         else
39:             <R, msg.sender>
40:         end if
```

```
41:  else if msg type = PD then
42:      Decrement PC.
43:      if SC= 0 & WC = 0 & PC = 0 then
44:          <PD , Γ_out > Γ_out = ⊥
45:      end if
46:      if SC= 0 & WC = 0 & PC = 0 then
47:          Delete the Node
48:      end if
49:  else if msg type = D then
50:      Decrement SC/WC.
51:      if  SC = 0 & WC > 0 then
52:          Toggle WC and SC.
53:          state = Phantomizing.
54:          <PH , Γ_out >
55:      else if  SC=0 & WC = 0 then
56:          <D , Γ_out >
57:          Delete the node.
58:      end if
59:  else if msg type = R & All return msgs received then
60:      if state = Phantomizing then
61:          state = Phantomized
62:          if Not Initiator then
63:              <R , Parent >
64:          else
65:              if SC > 0  then
66:                  state = Building
67:                  <B , Γ_out >
68:              else
69:                  state = Recovering
70:                  <RC , Γ_out >
71:              end if
72:          end if
73:      else if state = Recovering then
74:          state = Recovered
75:          if SC > 0 then
76:              state = Building
77:              <B , Γ_out >
78:          else
79:              if Initiator then
80:                  <PD , Γ_out >
81:              else
82:                  <R , Parent >
83:              end if
84:          end if
```

```
85:          else if state = Building then
86:              state = Healthy
87:              if  Not Initiator then
88:                  <R , Parent >
89:              end if
90:          end if
91:      end if
92:  end procedure
```

4.6.3 Multi-collector Algorithm

Algorithm 14 Edge Deletion

```
1:  procedure ONEDGEDELETION
2:      Decrement SC, WC, or PC
3:      if msg.CID = CID then
4:          Decrement RCC
5:      end if
6:      if Garbage then
7:          if state = Recovered then
8:              <PD , Γ_out >
9:          else if Simply Dead then
10:             <D , Γ_out >
11:             Delete the node.
12:         end if
13:     else if Weakly Supported then
14:         Toggle SC and WC
15:         if Γ_out = ⊥ then
16:             return
17:         end if
18:         CID = new Collector Id
19:         <PH , Γ_out >
20:         PhantomNodeFlag = true
21:     else if PC > 0 & SC = 0 then
22:         Toggle SC and WC.
23:         if Not Initiator then
24:             <R , Parent, CID, True>
25:         end if
26:         if Waiting for return msgs then
27:             Rerecover = true
28:             Rephantomize = true
29:         end if
30:         CID = new Higher Collector Id
31:     end if
32: end procedure
```

Algorithm 15 On receiving Phantomize messge

```
1: procedure ONPHANTOMIZERECEIVE
2:     Decrement SC/WC and Increment PC.
3:     if state = Phantomizing then
4:         if Initiator then
5:             if CID ≥ msg.CID then
6:                 <R , msg.sender, CID >
7:             else
8:                 Parent = msg.sender
9:                 CID = ⊥
10:            end if
11:        else
12:            <R , msg.sender, CID >
13:        end if
14:    else if state = Phantomized or Healthy then
15:        Parent = msg.sender
16:        < R , msg.sender, CID >
17:    else if SC = 0 then
18:        Toggle SC and WC
19:        Parent = msg.sender
20:        state = Phantomizing and PhantomNodeFlag = true
21:        < PH , Γ_out, CID>
22:        if Γ_out is empty then
23:            state = Phantomized
24:            <R , Parent>
25:        end if
26:    else if state = Building then
27:        if SC > 0 then
28:            <R , Msg.sender>
29:        else
30:            Rephantomize = true
31:            if Not an initiator then
32:                <R , Parent >
33:                Parent = msg.sender
34:            else
35:                if msg.CID > CID then
36:                    CID = ⊥ and Parent = msg.sender
37:                else
38:                    <R , Msg.sender, CID >
39:                    Rerecover = true
40:                end if
41:            end if
42:        end if
43:    end if
44: end procedure
```

Algorithm 16 On receiving Return message

```
1:  procedure ONRETURNRECEIVE
2:      if msg.SRO = true then
3:          SRO = true
4:      end if
5:      if All return msgs received & Phantomizing then
6:          if Initiator or Rerecover = true then
7:              Rerecover = false
8:              if SC>0 then
9:                  state = Building
10:                 < B , Γ_out, CID >
11:             else
12:                 state = Recovering
13:                 < RC , Γ_out, CID >
14:             end if
15:         else
16:             <R , Parent >
17:         end if
18:     end if
19:     if All return msgs recieved & Building then
20:         if SC > 0 then
21:             if not an Initiator then
22:                 < R , Parent >
23:             end if
24:             state = Healthy.
