Concurrent Cycle Collection
in Reference Counted Systems

David F. Bacon and V.T. Rajan

IBM T.J. Watson Research Center
P.O. Box 704, Yorktown Heights, NY 10598, U.S.A.
`dfb@watson.ibm.com` `vtrajan@us.ibm.com`

Abstract. Automatic storage reclamation via reference counting has important advantages, but has always suffered from a major weakness due to its inability to reclaim cyclic data structures.

We describe a novel cycle collection algorithm that is both *concurrent* — it is capable of collecting garbage even in the presence of simultaneous mutation — and *localized* — it never needs to perform a global search of the entire data space. We describe our algorithm in detail and present a proof of correctness.

We have implemented our algorithm in the Jalapeño Java virtual machine as part of the Recycler, a concurrent multiprocessor reference counting garbage collector that achieves maximum mutator pause times of only 6 milliseconds. We present measurements of the behavior of the cycle collection algorithm over a set of eight benchmarks that demonstrate the effectiveness of the algorithm at finding garbage cycles, handling concurrent mutation, and eliminating global tracing.

1   Introduction

Forty years ago, two methods of automatic storage reclamation were introduced: reference counting [7] and tracing [23]. Since that time tracing collectors and their variants (mark-and-sweep, semispace copying, mark-and-compact) have been much more widely used due to perceived deficiencies in reference counting.

Changes in the relative costs of memory and processing power, and the adoption of garbage collected languages in mainstream programming (particularly Java) have changed the landscape. We believe it is time to take a fresh look at reference counting, particularly as processor clock speeds increase while RAM becomes plentiful but not significantly faster. In this environment the locality properties of reference counting are appealing, while the purported extra processing power required is likely to be less relevant.

At the same time, Java's incorporation of garbage collection has thrust the problem into the mainstream, and large, mission critical systems are being built in Java, stressing the flexibility and scalability of the underlying garbage collection implementations. As a result, the supposed advantages of tracing collectors — simplicity and low overhead — are being eroded as they are being made ever more complex in an attempt to address the real-world requirements of large and varied programs.

Furthermore, the fundamental assumption behind tracing collectors, namely that it is acceptable to periodically trace all of the live objects in the heap, will not necessarily scale to the very large main memories that are becoming increasingly common.

There are three primary problems with reference counting, namely:

1. storage overhead associated with keeping a count for each object;
2. run-time overhead of incrementing and decrementing the reference count each time a pointer is copied; and
3. inability to detect cycles and consequent necessity of including a second garbage collection technique to deal with cyclic garbage.

The inability to collect cycles is generally considered to be the greatest weakness of reference counting collectors. It either places the burden on the programmer to break cycles explicitly, or requires special programming idioms, or requires a tracing collector to collect the cycles.

In this paper, we present first a synchronous and then a concurrent algorithm for the collection of cyclic garbage in a reference counted system. The concurrent algorithm is a variant of the synchronous algorithm with additional tests to maintain safety properties that could be undermined by concurrent mutation of the data structures.

Like algorithms based on tracing (mark-and-sweep, semispace copying, and mark-and-compact) our algorithms are linear in the size of the graph traced. However, our algorithms are able to perform this tracing locally rather than globally, and often trace a smaller subgraph.

These algorithms have been implemented in a new reference counting collector, the Recycler, which is part of the Jalapeño Java VM [1] implemented at the IBM T.J. Watson Research Center. Jalapeño is itself written in Java.

In concurrently published work [3] we describe the Recycler as a whole, and provide measurements showing that our concurrent reference counting system achieves maximum measured mutator pause times of only 6 milliseconds. End-to-end execution times are usually comparable to those of a parallel (but non-concurrent) mark-and-sweep collector, although there is occasionally significant variation (in both directions).

In this paper we concentrate on describing the cycle collection algorithm in sufficient detail that it can be implemented by others, and give a proof of correctness which gives further insight into how and why the concurrent algorithm works. We also provide measurements of the performance of the cycle collection algorithms for a suite of eight Java benchmarks.

The rest of the paper is organized as follows: Section 2 describes previous approaches to cycle collection; Section 3 describes our synchronous algorithm for collection of cyclic garbage; Section 4 then presents our concurrent cycle collection algorithm. Section 5 contains proofs of correctness for the concurrent cycle collection algorithms. Section 6 presents our measurements of the effectiveness of the algorithms. Section 7 describes related work on concurrent garbage collection. Finally, we present our conclusions.

Subsections 3.1 and 4.4 contain detailed pseudocode of the algorithms and can be skipped on a first reading of the paper.

2   Previous Work on Cycle Collection

Previous work on solving the cycle collection problem in reference counted collectors has fallen into three categories:

- special programming idioms, like Bobrow's groups [5], or certain functional programming styles;
- use of an infrequently invoked tracing collector to collect cyclic garbage [8]; or
- searching for garbage cycles by removing internal reference counts [6, 22].

An excellent summary of the techniques and algorithms is in chapter 3 ("Reference Counting") of the book by Jones and Lins [17]. The first algorithm for cycle collection in a reference counted system was devised by Christopher [6]. Our synchronous cycle collection algorithm is based on the work of Martínez et al [22] as extended by Lins [20], which is very clearly explained in the chapter of the book just mentioned.

There are two observations that are fundamental to these algorithms. The first observation is that garbage cycles can only be created when a reference count is decremented to a non-zero value — if the reference count is incremented, no garbage is being created, and if it is decremented to zero, the garbage has already been found. Furthermore, since reference counts of one tend to predominate, decrements to zero should be common.

The second observation is that in a garbage cycle, all the reference counts are internal; therefore, if those internal counts can be subtracted, the garbage cycle will be discovered.

As a result, when a reference count is decremented and does not reach zero, it is considered as a candidate root of a garbage cycle, and a local search is performed. This is a depth-first search which subtracts out counts due to internal pointers. If the result is a collection of objects with zero reference counts, then a garbage cycle has been found and is collected; if not, then another depth-first-search is performed and the counts are restored.

Lins [20] extended the original algorithm to perform the search lazily by buffering candidate roots instead of exploring them immediately. This has two advantages. Firstly, after a time, the reference count of a candidate root may reach zero due to other edge deletions, in which case the node can simply be collected, or the reference count may be re-incremented due to edge additions, in which case it may be ignored as a candidate root. Secondly, it will often prevent re-traversal of the same node.

Unfortunately, in the worst case Lins' algorithm is quadratic in the size of the graph, as for example in the cycle shown in Figure 1. His algorithm considers the roots one at a time, performing the reference count subtraction and restoration passes for that root before moving on.

Therefore, Lins' algorithm will perform a complete scan from each of the candidate roots until it arrives at the final root, at which point the entire compound cycle will be collected.

3   Synchronous Cycle Collection

In this section we describe our synchronous cycle collection algorithm, which applies the same principles as those of Martínez et al and Lins, but which only requires $O(N + E)$ worst-case time for collection (where N is the number of nodes and E is the number of edges in the object graph), and is therefore competitive with tracing garbage collectors.
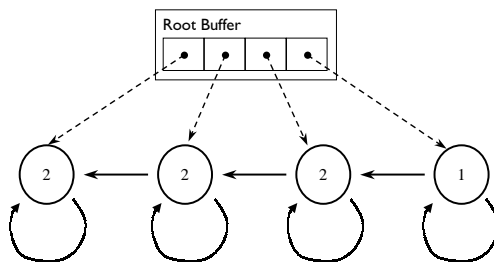
Fig. 1. Example of compound cycle that causes Lins' algorithm to exhibit quadratic complexity.

We also improve the practicality of the algorithm by allowing resizing of collected objects, and show how significant constant-time improvements can be achieved by ruling out inherently acyclic data structures.

Our synchronous algorithm is similar to Lins' algorithm: when reference counts are decremented, we place potential roots of cyclic garbage into a buffer called `Roots`. Periodically, we process this buffer and look for cycles by subtracting internal reference counts.

There are two major changes that make the algorithm linear time: first of all, we add a *buffered* flag to every object, which is used to prevent the same object being added to the root buffer more than once per cycle collection. This in turn places a linear bound on the size of the buffer.

Secondly, we analyze the entire transitive closure of `Roots` as a single graph, rather than as a set of graphs. This means that the complexity of the algorithm is limited by the size of that transitive closure, which in turn is limited by $N + E$ (since `Roots` is bounded by N by the use of the buffered flag). Of course, in practice we hope that the transitive closure will be significantly smaller.

In practice we found that the first change (the use of the buffered flag) made almost no difference in the running time of the algorithm; however, the second change (analyzing the entire graph at once) made an enormous difference in run-time. When we applied Lins' algorithm unmodified to large programs, garbage collection delays extended into minutes.

3.1 Pseudocode and Explanation

We now present detailed pseudocode and an explanation of the operation of each procedure in the synchronous cycle collection algorithm.