25:         else
26:             Rephantomize = false
27:             Rerecover = true
28:             state = phantomizing
29:             <PH , Γ_out, CID >
30:         end if
31:     end if
32:     if All return msgs received & Recovering then
33:         if RCC = PC then
34:             if Initiator then
35:                 if SRO = true then
36:                     Rerecover = true
37:                     CID = new Slightly Higher Collector Id
38:                     SRO = false
39:                 end if
40:                 if Rerecover = true or SC >0 then
41:                     Rerecover = false
42:                     RCC = 0
```

```
43:                     if SC = 0 then
44:                         state = Recovering
45:                         <RC , Γ_out,CID >
46:                     else
47:                         state = Building
48:                         < B , Γ_out, CID >
49:                     end if
50:                 else if Garbage then
51:                     < PD , Γ_out, CID >
52:                 end if
53:             else
54:                 if SRO = true then
55:                     State = Recovered
56:                     SRO = false
57:                     <R , Parent, CID, True >
58:                 else if Rerecover = true or SC >0 then
59:                     Rerecover = false
60:                     if SC = 0 then
61:                         state = Recovering
62:                         < RC , Γ_out, CID>
63:                     else if SC > 0 then
64:                         state = Building
65:                         < B , Γ_out, CID >
66:                     end if
67:                 else if Maybe Delete then
68:                     state = Recovered
69:                     <R , Parent>
70:                 end if
71:             end if
72:         else if RCC ≠ PC then
73:             if Rerecover = true then
74:                 Rerecover = false
75:                 if Initiator then
76:                     RCC = 0
77:                 end if
78:                 if SC = 0 then
79:                     state = Recover
80:                     < RC , Γ_out, CID >
81:                 else if SC > 0 then
82:                     state = Building
83:                     <B , Γ_out, CID >
84:                 end if
85:             end if
86:         end if
87:     end if
88: end procedure
```

Algorithm 17 On receiving Build message

```
1: procedure ONBUILDRECEIVE
2:     Increment SC/WC, and Decrement PC.
3:     if msg.CID = CID then
4:         Decrement RCC
5:     end if
6:     if state = Building or SC > 0 then
7:         < R , Msg.Sender, CID >
8:     else if state = Phantomizing then
9:         if CID ≥ msg.CID or CID = ⊥ then
10:            <R , msg.sender >
11:        else
12:            if Not Initiator then
13:                <R , Parent >
14:            end if
15:            Parent = msg.sender
16:            CID = msg.CID and Rerecover = True
17:        end if
18:    else if state = Phantomized then
19:        Parent = msg.sender
20:        CID = msg.CID
21:        if PhantomNodeFlag = True then
22:            state = Building
23:            < B , Γ_out, CID >
24:        else
25:            state = Healthy
26:            < R , Parent >
27:        end if
```

```
28:     else if state = Recovering & Waiting for return msgs then
29:         if CID≥msg.CID then
30:             <R , msg.sender, CID >
31:         else
32:             if Not Initiator then
33:                 < R , Parent, CID, True >
34:             end if
35:             CID = msg.CID
36:             Parent = msg.Sender
37:         end if
38:     else if state = Recovering &
39:          All return msgs received & RCC < PC then
40:         if CID≥msg.CID then
41:             <R , msg.sender, CID >
42:         else
43:             if Not Initiator then
44:                 <R , Parent , True >
45:             end if
46:             Parent = msg.sender
47:             CID = msg.CID
48:         end if
49:         Start Building
50:     else if state = Recovered then
51:         Update RCC if required
52:         CID = msg.CID
53:         Parent = msg.sender
54:         state = Building
55:         < B , Γ_out, CID >
56:     end if
57: end procedure
```

Algorithm 18 On receiving Recovery message

```
1: procedure ONRECOVERYRECEIVE
2:     if S > 0 then
3:         <R , msg.sender >