In addition to the buffered flag, each object contains a color and a reference count. For an object `T` these fields are denoted `buffered(T)`, `color(T)`, and `RC(T)`. In the implementation, these quantities together occupy a single word in each object.

Color	Meaning
Black	In use or free
Gray	Possible member of cycle
White	Member of garbage cycle
Purple	Possible root of cycle
Green	Acyclic
Red	Candidate cycle undergoing Σ-computation
Orange	Candidate cycle awaiting epoch boundary

Table 1. Object Colorings for Cycle Collection. Orange and red are only used by the concurrent cycle collector and are described in Section 4.

All objects start out black. A summary of the colors used by the collector is shown in Table 1. The use of green (acyclic) objects will be discussed below.

The algorithm is shown in Figure 2. The procedures are explained in detail below. `Increment` and `Decrement` are invoked externally as pointers are added, removed, or overwritten. `CollectCycles` is invoked either when the root buffer overflows, storage is exhausted, or when the collector decides for some other reason to free cyclic garbage. The rest of the procedures are internal to the cycle collector. Note that the procedures `MarkGray`, `Scan`, and `ScanBlack` are the same as for Lins' algorithm.

`Increment(S)` When a reference to a node `S` is created, the reference count of `T` is incremented and it is colored black, since any object whose reference count was just incremented can not be garbage.

`Decrement(S)` When a reference to a node `S` is deleted, the reference count is decremented. If the reference count reaches zero, the procedure `Release` is invoked to free the garbage node. If the reference count does not reach zero, the node is considered as a possible root of a cycle.

`Release(S)` When the reference count of a node reaches zero, the contained pointers are deleted, the object is colored black, and unless it has been buffered, it is freed. If it has been buffered, it is in the `Roots` buffer and will be freed later (in the procedure `MarkRoots`).

`PossibleRoot(S)` When the reference count of `S` is decremented but does not reach zero, it is considered as a possible root of a garbage cycle. If its color is already purple, then it is already a candidate root; if not, its color is set to purple. Then the *buffered* flag is checked to see if it has been purple since we last performed a cycle collection. If it is not buffered, it is added to the buffer of possible roots.

`CollectCycles()` When the root buffer is full, or when some other condition, such as low memory occurs, the actual cycle collection operation is invoked. This operation has three phases: `MarkRoots`, which removes internal reference counts; `ScanRoots`, which restores reference counts when they are non-zero; and finally `CollectRoots`, which actually collects the cyclic garbage.

`MarkRoots()` The marking phase looks at all the nodes `S` whose pointers have been stored in the `Roots` buffer since the last cycle collection. If the color of the node is purple (indicating a possible root of a garbage cycle) and the reference

```
Increment(S)
  RC(S) = RC(S) + 1
  color(S) = black

Decrement(S)
  RC(S) = RC(S) - 1
  if (RC(S) == 0)
    Release(S)
  else
    PossibleRoot(S)

Release(S)
  for T in children(S)
    Decrement(T)
  color(S) = black
  if (! buffered(S))
    Free(S)

PossibleRoot(S)
  if (color(S) != purple)
    color(S) = purple
    if (! buffered(S))
      buffered(S) = true
      append S to Roots

CollectCycles()
  MarkRoots()
  ScanRoots()
  CollectRoots()

MarkRoots()
  for S in Roots
    if (color(S) == purple)
      MarkGray(S)
    else
      buffered(S) = false
      remove S from Roots
      if (color(S) == black
          and RC(S) == 0)
        Free(S)

ScanRoots()
  for S in Roots
    Scan(S)
```

```
ScanRoots()
  for S in Roots
    Scan(S)

CollectRoots()
  for S in Roots
    remove S from Roots
    buffered(S) = false
    CollectWhite(S)

MarkGray(S)
  if (color(S) != gray)
    color(S) = gray
    for T in children(S)
      RC(T) = RC(T) - 1
      MarkGray(T)

Scan(S)
  if (color(S) == gray)
    if (RC(S) > 0)
      ScanBlack(S)
    else
      color(S) = white
      for T in children(S)
        Scan(T)

ScanBlack(S)
  color(S) = black
  for T in children(S)
    RC(T) = RC(T) + 1
    if (color(T) != black)
      ScanBlack(T)

CollectWhite(S)
  if (color(S) == white
      and ! buffered(S))
    color(S) = black
    for T in children(S)
      CollectWhite(T)
    Free(S)
```

Fig. 2. Synchronous Cycle Collection

count has not become zero, then `MarkGray(S)` is invoked to perform a depth-first search in which the reached nodes are colored gray and internal reference counts are subtracted. Otherwise, the node is removed from the `Roots` buffer, the `buffered` flag is cleared, and if the reference count is zero the object is freed.

`ScanRoots()` For each node `S` that was considered by `MarkGray(S)`, this procedure invokes `Scan(S)` to either color the garbage subgraph white or re-color the live subgraph black.

`CollectRoots()` After the `ScanRoots` phase of the `CollectCycles` procedure, any remaining white nodes will be cyclic garbage and will be reachable from the `Roots` buffer. This procedure invokes `CollectWhite` for each node in the `Roots` buffer to collect the garbage; all nodes in the root buffer are removed and their `buffered` flag is cleared.

`MarkGray(S)` This procedure performs a simple depth-first traversal of the graph beginning at `S`, marking visited nodes gray and removing internal reference counts as it goes.

`Scan(S)` If this procedure finds a gray object whose reference count is greater than zero, then that object and everything reachable from it are live data; it will therefore call `ScanBlack(S)` in order to re-color the reachable subgraph and restore the reference counts subtracted by `MarkGray`. However, if the color of an object is gray and its reference count is zero, then it is colored white, and `Scan` is invoked upon its children. Note that an object may be colored white and then re-colored black if it is reachable from some subsequently discovered live node.

`ScanBlack(S)` This procedure performs the inverse operation of `MarkGray`, visiting the nodes, changing the color of objects back to black, and restoring their reference counts.

`CollectWhite(S)` This procedure recursively frees all white objects, re-coloring them black as it goes. If a white object is buffered, it is not freed; it will be freed later when it is found in the `Roots` buffer.

3.2 Acyclic Data Types

A significant constant-factor improvement can be obtained for cycle collection by observing that some objects are inherently acyclic. We speculate that they will comprise the majority of objects in many applications. Therefore, if we can avoid cycle collection for inherently acyclic objects, we will significantly reduce the overhead of cycle collection as a whole.

In Java, dynamic class loading complicates the determination of inherently acyclic data structures. We have implemented a very simple scheme as part of the class loader. Acyclic classes may contain:

- scalars;
- references to classes that are both acyclic and final; and
- arrays of either of the above.

Our implementation marks objects whose class is acyclic with the special color green. Green objects are ignored by the cycle collection algorithm, except that when
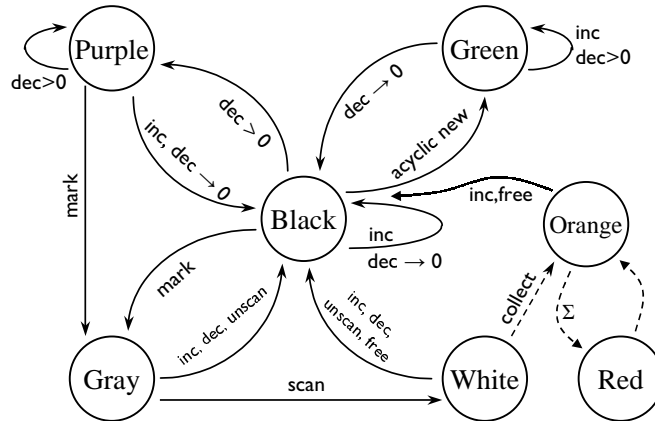
Fig. 3. State transition graph for cycle collection.

a dead cycle refers to green objects, they are collected along with the dead cycle. For simplicity of the presentation, we have not included consideration of green objects in the algorithms in this paper; the modifications are straightforward.

While our determination of acyclic classes is very simple, it is also very effective, usually reducing the objects considered as roots of cycles by an order of magnitude, as will be shown in Section 6. In a static compiler, a more sophisticated program analysis could be applied to increase the percentage of green objects.

4   Concurrent Cycle Collection

We now describe a concurrent cycle collection algorithm based on the principles of our synchronous algorithm of the previous section.

For the purposes of understanding the cycle collection algorithm, the multiprocessor reference counting system can be viewed very abstractly as follows: as mutators create and destroy references to objects (either on the stack or in the heap), corresponding increment and decrement operations are enqueued into a local buffer, called the *mutation buffer*. Periodically, the mutators send these mutation buffers to the *collector*, which applies the reference count updates, frees objects whose counts drop to zero, and periodically also performs cycle collection.

Time is divided into *epochs*, and each mutator must transfer its mutation buffer to the collector exactly once per epoch. However, aside from this requirement, in normal operation no synchronization is required between mutators and the collector.