4:     else if state = Phantomizing then
5:         if Not Initiator then
6:             <R , Parent, CID, True >
7:         end if
8:         Increment RCC
9:         Rerecover = True
10:        CID = msg.CID
11:        Parent = msg.sender
12:    else if state = Phantomized then
13:        Increment RCC
14:        state = Recovering
15:        CID = msg.CID
16:        if SC > 0 then
17:            <B , Γ_out, CID >
18:        else
19:            <RC , Γ_out, CID >
20:        end if
21:        Parent = msg.sender
22:    else if state = Building then
23:        if msg.CID < CID then
24:            Rephantomize = true
25:            Rerecover = true
26:            <R , msg.sender >
27:        else if msg.CID = CID then
28:            <R , msg.sender >
29:        else
30:            if Not Initiator then
31:                <R , Parent, CID, True >
32:            end if
33:            CID = msg.CID
34:            Rerecover = true
35:            Parent = msg.sender
36:            Rephantomize = true
37:        end if
```

```
38:     else if state = Recovering then
39:         if msg.CID < CID then
40:             <R , msg.sender >
41:         else if msg.CID = CID then
42:             Increment RCC
43:             <R , msg.sender >
44:             if Not Initiator & RCC = PC &
45:                         All return msgs received then
46:                 <R , Parent >
47:             end if
48:         else
49:             if Not Initiator then
50:                 <R , Parent, CID, True >
51:             end if
52:             CID = msg.CID
53:             RCC = 1
54:             Parent = msg.sender
55:             if Waiting for return msgs then
56:                 Rerecover = True
57:             else
58:                 <RC , Γ_out, CID >
59:             end if
60:         end if
61:     else if state = Recovered then
62:         RCC = 1;
63:         CID = msg.CID
64:         state = Recovering
65:         Parent = msg.sender
66:         <RC, Γ_out, CID, >
67:     end if
68: end procedure
```

4.6.4 Appendix of proofs

Lemma 2. After phantomization, nodes in the build set are not phantom nodes.

Proof. By definition, a strong path from R to each node in the build set exists, therefore phantomization spreading from the initiator cannot take away the last strong edge, and so the build set will not phantomize. □

Lemma 3. After phantomization, if the initiator has strong incoming edges, then the initiator is not dead.

Proof. Only nodes in the supporting set can keep the initiator, I, alive. When the initiator became a phantom node, it converted its weak incoming edges to strong. However, since nodes in the recovery set all became phantom nodes by Lemma 4, they will not contribute strong support to the initiator. Since nodes in the build set do not phantomize by Lemma 2, the edges connecting them to the initiator remain strong. □

Lemma 4. After phantomization, nodes in the recovery set are phantom nodes.

Proof. By definition, no strong path from R to the nodes in the recovery set exists. The strong edges they do have come from the initiator, and these will phantomize after the initiator phantomizes. Once that happens, the recovery set will phantomize. □

Lemma 5. After phantomization, if a phantom node contains at least one strong incoming edge, it belongs to the recovery set.

Proof. Every node in the recovery set will phantomize by Lemma 4, and every node in the recovery set will have a non-strong path from R prior to phantomization. Phantomization, however, will cause the nodes to toggle, converting the non-strong path from R to a strong path for at least one node in the recovery set. Therefore, the recovery set will contain at least one node with at least one strong incoming edge.

□

Lemma 6. SCA finishes in O(Rad(i,G_a)) time, where G_a is the graph induced by affected nodes, i is the initiator, and Rad(i, G_a) is radius of the graph from i.

Proof. In each time step, *phantomization* spreads to the $out\text{-}neighbors$ of the previously affected nodes, increasing the radius of the graph of phantom nodes by 1. In O(Rad(i,G_a)) time, all the nodes in G_a receive a *phantomize* message, since all the nodes in G_a are at distance less than or equal to Rad(i,G_a) from i. In the reverse step, the same argument can be applied backward. So phantomization completes in O(Rad(i,G_a) time.