When all mutation buffers for an epoch have been transferred to the collector, the increments for the just-completed epoch are applied; however, the decrements are not applied until the next epoch boundary. This prevents freeing of live data which might

otherwise occur due to race conditions between the mutators. The advantage of this approach is that it is never necessary to halt all the mutators simultaneously.

In its implementation, the Recycler only tracks pointer updates to the heap, and snapshots pointers in the stack at epoch boundaries. Our algorithm is similar to Deutsch-Bobrow deferred reference counting [9], but is superior in a number of important respects. Our implementation of concurrent reference counting is most similar to the reference counting collector of DeTreville [8]. The Recycler is described in detail by Bacon et al [3].

4.1   Two Phase Cycle Collection

Now that we have abstracted the concurrent system to a collection of mutators emitting streams of increment and decrement operations, and a reference counting collector which merges and applies these operations, we can describe in overview how the algorithm operates.

The concurrent cycle collection algorithm is more complex than the synchronous algorithm. As with other concurrent garbage collection algorithms, we must contend with the fact that the object graph may be modified simultaneously with the collector scanning it; but in addition the reference counts may be as much as a two epochs out of date (because decrements are deferred by an epoch).

Our algorithm relies on the same basic premise as the synchronous algorithm: namely, that given a subset of nodes, if deleting the internal edges between the nodes in this subset reduces the reference count of every node in the subset to zero, then the whole subset of nodes is cyclic garbage. (The subset may represent more than one independent cycles, but they are all garbage cycles.)

However, since the graph may be modified, we run into three basic difficulties. Firstly, since we can not rely on being able to retrace the same graph, the repeated traversal of the graph does not define the same set of nodes. Secondly, the deletion of edges can disconnect portions of the graph, thus making the global test by graph traversal difficult. Thirdly, reference counts may be out of date.

Our algorithm proceeds in two phases. In the first phase, we use a variant of the synchronous algorithm as described in Section 3 to obtain a candidate set of garbage nodes. We then wait until an epoch boundary and then perform the second phase in which we test these to ensure that the candidates do indeed satisfy the criteria for garbage cycles.

The two phases can be viewed as enforcing a "liveness" and a "safety" property. The first phase enforces liveness by ensuring that potential garbage cycles are considered for collection. The second phase ensures safety by preventing the collection of false cycles induced by concurrent mutator activity.

4.2   Liveness: Finding Cycles to Collect

Essentially, we use the synchronous algorithm to find candidate cycles. However, due to concurrent mutator activity, the graph may be changing and the algorithm may produce incorrect results.

To perform the concurrent cycle collection, we need a second reference count for each object, denoted `CRC(S)`. This is a hypothetical reference count which may become incorrect due to concurrent mutator activity. In the implementation, we are able to fit both reference counts, the color, and the buffered flag into a single header word by using a hash table to hold count overflows, which occur very rarely.

The liveness phase of the concurrent algorithm proceeds in a similar manner to the synchronous cycle collection algorithm, except that when an object is marked gray its cyclic reference count (`CRC`) is initialized to its true reference count — the "true" reference count (`RC`) is not changed. Henceforward, the mark, scan, and collect phases operate upon the cyclic reference count instead of the true reference count. In the `CollectWhite` procedure, instead of collecting the white nodes as garbage, we color them orange and add them to a set of *possible* garbage.

By using the cyclic reference count we ensure that in the event of concurrent mutator activity, the information about the true reference count of the objects is never lost.

In absence of mutator activity, the liveness phase will yield the set of garbage nodes, and the safety phase will certify that this indeed is a set of garbage of nodes and we can collect them.

However, the presence of concurrent mutator activity can cause live nodes to enter the list in three different ways. Firstly, the mutator can add an edge, thus causing the `MarkGray` procedure to incorrectly infer that there are no external edges to a live object. Secondly, the mutator can delete an edge, thus causing scan procedure to incorrectly infer a live object to be garbage. Thirdly, the deletion of edges concurrent to running of the `MarkGray` and scan procedure can create gray and white nodes with various values of cyclic reference counts. While eventually the reporting of the mutator activity will cause these nodes to be detected and re-colored, if these nodes are encountered before they are re-colored they can mislead the runs of the above procedures into inferring that they are garbage.

The output of phase one is a set of nodes believed to be garbage in the `Cycle-Buffer` data structure. The `CycleBuffer` is divided into discrete connected components, each of which forms a potential garbage cycle. Due to mutator activity, the contents of the `CycleBuffer` can be a superset of the actual set of garbage nodes and can contain some nodes that fail tests in the safety phase (this is discussed in detail in Section 5).

4.3   Safety: Collecting Cycles Concurrently

The second ("safety") phase of the algorithm takes as input a set of nodes and determines whether they form a garbage cycle. These nodes are marked with a special color, orange, which is used to identify a candidate set in the concurrent cycle collector.

The safety phase of the algorithm consists of two tests we call the Σ-test and the Δ-test. If a subset of nodes of the object graph passes both the Σ-test and the Δ-test, then we can be assured that the nodes in the subset are all garbage. Thus, correctness of the safety phase of our algorithm is not determined by any property of the output of the liveness phase which selects the subgraphs. This property of the safety phase of the algorithm considerably simplifies the proof of correctness as well as modularizing the code.

In theory, it would be possible to build a cycle collector which simply passed random sets of nodes to the safety phase, which would then either accept them as garbage or reject them as live. However, such a collector would not be practical: if we indeed pick a random subset of nodes from the object graph, the chances that they form a complete garbage subgraph is very small. The job of the liveness phase can be seen as finding likely sets of candidates for garbage cycles. If the mutator activity is small in a given epoch, this would indeed be very likely to be true.

The Σ-test consists of two parts: a *preparation* and an actual *test*. In the preparation part, which is performed immediately after the candidate cycles have been found, we iterate over the subset and initialize the cyclic reference count of every node in the subset to the reference count of the node. Then we iterate over every node in the subset again and decrement the cyclic reference count of any children of the node that are also in the subset. At the end of the preparation computation, the cyclic reference count of each node in the subset represents the number of references to the node from nodes external to the subset. In the actual test, which is performed after the next epoch boundary, we iterate over every node in the subset and test if its cyclic reference count is zero.

If it is zero for every member of the set, then we know that there exists no reference to this subset from any other node. Therefore, any candidate set that passes the Σ-test is garbage, unless the reference count used during the running of the preparation procedure is outdated due to an increment to one of the nodes in the subset.

This is ascertained by the Δ-test. We wait until the next epoch boundary, at which point increment processing re-colors all non-black nodes and their reachable subgraphs black. Then we scan the nodes in the candidate set and test whether their color is still orange. If they are all orange, we know that there has been no increment to the reference count during the running of the preparation procedure and we say that the candidate set passed the Δ-test.

Any subset of garbage nodes that does not have any external pointers to it will pass both the tests. Note that we do not have to worry about concurrent decrements to the members of the subset, since it is not possible for the reference count of any node to drop below zero.

However, it is possible for a set of garbage to have pointers to it from other garbage cycles. For example in Figure 1, only the candidate set consisting of the last node forms isolated garbage cycle. The other cycles have pointers to them from the cycle to their right.

We know that the garbage cycles in the cycle buffer cannot have any forward pointers to other garbage cycles (if they did, we would have followed them and included them in a previous garbage cycle). Hence, we process the candidate cycles in the cycle buffer in the reverse of the order in which we found them. This reasoning is described more formally in Lemma 3 in Section 5.

When a candidate set passes both tests, and hence is determined to be garbage, then we free the nodes in the cycle, which causes the reference counts of other nodes outside of the cycle to be decremented. By the stability property of garbage, we can decrement such reference counts without concern for concurrent mutation.

When we decrement a reference count to an orange node, we also decrement its cyclic reference count (CRC). Therefore, when the next candidate cycle is considered

(the previous cycle in the buffer), if it is garbage the Σ-test will succeed because we have augmented the computation performed by the preparation procedure.

Hence when we reach a candidate set, the cyclic reference count does not include the count of any pointers from a known garbage node. This ensures that all the nodes in Figure 1 would be collected.

A formalism for understanding the structure of the graph in the presence of concurrent mutation, and a proof of correctness of the algorithm is presented in Section 5.

4.4   Pseudocode and Explanation

We now present the pseudocode with explanations for each procedure in the concurrent cycle collection algorithm. The pseudocode is shown in Figures 4 and 5. The operation of `CollectCycles` and its subsidiary procedures is very similar to the operation of the synchronous algorithm of Figure 2, so for those procedures we will only focus on the differences in the concurrent versions of the procedures.

`Increment(S)` The true reference count is incremented. Since the reference count is being incremented, the node must be live, so any non-black objects reachable from it are colored black by invoking `ScanBlack`. This has the effect of re-blackening live nodes that were left gray or white when concurrent mutation interrupted a previous cycle collection.

`Decrement(S)` At the high level, decrementing looks the same as with the synchronous algorithm: if the count becomes zero, the object is released, otherwise it is considered as a possible root.