In the correction phase, during the forward step, in r time, r neighborhoods of i received *recovery* or *build* or *plague delete* message, until the affected subgraph is traversed. In the reverse step of *build* or *recovery*, however, a *return* message might initiate the build process. While the build process nodes send return messages, the nodes will become healthy thereby reducing the Rad(i,G_a). So in the worst case, all nodes that received recovery might build. Because each node will have only one parent, the return step cannot take more than O(Rad(i, G_a) time. □

Lemma 8. SCA sends O(E_a) messages, where E_a is the number of edges in the graph induced by affected nodes.

Proof. In the forward step of the phantomization, all the nodes in the dependent set send the *phantomize* message to their $out\text{-}neighbors$, and each node can do this at most once (after which the phantom node flag is set). So the forward step of the algorithm uses only O(E_a) messages. In the reverse step, the *return* messages are sent backward along the edges of the spanning tree. So the reverse step sends O(V_a) messages, where V_a is the number of nodes in the affected subgraph. So Phantomization uses O(E_a) messages.

In the forward step of the recovery/building, either a *recovery* message, a *build* message, or both traverses every edge, so the forward step of the algorithm uses O(E_a) messages. In the reverse step of the algorithm, the *return* messages are sent back to the parent a maximum of two times (once for recovery, once for build), traversing a subset of the edges in the reverse direction. Thus, there is a maximum of four message traversals for any edge. In every forward step of the plague delete, all outgoing edges are consumed, and therefore it takes O(E_a) messages. □

Lemma 9. SCA sends messages of O(log(n)) size, where n is the total number of nodes in the graph.

Proof. The messages have to hold a value at least as large as the count of nodes in the system which are O(log(n)) size. Apart from the ids, the message also contains the weight of node which is constant in size. In the return message, our algorithm only uses the id of the sender and receiver. So all the messages in the SCA are of O(log(n)) size. □

Theorem 5. All dead nodes will be deleted at the end of the correction phases.

Proof. Assume a node y, is a dead node, but flagged as live node by the correction phase. If y becomes live, then it must have done so because its edges were rebuilt during building. If so, then either y has a strong count, or there exists a node x with a strong count which also has a phantom path to y. However, any node with a strong count at the end of phantomization is a live node by Lemma 3 and Lemma 5, and because a path from x to y exists, x is also live. This contradicts our assumption. □

Theorem 6. No live node will be deleted at the end of the correction phases.

Proof. Assume a node, x is live, but it is deleted. If x is the initiator, and is live then the supporting set is not empty. If the build set isn't empty, then x will have a strong count, so it won't be deleted. If the build set is

empty and the recovery set is non-empty, then correction will build a strong edge to x, so it won't be deleted. This contradicts our assumption. Now assume x is dead, but it causes some other node to be deleted. If x is dead, then the supporting set is empty and the purely dependent set is dead. The independent set has a strong count before the collection process, so it won't delete. The nodes in the dependent set that had a non-strong path from R before phantomization, will have a strong path from R after toggling and recovery/build, so they will not delete. This also contradicts our assumption. □

Lemma 10. The collection process graph is eventually acyclic.

Proof. If a cycle exists between two process graphs, A and B, then recovery or build messages must eventually propagate from one to the other. Without loss of generality, assume A has higher priority than B. During the correction phases, messages propagate along all phantom edges, and in the process take ownership of any nodes they touch. Eventually, therefore, there should be no edges from A to B. □

Lemma 11. If an entity A precedes an entity B topologically in the collection process graph, and A has a lower priority than B, entity A will complete before entity B and both will complete in isolation.

Proof. Consider a node, x, that has incoming edges from both A and B. Process B will have ownership of the node during the recovery or build phase, but until the edges from A are either built or deleted, x will have an RCC equal to the number of edges from B, and PC equal to the sum of the edges from $A + B$. So the recovery or build phase of B must take place after A, and so B will operate in isolation. Since there are no edges from B to A, and since B is not making progress, B does not affect A, and A is in isolation. □

Lemma 14. Edge deletion by the Adversary does not violate isolation.