`PossibleRoot(S)` For a possible root, we first perform `ScanBlack`. As with `Increment`, this has the effect of re-blackening leftover gray or white nodes; it may also change the color of some purple nodes reachable from `S` to black, but this is not a problem since they will be considered when the cycle collector considers `S`. The rest of `PossibleRoot` is the same as for the synchronous algorithm.

`ProcessCycles()` Invoked once per epoch after increment and decrement processing due to the mutation buffers from the mutator threads has been completed. First, `FreeCycles` attempts to free candidate cycles discovered during the previous epoch. Then `CollectCycles` collects new candidate cycles and `Sigma-Preparation` prepares for the Σ-test to be run in the next epoch.

`CollectCycles()` As in the synchronous algorithm, three phases are invoked on the candidate roots: marking, scanning, and collection.

`MarkRoots()` This procedure is the same as in the synchronous algorithm.

`ScanRoots()` This procedure is the same as in the synchronous algorithm.

`CollectRoots()` For each remaining root, if it is white a candidate cycle has been discovered starting at that root. The `CurrentCycle` is initialized to be empty, and the `CollectWhite` procedure is invoked to gather the members of the cycle into the `CurrentCycle` and color them orange. The collected cycle is then appended to the `CycleBuffer`. If the root is not white, a candidate cycle was not found from this root or it was already included in some previously collected candidate, and the buffered flag is set to false. In either case, the root is removed from the `Roots` buffer, so that at the end of this procedure the `Roots` buffer is empty.

```
Increment(S)
  RC(S) = RC(S) + 1
  ScanBlack(S)

Decrement(S)
  RC(S) = RC(S) - 1
  if (RC(S) == 0)
    Release(S)
  else
    PossibleRoot(S)

Release(S)
  for T in children(S)
    Decrement(T)
  color(S) = black
  if (! buffered(S))
    Free(S)

PossibleRoot(S)
  ScanBlack(S)
  color(S) = purple
  if (! buffered(S))
    buffered(S) = true
    append S to Roots

ProcessCycles()
  FreeCycles()
  CollectCycles()
  SigmaPreparation()

CollectCycles()
  MarkRoots()
  ScanRoots()
  CollectRoots()

MarkRoots()
  for S in Roots
    if (color(S) == purple
        and RC(S) > 0)
      MarkGray(S)
    else
      remove S from Roots
      buffered(S) = false
      if (RC(S) == 0)
        Free(S)
```

```
ScanRoots()
  for S in Roots
    Scan(S)

CollectRoots()
  for S in Roots
    if (color(S) == white)
      CurrentCycle = empty
      CollectWhite(S)
      append CurrentCycle
        to CycleBuffer
    else
      buffered(S) = false
    remove S from Roots

MarkGray(S)
  if (color(S) != gray)
    color(S) = gray
    CRC(S) = RC(S)
    for T in children(S)
      MarkGray(T)
  else if (CRC(S) > 0)
    CRC(S) = CRC(S) - 1

Scan(S)
  if (color(S) == gray
      and CRC(S) == 0)
    color(S) = white
    for T in children(S)
      Scan(T)
  else
    ScanBlack(S)

ScanBlack(S)
  if (color(S) != black)
    color(S) = black
    for T in children(S)
      ScanBlack(T)

CollectWhite(S)
  if (color(S) == white)
    color(S) = orange
    buffered(S) = true
    append S to CurrentCycle
    for T in children(S)
      CollectWhite(T)
```

Fig. 4. Concurrent Cycle Collection Algorithm (Part 1)

```
SigmaPreparation()
  for C in CycleBuffer
    for N in C
      color(N) = red
      CRC(N) = RC(N)
    for N in C
      for M in children(N)
        if (color(M) == red
            and CRC(M) > 0)
          CRC(M) = CRC(M)-1
    for N in C
      color(N) = orange

FreeCycles()
  last = |CycleBuffer|-1
  for i = last to 0 by -1
    C = CycleBuffer[i]
    if (DeltaTest(C)
        and SigmaTest(C))
      FreeCycle(C)
    else
      Refurbish(C)
  clear CycleBuffer

DeltaTest(C)
  for N in C
    if (color(N) != orange)
      return false
  return true

SigmaTest(C)
  externRC = 0
  for N in C
    externRC = externRC+CRC(N)
  return (externRC == 0)
```

```
Refurbish(C)
  first = true
  for N in C
    if ((first and
         color(N)==orange) or
        color(N)==purple)
      color(N) = purple
      append N to Roots
    else
      color(N) = black
      buffered(N) = false
    first = false

FreeCycle(C)
  for N in C
    color(N) = red
  for N in C
    for M in children(N)
      CyclicDecrement(M)
  for N in C
    Free(N)

CyclicDecrement(M)
  if (color(M) != red)
    if (color(M) == orange)
      RC(M) = RC(M) - 1
      CRC(M) = CRC(M) - 1
    else
      Decrement(M)
```

Fig. 5. Concurrent Cycle Collection Algorithm (Part 2)

`MarkGray(S)` This is similar to the synchronous version of the procedure, with adaptations to use the cyclic reference count (CRC) instead of the true reference count (RC). If the color is not gray, it is set to gray and the CRC is copied from the RC, and then `MarkGray` is invoked recursively on the children. If the color is already gray, and if the CRC is not already zero, the CRC is decremented (the check for non-zero is necessary because concurrent mutation could otherwise cause the CRC to underflow).

`Scan(S)` As with `MarkGray`, simply an adaptation of the synchronous procedure that uses the CRC. Nodes with zero CRC are colored white; non-black nodes with CRC greater than zero are recursively re-colored black.

`ScanBlack(S)` Like the synchronous version of the procedure, but it does not need to re-increment the true reference count because all reference count computations were carried out on the CRC.

`CollectWhite(S)` This procedure recursively gathers white nodes identified as members of a candidate garbage cycle into the `CurrentCycle` and colors them orange as it goes. The `buffered` flag is also set true since a reference to the node will be stored in the `CycleBuffer` when `CurrentCycle` is appended to it.

`SigmaPreparation()` After the candidate cycles have been collected into the `CycleBuffer`, this procedure prepares for the execution of the Σ-test in the next epoch. It operates individually on each candidate cycle `C`. First, each node `S` in `C` has its CRC initialized to its RC and its color set to red. After this only the nodes of `C` are red. Then for any pointer from one node in `C` to another node in `C`, the CRC of the target node is decremented. Finally, the nodes in `C` are re-colored orange. At the end of `SigmaPreparation`, the CRC field of each node `S` contains a count of the number of references to `S` from outside of `C`.

`FreeCycles()` This procedure iterates over the candidate cycles in the reverse order in which they were collected. It applies the safety tests (the Σ-test and the Δ-test) to each cycle and if it passes both tests then the cycle is freed; otherwise it is refurbished, meaning that it may be reconsidered for collection in the next epoch.

`DeltaTest(C)` This procedure returns true if the color of all nodes in the cycle are orange, which indicates that their have been no increments to any of the nodes in the cycle.

`SigmaTest(C)` This procedure calculates the total number of external references to nodes in the cycle, using the CRC fields computed by the `SigmaPreparation` procedure. It returns true if the number of external references is zero, false otherwise.

`Refurbish(C)` If the candidate cycle has not been collected due to failing a safety test, this procedure re-colors the nodes. If the first node in the candidate cycle (which was the purple node from which the candidate was found) is still orange, or if any node has become purple, then those nodes are colored purple and placed in the `Roots` buffer. All other nodes are colored black and their `buffered` flags are cleared.

`FreeCycle(C)` This procedure actually frees the members of a candidate cycle that has passed the safety tests. First, the members of `C` are colored red; after this, only the nodes in `C` are red. Then for each node `S` in `C`, `CyclicDecrement` decrements reference counts in non-red nodes pointed to by `S`.

`CyclicDecrement(M)` If a node is not red, then it either belongs to some other candidate cycle or not. If it belongs to some other candidate cycle, then it is orange, in which case both the RC and the CRC fields are decremented (the CRC field is decremented to update the computation performed previously by the `Sigma-Preparation` procedure to take the deletion of the cycle pointing to `M` into account). If it does not belong to some other candidate cycle, it will not be orange and a normal `Decrement` operation is performed.

For ease of presentation, we have presented the pseudocode in a way that maximizes readability. However, this means that as presented the code makes more passes over the nodes than is strictly necessary. For instance, the first pass by `SigmaPreparation` can be merged with `CollectWhite`, and the passes performed by `DeltaTest` and `SigmaTest` can be combined. In the implementation, the passes are combined to minimize constant-factor overheads.

5   Proofs

In this section we prove the correctness of the concurrent cycle collection algorithm presented in Section 4.

5.1   The Abstract Graph

For the purpose of the proof of correctness of the above tests for garbage it is useful to define an abstract graph G_i for the ith epoch of the garbage collector. At beginning of each epoch the collector thread gets a set of increments and decrements from each of the mutator threads. If the increment refers to a new node, it implies the creation of that node. In addition, each increment implies addition of a directed edge between two nodes and each decrement implies deletion of an edge. The increments and decrements do not provide the source of the edges, so in practice we cannot build this graph, nor do we need to build it for the purpose of the algorithm. But for the purposes of the proof it is useful to conceptualize this graph. The graph G_i denotes the graph that is generated by adding nodes for each reference to a new node, and inserting and deleting edges to G_{i-1} corresponding to the increments and decrements at the beginning of ith epoch. In addition, when a node is determined to be garbage and is freed, it is deleted from G_i. At the beginning of the first epoch we start with an empty graph G_0.

We can similarly define an abstract set of roots R_i for each epoch i. The roots are either in the mutator stacks or in global (class static) variables. The roots in the mutator stacks are named by the increments collected from the stack snapshots of each mutator for the epoch. The roots from the global variables are the sources in edges implied by increment and decrement operations whose source is a global variable instead of a heap variable. R_i is simply the union of these two types of roots.

Given G_i and R_i, we can then define the set of garbage objects in G_i, which we denote Γ_i, as

$$\Gamma_i = G_i - R_i^\star$$

that is, the set difference G_i minus the transitive closure of the roots R_i.

5.2   Safety: Proof of Correctness

A garbage collector is *safe* if every object collected is indeed garbage. In this section we prove the safety of our algorithm.

At the end of epoch $i + 1$, the procedure ProcessCycles invokes FreeCycles to collect the cycles identified as potential garbage during epoch i.

Let the set B_i denote the contents of the CycleBuffer generated during the cycle collection in epoch i. This is the collection of orange nodes generated by the concurrent variant of the synchronous cycle detection algorithm, which used the set of purple nodes (denoted P_i) as roots to search for cyclic garbage.

B_i is partitioned into the disjoint sets $B_{i1} \ldots B_{in}$. Each B_{ik} is a candidate garbage cycle computed by the cycle collection algorithm from a particular purple node in P_i.

Due to concurrent mutation, B_i may contain nodes from $G_i \cup G_{i+1}$.

Lemma 1. *Any set B_{ik} containing nodes that do not exist in G_i (that is, $B_{ik} - G_i \neq \emptyset$) will fail the Δ-test.*

Proof. The only way for new nodes to be added to G_i in G_{i+1} is by increment operations. However, all concurrent increment operations will have been processed before we apply the Δ-test. Processing an increment operation invokes ScanBlack, so that if the node in question is in B_{ik}, then that node (at least) will be re-colored from orange to black. The presence of that black node in B_{ik} will cause the Δ-test to fail. □

Therefore, for a given epoch i with G_i, R_i, and Γ_i, let

Ω_i denote the set containing the sets $B_{ik} \in B_i$ that passed the Δ-test. The sets in Ω_i are denoted Ω_{ik}. Since all Ω_{ik} have passed the Δ-test, all $\Omega_{ik} \subseteq G_i$.

C denote one of the sets Ω_{ik}, namely a set of nodes believed to be a garbage cycle that has passed the Δ-test.

S denote a specific node in that collection ($S \in C$).

$RC(S)$ denote the reference count at a node S in the ith epoch. By definition RC(S) is the reference count of node S in graph G_i.

$RC(S, C)$ denote the number of references to S from nodes within C.

$RC^{\Sigma}(S, C)$ denote the number of references to S from nodes within C as determined by the Σ-test.

$CRC(S)$ denote the hypothetical reference count for $S \in C$ as computed by the Σ-test.

$\overline{C}$ denote $G_i - C$, the complement of C in G_i.

Theorem 1. *If C passes the Σ-test, that is if we have computed the values of $CRC(S)$ for each $S \in C$ as described in the procedure SigmaPreparation in Section 4 and*

$$\sum_{S \in C} CRC(S) = 0$$

then C is a set of garbage nodes ($C \subseteq \Gamma_i$).

Proof. From the above definitions, for every $S \in C$,

$$RC(S) = RC(S, C) + RC(S, \overline{C})$$

Since we delay the processing of the decrements by one epoch, this ensures that the following properties are true:

> $RC(S)$ is non-negative and if it is zero, then S is a garbage node.

> $RC(S, \overline{C})$ is non-negative and if it is zero for every node S in C then C is a collection of garbage nodes.

During the Σ-test, we determine the number of references to node S from nodes within C. Therefore by definition,

$$CRC(S) = RC(S) - RC^{\Sigma}(S, C).$$

The $RC^{\Sigma}(S, C)$ may differ from the $RC(S, C)$ because there may be new references to S from nodes within C that were added to G_i, thereby increasing $RC^{\Sigma}(S, C)$; or because there may be references to S from nodes in C that were deleted from G_i, thereby decreasing $RC^{\Sigma}(S, C)$. If the collection passes the Δ-test, then no references were added to S at any time during the last epoch.

Therefore,

$$RC(S, C) \geq RC^{\Sigma}(S, C)$$

and

$$\begin{aligned} CRC(S) = RC(S) - RC^{\Sigma}(S, C) \\ \geq \; RC(S) - RC(S, C) \; = RC(S, \overline{C}) \end{aligned}$$

If $CRC(S)$ is zero from the Σ-test, since $RC(S, \overline{C})$ is non-negative, it follows that $RC(S, \overline{C})$ has to be zero too. Further, if $RC(S, \overline{C}) = 0$ for every node in the collection C, then the whole collection C is garbage. □

Lemma 1 and Theorem 1 show that the Δ-test and the Σ-test are sufficient to ensure that any set C of nodes that passes both tests is garbage. The following theorem shows that both the tests are necessary to ensure this as well.

Theorem 2. *Both Δ-test and Σ-test are necessary to ensure that a candidate set B_{ij} contains only garbage nodes.*

Proof. We will prove by example. Consider the graph of nodes shown in Figure 6 (a). The cycle was detected from the purple node a, which is the starting point from which cycle collection is run. If the edge between nodes c and d is cut between the MarkGray and the Scan routines, then the nodes a and b will be collected by the CollectWhite routine and form a set B_{ij}. These nodes are not garbage. However, since there have been no increments to the reference counts of either of these nodes, this set will pass Δ-test.

The decrements will be processed an epoch later, at epoch $i + 2$, so the decrement to node d will not have an effect on the nodes a and b in the FreeCycles operation performed in epoch $i + 1$. Even waiting for an additional epoch does not guarantee that the fact that nodes a and b will be detected by Δ-test, since during epoch $i + 1$ the edge
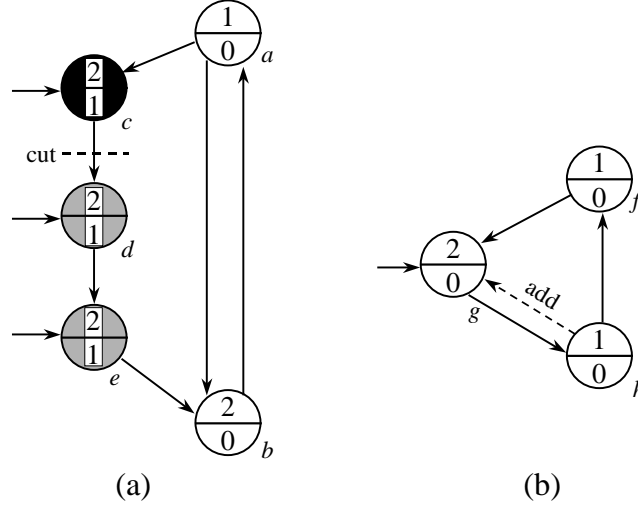
Fig. 6. Race conditions uniquely detected (a) by the Σ-test, and (b) by the Δ-test. The purple nodes from which cycle collection started were a and f. Inside of each node is shown the reference count, or RC (top) and the cyclic reference count, or CRC (bottom).

from d to e could be cut. Indeed, by making the chain of nodes $\{c, d, e\}$ arbitrarily long and having a malicious mutator cut edges at just the right moment, we can have the non-garbage set of nodes B_{ij} pass the Δ-test for arbitrarily many epochs. Hence the Δ-test alone cannot detect all live nodes in B_i.

Now consider the graph of nodes shown in Figure 6 (b). The cycle is detected starting with the purple node f, from which cycle collection is run. If a new edge is added from node h to node g before the `MarkGray` routine is run (shown as the dashed line in the figure), the reference count of the node g will be out of date. If the cycle collector observes the newly added edge, the sum of the reference counts in $\{f, g, h\}$ will equal the sum of the edges. Hence the set of nodes $\{f, g, h\}$ will be collected by the `CollectWhite` routine and form the set B_{ik}. If the increments are not processed before the Σ-test is done, then B_{ik} will pass the Σ-test. Hence Σ-test alone cannot detect all live nodes in B_i. $\square$

Notice that we are not claiming that the two race conditions shown in Figure 6 are an exhaustive list of all possible race conditions that our algorithm will face. But these two are sufficient to show the necessity of both the tests. Thus the two tests are both necessary and sufficient to ensure the safety of the algorithm.