Proof. If we simply remove an edge $x \rightarrow y$ from inside a collection process graph, that might violate isolation because it removes y from the graph. However, we have already developed a method to remove a node from the graph without violating isolation, and that is to give the node to a higher collection process id. So when the edge is deleted, a new, higher collection process is created and it is given ownership of y. By Theorem 7, the old and new collection process will proceed in isolation. □

Chapter 5
Conclusion

In the multi-core and distributed systems era, a complete high performance garbage collector is necessary. The garbage collection algorithms discussed in this thesis are designed for models that mimic the reality in shared memory and distributed memory systems. These garbage collectors can be implemented without any changes for any programming language. The thesis contributions are summarized as below:

- Presented a novel, cyclic, hybrid reference counting and tracing garbage collector for general graphs.
- Designed the first known concurrent, multi-collector, shared memory garbage collector.
- Flawed Brownbridge garbage collector algorithm is corrected and extended by our garbage collectors.
- Significantly reduced the number of traversals required to detect garbage in dynamic concurrent garbage collection environment when compared to state of the art reference counted concurrent garage collectors.
- Introduced the first known distributed garbage collection algorithm with no global synchronization.
- Introduced the first known locality-based distributed garbage collection algorithm.
- Designed highly scalable garbage collectors for both shared and distributed memory enviornments.
- Introduced low memory cost, low communication cost, weight-based approach to convert the reference graph into a directed acyclic graph and thereby detect cycles faster.
- Both garbage collectors operate in linear time in quiescent state.

Bibliography

[1] Saleh E. Abdullahi and Graem A. Ringwood. Garbage collecting the internet: A survey of distributed garbage collection. *ACM Comput. Surv.*, 30(3):330–373, September 1998.

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

[3] 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.

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

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

[6] 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.

[7] Martin Berzins, Justin Luitjens, Qingyu Meng, Todd Harman, Charles A Wight, and Joseph R Peterson. Uintah: a scalable framework for hazard analysis. In *TG*, pages 3.1–3.8, 2010.

[8] DI Bevan. Distributed garbage collection using reference counting. In J.W. de Bakker, A.J. Nijman, and P.C. Treleaven, editors, *PARLE Parallel Architectures and Languages Europe*, volume 259 of *Lecture Notes in Computer Science*, pages 176–187. Springer Berlin Heidelberg, 1987.

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

[10] Steven R Brandt and Hari Krishnan. Multithreaded garbage collector. `https://github.com/stevenrbrandt/MultiThreadBrownbridge`, 2013.

[11] Steven R. Brandt, Hari Krishnan, Gokarna Sharma, and Costas Busch. Concurrent, parallel garbage collection in linear time. In *Proceedings of the 13th International Symposium on Memory Management*. ACM, 2014.

[12] 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.

[13] Albin M. Butters. Total cost of ownership: A comparison of C/C++ and Java. Technical report, Evans Data Corporation, June 2007.

[14] C. J. Cheney. A non-recursive list compacting algorithm. *CACM*, 13(11):677–8, November 1970.

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

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

[17] 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.

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

[19] 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. ACM.

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

[21] 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.

[22] Ionel Gog, Jana Giceva, Malte Schwarzkopf, Kapil Vaswani, Dimitrios Vytiniotis, Ganesan Ramalingam, Manuel Costa, Derek G Murray, Steven Hand, and Michael Isard. Broom: Sweeping out garbage collection from big data systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, 2015.

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

[24] Paul Hudak and Robert M. Keller. Garbage collection and task deletion in distributed applicative processing systems. In *Proceedings of the 1982 ACM Symposium on LISP and Functional Programming*, LFP '82, pages 168–178, New York, NY, USA, 1982. ACM.