Finally we prove here the following Lemma that will be used in the next section, since the proof uses the notation from the present section. We define a *complete* set of nodes as one which is closed under transitive closure in the transpose graph; that is, a complete set of nodes includes all of its parents.

Lemma 2. *If $C \subseteq \Gamma_i$ is a complete set of nodes, then C will pass both the Σ-test and the Δ-test.*

Proof. By the stability property of garbage, there can be no changes to the reference counts of the nodes $S \in C$, since $S \in \Gamma_i$. Therefore, C passes the Δ-test.

By the same reasoning,

$$RC(S, C) = RC^{\Sigma}(S, C).$$

Since C is a complete set,

$$RC(S) = RC(S, C).$$

Therefore,

$$\begin{aligned} CRC(S) &= RC(S) - RC^{\Sigma}(S, C) \\ &= RC(S) - RC(S, C) \\ &= 0 \end{aligned}$$

Hence, C will pass the Σ-test.   □

5.3   Liveness: Proof of Correctness

A garbage collector is *live* if it eventually collects all unreachable objects. Our concurrent algorithm is subject to some extremely rare race conditions, which may prevent the collection of some garbage. Therefore, we prove a weak liveness condition which holds provided that the race condition does not occur in every epoch.

We can only demonstrate weak liveness because it is possible that a candidate cycle $C \in \Omega_i$ contains a subset which is a complete garbage cycle, and some nodes in that subset point to other nodes in C which are live (see Figure 7). This is a result of our running a variant of the synchronous cycle collection algorithm while mutation continues, thus allowing race conditions such as the ones shown in Figure 6 to cause Collect-Cycles to occasionally place live nodes in a candidate set B_{ik}. The resulting candidate set B_{ik} will fail either the Σ-test or the Δ-test for that epoch. If this occurs, the cycle will be reconsidered in the following epoch. Therefore, unless the race condition occurs indefinitely, the garbage will eventually be collected.

Any garbage nodes that are not collected in epoch i are called the *undiscovered garbage* of epoch i.

In practice, we have been unable to induce the race condition that leads to undiscovered garbage, even with adversary programs. However, this remains a theoretical limitation of our approach.

We have solutions to this problem (for instance, breaking up a set that fails either of the two tests into strongly connected components), but have not included them because they complicate the algorithm and are not required in practice. We are also investigating another alternative, in which the entire cycle collection is based on performing a strongly-connected component algorithm [4]; this alternative is also promising in that the number of passes over the object graph is substantially reduced.
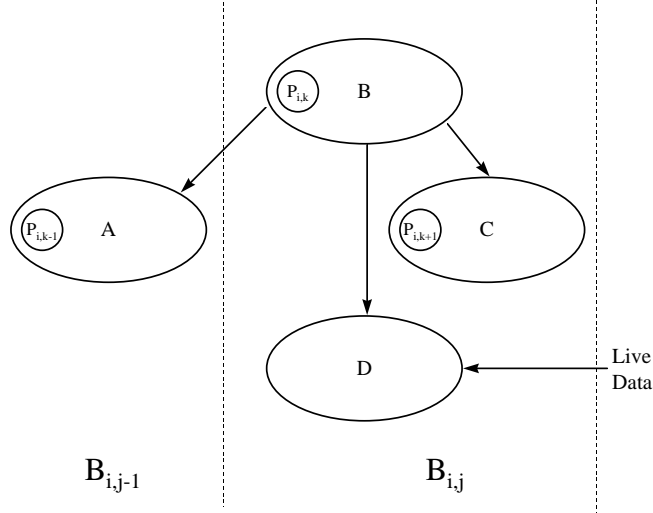
Fig. 7. Concurrent mutation of the nodes in D can cause candidate sets $B_{i,j-1}$ and $B_{i,j}$ to become undiscovered garbage.

In this section we will prove that if a set of garbage nodes is free from the race condition leading to undiscovered garbage, it will be collected in the epoch i; otherwise it will be considered again in the epoch $i + 1$.

We previously defined P_i as the set of purple nodes in epoch i, that is the set of nodes from which the cycle detection algorithm begins searching for cyclic garbage in `CollectCycles`.

Theorem 3. *The purple set is maintained correctly by the concurrent cycle collection algorithm: every garbage node is reachable from some purple node. That is,*

$$\Gamma_i \subseteq P_i^\star.$$

Proof. The `Decrement` procedure ensures that the only garbage in the set G_i is cyclic garbage. In addition, `Decrement` adds all nodes having decrement to non-zero to the purple set. Thus we know that any cyclic garbage generated during the processing of increments and decrements in epoch i is reachable from the purple set. What remains to be proved is that the purple set contains roots to any uncollected cyclic garbage from the previous epochs.

We know this to be trivially true for epoch 1. We assume that it is true for epoch i and prove that, after running the `CollectCycles` routine, it is true for epoch $i + 1$. The result then follows by induction.

Let P be a member of the purple set in epoch i that is the root of a garbage cycle. The `CollectRoots` routine ensures that the root of each cycle generated by it is stored in the first position in the buffer and takes all white nodes that are reachable from it and unless it has been collected before (therefore reachable from another root node) puts it in the current cycle. Since `CollectCycles` is a version of the synchronous

garbage collection algorithm, and there can be no concurrent mutation to a subgraph of garbage nodes, all such garbage nodes will be in the current cycle. In addition, any other uncollected purple node reachable from P and the cycle associated with will be added to the current cycle. If this latter purple node is garbage, then it will continue to be reachable from P and hence proper handling of P will ensure proper handling of this node and its children.

The `Refurbish` routine will put this first node back into the purple root set unless: a) the current cycle is determined to be garbage, in which case the entire cycle is freed or b) the first node is determined to be live, in which case it is not the root of a garbage cycle.

Hence the purple set P_{i+1} will contain the roots of all the garbage cycles that survive the cycle collect in epoch i.   □

Corollary 1. *The cycle buffer generated in the ith epoch contains all the garbage nodes present in G_i. That is,*

$$\Gamma_i \subseteq B_i \subseteq P_i^\star.$$

Proof. By Theorem 3, the root of every garbage cycle is contained in P_i. The procedure `CollectCycles` is a version of the synchronous garbage collection algorithm. There can be no concurrent mutation to a subgraph of garbage nodes. Therefore, all garbage nodes will be in put into the cycle buffer B_i.   □

Unfortunately, due to concurrent mutation B_i may contain some live nodes too. Let $B_{ij} \in B_i$ denote a set of nodes that were collected starting from a root node $P_{ik} \in P_i$. If all the nodes in B_{ij} are live, then it will fail one of the two tests (Δ-test or Σ-test) and its root will be identified as a live node and discarded.

It is however possible that B_{ij} contains a set of garbage nodes as well as a set of live nodes as shown in Figure 7. There are three purple nodes $P_{i,k-1}, P_{i,k}, P_{i,k+1}$. `CollectWhite` processes $P_{i,k-1}$ first and creates the candidate set $B_{i,j-1}$ which contains the nodes in A. Then `CollectWhite` processes $P_{i,k}$ and creates the candidate set $B_{i,j}$ which contains the nodes in B, C, and D. This includes both the garbage nodes in C reachable from another purple node not yet considered ($P_{i,k+1}$) and live nodes in D.

In this case B_{ij} will fail one of the two safety tests and the algorithm will fail to detect that it contained some garbage nodes. Furthermore, the algorithm will fail to detect any other garbage nodes that are pointed to by this garbage, such as $B_{i,j-1}$ in the figure. The roots $P_{i,k-1}$ and P_{ik} will be put into P_{i+1}, so the garbage cycles will be considered again in $(i+1)^{st}$ epoch, and unless a malicious mutator is able to fool `CollectCycles` again, it will be collected in that epoch. But the fact remains that it will not be collected during the current epoch.

Let U_i be the set of nodes undiscovered garbage in epoch i. That is, every member of U_i either has a live node in its cycle set or its cycle set is pointed to by a member of U_i. We will show that all the other garbage nodes (i.e. the set $\Gamma_i - U_i$) will be collected in epoch i.

Lemma 3. *There are no edges from garbage nodes in B_{ik} to B_{il} where $k < l$.*

Proof. The `CollectRoots` routine takes all white nodes that are reachable from the root of the current cycle, colors them orange, and places them in a set B_{il}. Any nodes that it reaches that were previously colored orange are not included in the set because they have already been included in some previous set B_{ik}. Thus all nodes that are reachable from the current root exist in the current cycle or in a cycle collected previously. Since the nodes in B_{ik} were collected before the nodes in B_{il} there can be no forward pointers from the first to the second set, unless some edges were added after the running of the `CollectRoots` routine. However, since there can be no mutation involving garbage nodes, this is not possible. □

Let K_i be the set of all nodes collected by the procedure `FreeCycles`.

Theorem 4. *All the garbage that is not undiscovered due to race conditions will be collected in the ith epoch. That is,*

$$K_i = \Gamma_i - U_i.$$

Proof. From Corollary 1 above, we know that every node in Γ_i is contained in B_i.

If a cycle B_{ij} fails the Δ-test, then we know that it has a live node. In that case, a node in this cycle is either live, in which case it does not belong to Γ_i, or it is a garbage node that is undiscovered garbage, hence it belongs to U_i. Thus none of these nodes belong to $\Gamma_i - U_i$.

If a cycle B_{ij} fails the Σ-test, it means that there is some undeleted edge from outside the set of nodes in B_{ij}. By Lemma 3, it cannot be from a garbage node that comes earlier in the cycle buffer. If this is from a live node or from a garbage node that is undiscovered garbage, then garbage nodes in this cycle, if any, belong to the set U_i. If it is not from an undiscovered garbage node, then that garbage node belongs to a discovered garbage set later in the cycle buffer.

But in the `FreeCycles` routine we process the cycles in the reverse of the order in which they were collected. As we free garbage cycles, we delete all edges from the nodes in it. By Lemma 3, the last discovered garbage set cannot have any external pointers to it. Therefore it will pass the Σ-test also. In addition, when we delete all edges from this set, the next discovered garbage set will pass the Σ-test. Hence, every discovered garbage cycle set B_{ij} will pass both the tests. □

Corollary 2. *In the absence of the race condition leading to undiscovered garbage, namely a mixture of live and garbage nodes in some set B_{ik}, all garbage will be collected. That is,*

$$K_i = \Gamma_i.$$

Proof. In this case, there are no live nodes in any B_{ik} that contains garbage and hence U_i is a null set. The result follows from Theorem 4. □

6   Measurements

We now present measurements of the effectiveness of our concurrent cycle collection algorithm within the Recycler, a reference counting garbage collector implemented as

Program	Description	Applic. Size	Threads	Objects Allocated	Percent Acyclic
201_compress	Compression	18 KB	1	0.2 M	73%
202_jess	Java expert system	11 KB	1	17.4 M	19%
209_db	Database	10 KB	1	6.6 M	10%
227_mtrt	Multithreaded raytracer	571 KB	2	14.2 M	87%
228_jack	Parser generator	131 KB	1	16.8 M	78%
portbob	Business Object Benchmark	138 KB	3	7.9 M	61%
jalapeño	Jalapeño compiler	1378 KB	1	19.2 M	7%
ggauss	Cyclic torture test (synth.)	8 KB	1	32.5 M	< 1%

Table 2. Benchmarks and their overall characteristics.

part of, Jalapeño, a Java virtual machine written in Java at the IBM T.J. Watson Research Center.

The measurements in this section concentrate on the operation of the reference counting system within the Recycler. In concurrently published work [3] we present detailed measurements of the system as a whole, including a comprehensive performance evaluation which shows that with sufficient resources, the Recycler achieves a maximum 6 millisecond pause time without appreciably slowing down the applications.

6.1   Benchmarks

Table 2 summarizes the benchmarks we used. Our benchmarks consist of a mixture of SPEC benchmarks and other programs: `portbob` is an early version of the benchmark recently accepted by SPEC under the name `jbb`; `jalapeño` is the Jalapeño optimizing compiler compiling itself; and `ggauss` is a synthetic benchmark designed as a "torture test" for the cycle collector: it does nothing but create cyclic garbage, using a Gaussian distribution of neighbors to create a smooth distribution of random graphs.

Since we did no have source code for all benchmarks, application size is given as the total class file size in kilobytes.

SPEC benchmarks were run with "size 100" for exactly two iterations, and the entire run, including JIT time, was counted.

We ran the benchmarks with one more CPU than there are threads; the extra CPU ran the concurrent collector.

The largest benchmark is `jalapeño`, the Jalapeño optimizing compiler compiling itself. It allocates 19 million objects, of which only 8% are determined by the classloader to be acyclic (and therefore marked green). The optimizer represents the worst-case type of program likely to be seen by the cycle collector in practice: its data structures consist almost entirely of graphs and doubly-linked lists.

6.2   Cycle Collection

Table 3 summarizes the operation of the concurrent cycle collection algorithm. Cycle collection was performed every eight epochs or when the `Roots` buffer exceeded a

Program	Ep.	Cyc. Coll.	Roots Checked	Cycles Found			Marked			Refs. Traced	Trace/ Alloc.
				Coll.	Σ	Δ	Gray	White	Orange		
201_compress	40	9	12153	97	0	0	13020	726	484	62971	0.42
202_jess	127	33	155507	0	13	0	293995	14523	52	2281521	0.13
209_db	275	61	1261177	0	0	0	2793425	14551	0	23737713	3.57
227_mtrt	152	29	467334	10	0	2	3097618	1122418	654137	18558847	1.31
228_jack	230	47	151160	782	0	0	231356	75610	49141	875234	0.05
portbob	50	16	434071	0	0	5	678077	2279	5	6345488	0.80
jalapeño	476	106	6382521	279790	0	0	7621940	3049754	2019731	49571627	2.58
ggauss	489	105	7111449	266666	0	0	7511620	6782726	6484782	37715868	1.16

Table 3. Cycle Collection. "Ep." is the number of epochs; "Cyc. Coll." is the number of cycle collections. For "Cycles Found", the number collected, and rejected due to the Σ- and Δ-tests.

threshold size; usually the latter condition triggered cycle collections sooner. They seem to occur once every four to five epochs.

There were a number of surprising results. First of all, despite the large number of roots considered, the number of garbage cycles found was usually quite low. Cyclic garbage was significant in jalapeño and our torture test, ggauss. It was also significant in compress, although the numbers do not show it: multi-megabyte buffers hang from cyclic data structures in compress, so the application runs out of memory if its 97 cycles are not collected in a timely manner.

The Jalapeño optimizing compiler and the synthetic graph generator both freed about 20% of their objects with the cycle collector. We were surprised that the number was not higher, given that virtually all of the data structures were potentially cyclic. However, it appears that even so, a large proportion of objects are simple (rather than cyclic) garbage.

Of more than half a million candidate cycles found for the eight benchmarks, concurrent mutation introduced only 20 false candidate cycles, with most of the false cycles being rejected by the Σ-test in jess. None of these rejected cycles was undiscovered garbage that was collected later (that is, part of a set U_i as described in Section 5).

In fact, we were unable to create false cycles artificially when we tried modifying the ggauss program to turn it into a malicious mutator designed solely for the purpose of fooling the CollectCycles algorithm. This demonstrates conclusively that undiscovered garbage is a problem only in theory, and not in practice.

Table 3 also shows how the different phases of the cycle collection algorithm proceeded: marking (gray), looking like cyclic garbage (white), and provisionally identified as cyclic garbage (orange). The amount of marking varied widely according to the benchmarks.

Finally, Table 3 shows the number of references that must be followed by the concurrent reference counting collector ("Refs. Traced"). We have also normalized this against the total number of objects allocated ("Trace/Alloc"). The db benchmark required the most tracing per object. Apparently it performs far more modification of its potentially cyclic data structures than other programs – presumably inserting and removing database objects into its index data structure.

Compared to tracing garbage collectors, the reference counting collector has an advantage in that it only traces locally from potential roots, but has a disadvantage in that the algorithm requires multiple passes over the subgraph. Furthermore, if the root of a large data structure is entered into the root buffer frequently and high mutation rates force frequent epoch boundaries, the same live data structure might be traversed multiple times.

Our measurements show that a reference-counting based collector using our cycle collection technique may perform very little tracing, or a large amount of tracing, and that this is very application dependent.

Over all, the measurements presented here show that our cycle collection algorithm is practical and capable of handling large programs, and in many cases should provide significantly increased locality and reduction in memory traffic over tracing-based collectors.

The Recycler is described in greater detail and compared quantitatively to a parallel mark-and-sweep collector by Bacon et al [3].

7   Related Work on Concurrent Collection

While numerous concurrent, multiprocessor collectors for general-purpose programming languages have been described in the literature [8, 10, 11, 14, 15, 18, 19, 21, 26, 27], the number that have been implemented is quite small and of these, only a few actually run on a multiprocessor [2, 8, 14, 11, 13, 24].

DeTreville's work on garbage collectors for Modula-2+ on the DEC Firefly workstation [8] is the only comparative evaluation of multiprocessor garbage collection techniques. His algorithm is based on Rovner's reference counting collector [26] backed by a concurrent tracing collector for cyclic garbage. Unfortunately, despite having implemented a great variety of collectors, he only provides a qualitative comparison. Nevertheless, our findings agree with DeTreville's in that he found reference counting to be highly effective for a general-purpose programming language on a multiprocessor. The Recycler differs in its use of cycle collection instead of a backup mark-and-sweep collector.

Huelsbergen and Winterbottom [15] describe a concurrent algorithm (VCGC) that is used in the Inferno system to back up a reference counting collector. They report that reference counting collects 98% of data; our measurements for Java show that the proportion of cyclic garbage is often small but varies greatly. The only measurements provided for VCGC were on a uniprocessor for SML/NJ, so it is difficult to make meaningful comparisons.