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

[26] John Hughes. A distributed garbage collection algorithm. In *Proc. Of a Conference on Functional Programming Languages and Computer Architecture*, pages 256–272, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

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

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

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

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

[31] Hartmut Kaiser, Maciek Brodowicz, and Thomas Sterling. Parallex an advanced parallel execution model for scaling-impaired applications. In *ICPPW*, pages 394–401, 2009.

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

[33] R. Ladin and B. Liskov. Garbage collection of a distributed heap. In *Distributed Computing Systems, 1992., Proceedings of the 12th International Conference on*, pages 708–715, Jun 1992.

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

[35] Fabrice Le Fessant, Ian Piumarta, and Marc Shapiro. An implementation of complete, asynchronous, distributed garbage collection. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 152–161, New York, NY, USA, 1998. ACM.

[36] 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.

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

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

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

[40] Barbara Liskov and Rivka Ladin. Highly available distributed services and fault-tolerant distributed garbage collection. In *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing*, PODC '86, pages 29–39, New York, NY, USA, 1986. ACM.

[41] Martin Maas, Tim Harris, Krste Asanović, and John Kubiatowicz. Trash day: Coordinating garbage collection in distributed systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, 2015.

[42] Umesh Maheshwari and Barbara Liskov. Collecting cyclic distributed garbage by controlled migration. *Distrib. Comput.*, 10(2):79–86, 1997.

[43] Umesh Maheshwari and Barbara Liskov. Collecting cyclic distributed garbage by controlled migration. *Distributed Computing*, 10(2):79–86, 1997.

[44] Umesh Maheshwari and Barbara Liskov. Collecting distributed garbage cycles by back tracing. In *Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing (PODC)*, pages 239–248, 1997.

[45] Umesh Maheshwari and Barbara Liskov. Collecting distributed garbage cycles by back tracing. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '97, pages 239–248, New York, NY, USA, 1997. ACM.

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

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

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

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

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

[51] 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.

[52] David Peleg. *Distributed Computing: A Locality-sensitive Approach*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[53] 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.

[54] José M. Piquer. Indirect reference counting: A distributed garbage collection algorithm. In *Proceedings on Parallel Architectures and Languages Europe : Volume I: Parallel Architectures and Algorithms: Volume I: Parallel Architectures and Algorithms*, PARLE '91, pages 150–165, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

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

[56] David Plainfossé and Marc Shapiro. A survey of distributed garbage collection techniques. In *Proceedings of the International Workshop on Memory Management*, IWMM '95, pages 211–249, London, UK, UK, 1995. Springer-Verlag.

[57] David Plainfossé and Marc Shapiro. A survey of distributed garbage collection techniques. In *Proceedings of the International Workshop on Memory Management*, IWMM '95, pages 211–249, London, UK, UK, 1995. Springer-Verlag.

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

[59] N. Richer and M. Shapiro. The memory behavior of www, or the www considered as a persistent store. In *POS*, pages 161–176, 2000.

[60] 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.

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

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

[63] 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.

[64] Marc Shapiro, Peter Dickman, and David Plainfossé. Robust, distributed references and acyclic garbage collection. In *Proceedings of the Eleventh Annual ACM Symposium on Principles of Distributed Computing*, PODC '92, pages 135–146, New York, NY, USA, 1992. ACM.

[65] L. Veiga and P. Ferreira. Asynchronous complete distributed garbage collection. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pages 24a–24a, April 2005.

[66] L. Veiga and P. Ferreira. Asynchronous complete distributed garbage collection. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pages 24a–24a, April 2005.

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

[68] Wei-Jen Wang and Carlos A. Varela. Distributed garbage collection for mobile actor systems: the pseudo root approach. In *Proceedings of the First international conference on Advances in Grid and Pervasive Computing (GPC)*, pages 360–372, 2006.

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

Vita

Hari Krishnan was born in Villupuram, Tamil Nadu, India, in 1989. He obtained his bachelor of technology degree in Information Technology in 2010 from Anna University, Chennai, India. He started his Ph.D. in computer science in Louisiana State University in Spring 2011. Hari Krishnan will formally receive his Doctor of Philosophy degree in August 2016. His research interests are parallel and distributed computing, algorithms, theory and programming languages.