The only other concurrent, multiprocessor collector for Java that we know of is the work of Domani et al [13, 12]. This is a generational collector based on the work of Doligez et al [11], for which generations were shown to sometimes provide significant improvements in throughput.

The other implemented concurrent multiprocessor collectors [2, 14, 11, 24] are all tracing-based algorithms for concurrent variants of ML, and generally have significantly longer maximum pause times than our collector. In addition, ML produces large amounts of immutable data, thereby simplifying the collection process.

The garbage collector of Huelsbergen and Larus [14] for ML achieved maximum pause times of 20 ms in 1993, but only for two small benchmarks (Quicksort and Knuth-Bendix). Their collector requires a read barrier for mutable objects that relies on processor consistency to avoid locking objects while they are being forwarded. Read barriers, even without synchronization instructions, are generally considered impractical for imperative languages [17], and on weakly ordered multiprocessors their barrier would require synchronization on every access to a mutable object, so it is not clear that the algorithm is practical either for imperative languages or for the current generation of multiprocessor machines.

Lins has presented a concurrent cycle collection algorithm [21] based on his synchronous algorithm. Unlike the Recycler, Lins does not use a separate reference count for the cycle collector; instead he relies on processor-supported asymmetric locking primitives to prevent concurrent mutation to the graph. His scheme has, to our knowledge, never been implemented. It does not appear to be practical on stock multiprocessor hardware because of the fine-grained locking required between the mutators and the collector. Our algorithm avoids such fine-grained locking by using a second reference count field when searching for cycles, and performing safety tests (the Σ-test and the Δ-test) to validate the cycles found.

Jones and Lins [16] present an algorithm for garbage collection in distributed systems that uses a variant of the lazy mark-scan algorithm for handling cycles. However, they rely on much more heavy-weight synchronization (associated with message sends and receives and global termination detection) than our algorithm. The algorithm has never been implemented.

In terms of cycle collection systems that have been implemented, the closest to our work is that of Rodrigues and Jones [25], who have implemented an algorithm for cycle collection in distributed systems. However, they use a tracing collector for local cycles and assume that inter-processor cycles are rare, and they use considerably more heavy-weight mechanisms (such as lists of back-pointers) than we do; on the other hand they also solve some problems that we do not address, like fault tolerance.

8   Conclusions

We have presented algorithms for the collection of cyclic data structures in reference counted systems, starting with a synchronous algorithm which we then extended to handle concurrent mutation without requiring any but the loosest synchronization between mutator threads and the collector.

We presented detailed pseudocode and a proof of correctness of the concurrent algorithm. We have implemented these algorithms as part of the Recycler, a concurrent multiprocessor reference counting garbage collector for Java, and we presented measurements that show the effectiveness of our algorithm over a suite of eight significant Java benchmarks.

Our work is novel in two important respects: it represents the first practical use of cycle collection in a reference counting garbage collector for a mainstream programming language; and it requires no explicit synchronization between the mutator threads or between the mutators and the collector.

Another contribution of our work is our proof methodology, which allows us to reason about an abstract graph that never exists in the machine, but is implied by the stream of increment and decrement operations processed by the collector. In effect we are able to reason about a consistent snapshot without ever having to take such a snapshot in the implementation.

Our cycle collection algorithm forms a key part of the Recycler, a garbage collector for Java, which achieves end-to-end execution times competitive with a parallel mark-and-sweep collector while holding maximum application pause times to only six milliseconds.

Acknowledgements

References

[1] ALPERN, B., ET AL. Implementing Jalapeño in Java. In *OOPSLA'99 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications* (Denver, Colorado, Oct. 1999). *SIGPLAN Notices, 34*, 10, 314–324.

[2] APPEL, A. W., ELLIS, J. R., AND LI, K. Real-time concurrent collection on stock multiprocessors. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation* (Atlanta, Georgia, June 1988). *SIGPLAN Notices, 23*, 7 (July), 11–20.

[3] BACON, D. F., ATTANASIO, C. R., LEE, H. B., RAJAN, V. T., AND SMITH, S. Java without the coffee breaks: A nonintrusive multiprocessor garbage collector. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation* (Snowbird, Utah, June 2001). *SIGPLAN Notices, 36*, 5 (May).

[4] BACON, D. F., KOLODNER, H., NATHANIEL, R., PETRANK, E., AND RAJAN, V. T. Strongly-connected component algorithms for concurrent cycle collection. Tech. rep., IBM T.J. Watson Research Center and IBM Haifa Scientific Center, Apr. 2001.

[5] BOBROW, D. G. Managing re-entrant structures using reference counts. *ACM Trans. Program. Lang. Syst. 2*, 3 (July 1980), 269–273.

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

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

[8] DETREVILLE, J. Experience with concurrent garbage collectors for Modula-2+. Tech. Rep. 64, DEC Systems Research Center, Palo Alto, California, Aug. 1990.

[9] DEUTSCH, L. P., AND BOBROW, D. G. An efficient incremental automatic garbage collector. *Commun. ACM 19*, 7 (July 1976), 522–526.

[10] DIJKSTRA, E. W., LAMPORT, L., MARTIN, A. J., SCHOLTEN, C. S., AND STEFFENS, E. F. M. On-the-fly garbage collection: An exercise in cooperation. In *Hierarchies and Interfaces*, F. L. Bauer et al., Eds., vol. 46 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, 1976, pp. 43–56.

[11] DOLIGEZ, D., AND LEROY, X. A concurrent generational garbage collector for a multi-threaded implementation of ML. In *Conference Record of the Twentieth ACM Symposium on Principles of Programming Languages* (Charleston, South Carolina, Jan. 1993), ACM Press, New York, New York, pp. 113–123.

[12] DOMANI, T., ET AL. Implementing an on-the-fly garbage collector for Java. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management* (Minneapolis, MN, Oct. 2000). *SIGPLAN Notices*, *36*, 1 (Jan., 2001), 155–166.

[13] DOMANI, T., KOLODNER, E. K., AND PETRANK, E. A generational on-the-fly garbage collector for Java. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation* (June 2000). *SIGPLAN Notices*, *35*, 6, 274–284.

[14] HUELSBERGEN, L., AND LARUS, J. R. A concurrent copying garbage collector for languages that distinguish (im)mutable data. In *Proceedings of the Fourth ACM Symposium on Principles and Practice of Parallel Programming* (May 1993). *SIGPLAN Notices*, *28*, 7 (July), 73–82.

[15] HUELSBERGEN, L., AND WINTERBOTTOM, P. Very concurrent mark-&-sweep garbage collection without fine-grain synchronization. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management* (Mar. 1999). *SIGPLAN Notices*, *34*, 3, 166–174.

[16] JONES, R. E., AND LINS, R. D. Cyclic weighted reference counting without delay. In *PARLE'93 Parallel Architectures and Languages Europe* (June 1993), A. Bode, M. Reeve, and G. Wolf, Eds., vol. 694 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 712–715.

[17] JONES, R. E., AND LINS, R. D. *Garbage Collection*. John Wiley and Sons, 1996.

[18] KUNG, H. T., AND SONG, S. W. An efficient parallel garbage collection system and its correctness proof. In *IEEE Symposium on Foundations of Computer Science* (1977), IEEE Press, New York, New York, pp. 120–131.

[19] LAMPORT, L. Garbage collection with multiple processes: an exercise in parallelism. In *Proceedings of the 1976 International Conference on Parallel Processing* (1976), pp. 50–54.

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

[21] LINS, R. D. A multi-processor shared memory architecture for parallel cyclic reference counting. *Microprocessing and Microprogramming 35*, 1–5 (Sept. 1992), 563–568. *Proceedings of the 18th EUROMICRO Conference* (Paris, France).

[22] MARTÍNEZ, A. D., WACHENCHAUZER, R., AND LINS, R. D. Cyclic reference counting with local mark-scan. *Inf. Process. Lett. 34*, 1 (1990), 31–35.

[23] MCCARTHY, J. Recursive functions of symbolic expressions and their computation by machine. *Commun. ACM 3* (1960), 184–195.

[24] NETTLES, S., AND O'TOOLE, J. Real-time garbage collection. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation* (Albuquerque, New Mexico, June 1993). *SIGPLAN Notices*, *28*, 6, 217–226.

[25] RODRIGUES, H. C. C. D., AND JONES, R. E. Cyclic distributed garbage collection with group merger. In *Proceedings of the Twelfth European Conference on Object-Oriented Programming* (Brussels, July 1998), E. Jul, Ed., vol. 1445 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 249–273.

[26] ROVNER, P. On adding garbage collection and runtime types to a strongly-typed, statically-checked, concurrent language. Tech. Rep. CSL–84–7, Xerox Palo Alto Research Center, July 1985.

[27] STEELE, G. L. Multiprocessing compactifying garbage collection. *Commun. ACM 18*, 9 (Sept. 1975), 495–508.