Efficient On-the-Fly Cycle Collection

Harel Paz* David F. Bacon† Elliot K. Kolodner‡ Erez Petrank§ V. T. Rajan¶

ABSTRACT

A reference counting garbage collector cannot reclaim unreachable cyclic structures of objects. Therefore, reference counting collectors either use a backup tracing collector seldom, or employ a cycle collector to reclaim cyclic structures. Recently, the first on-the-fly cycle collector, that may run concurrently with program threads, was presented by Bacon and Rajan [3]. This demonstrated the ability to run on-the-fly reference counting without resorting to an auxiliary tracing collector.

In this paper, we present an improved on-the-fly cycle collector by combining techniques developed in that paper with the sliding views collectors. The resulting collector gains two advantages. First, it improves over the efficiency of the original collector significantly, making the cycle collection solution usable in practice. Second, it eliminates the termination problem that appeared in the original algorithm. There, a rare race may delay the reclamation of an unreachable cyclic structure forever. The new cycle collector guarantees reclamation of all unreachable cyclic structures.

Categories and Subject Descriptors: D.3.4 [Programming Languages]: Processors- Memory management (garbage collection).
General Terms: Languages, Algorithms.
Keywords: Runtime systems, Memory management, Garbage collection, Concurrent garbage collection, On-the-fly garbage collection.

1. INTRODUCTION

Reference counting is a classical garbage collection algorithm. Systems using reference counting were implemented starting from the sixties ([12]). However, reference counting garbage collectors cannot reclaim cyclic structures of objects (as first noticed by McBeth [29]). Thus, reference counting collectors must be either accompanied by a backup mark and sweep collector (run infrequently to collect garbage cyclic structures) or by a cycle collector.

Trying to avoid developing and maintaining an additional mark and sweep collector on the reference counting collected system, several attempts were made to design a cycle collector [9, 11]. The most popular algorithm for cycle collection was proposed by Martinez et al. [28]. Their algorithm performs a local mark and scan on objects suspected to belong to a garbage cycle, and therefore avoids global tracing. This algorithm was later enhanced several times, finally, being modified to fit an on-the-fly reference counting collector by Bacon et al. [4, 3]. An on-the-fly reference counting collector and an accompanying cycle collector are collectors that may reclaim unreachable objects concurrently with program run.

1.1 On-the-Fly Garbage Collection

Many garbage collectors were designed to work on a single thread while program threads are stopped, the so-called *stop the world* setting. On multiprocessor platforms, it is not desirable to stop the program and perform the collection in a single thread on one processor, as this leads both to long pause times and poor processor utilization. A concurrent collector runs concurrently with the program threads. The program threads may be stopped for a short time to initiate and/or finish the collection. An *on-the-fly* collector does not need to stop the program threads simultaneously, not even for the initialization or the completion of the collection cycle.

The study of on-the-fly garbage collectors was initiated by Steele and Dijkstra, et al. [35, 36, 14] and continued in a series of papers [20, 6, 7, 23, 24, 16, 15, 26, 17, 18, 3]. The advantage of an on-the-fly collector over a parallel collector and other types of concurrent collectors [5, 19, 30, 33, 10], is that it avoids the operation of stopping all the program threads. Such an operation can be costly, and it usually increases the pause times. Today, on-the-fly collectors typically achieve pauses as short as a couple of milliseconds.

1.2 The challenge

Bacon and Rajan's first on-the-fly cycle collector has two main drawbacks: a practical problem and a theoretical one. A typical cycle collector traces cycle candidates repeatedly to discover which cycles are only referenced by pointers from within the cycle. A crucial problem with repeated scanning arises when the program threads are allowed to modify the objects graph during the scan. This means that a scan cannot really repeat: the objects graph in one scan is not equal to the objects graph in the next scan. Furthermore, as modifications occur concurrently with the scan, each one of these scans cannot be guaranteed to view a consistent snapshot of the

*Dept. of Computer Science, Technion - Israel Institute of Technology, Haifa 32000, Israel. Email: pharel@cs.technion.ac.il.
†IBM Research.
‡IBM Research.
§Dept. of Computer Science, Technion - Israel Institute of Technology, Haifa 32000, Israel. Email: erez@cs.technion.ac.il. Research supported by the Bar-Nir Bergreen Software Technology Center of Excellence and by the IBM Faculty Partnership Award.
¶IBM Research.

objects graph at any specific point in time. This problem is the source of the two drawbacks of the previous algorithm: a practical and a theoretical drawback.

The practical problem is that in order to achieve safety, the algorithm in [4] makes many repeated scans over the candidates. This reduces the overall efficiency of the reference counting collector. The theoretical problem is that liveness cannot be guaranteed. A rare race condition may prevent an unreachable cyclic structure from being ever reclaimed.

Note, that *liveness* of the algorithm here is used in the standard sense in distributed computing. It should not be confused with liveness of objects in the heap.

1.3 The solution

In this work, we propose an algorithm for on-the-fly cycle collection which solves these drawbacks. Our solution employs several new techniques recently developed for concurrent garbage collection in [3, 26, 4, 2]. The main idea is to virtually fix the graph processed by the cycle collector. Suppose first that we stopped the threads and took a replica of the heap snapshot. Running the synchronous (and efficient) algorithm of [4] on this snapshot efficiently detects any cyclic structure. Of-course, taking a replica of the heap is not realistic. However, a virtual snapshot of the heap may be taken using the ideas in [26]. Furthermore, if we use a sliding view instead of a snapshot (as in [26]) and make the appropriate adjustment to use a sliding view to scan the objects graph (as in [2]), then we obtain an on-the-fly cycle collector with the same short pauses of recent on-the-fly collectors ([3, 26, 2]). In this paper, we also suggest further improvements on the synchronous algorithm in [4] making it run even faster.

The theoretical liveness problem is immediately solved. If an unreachable cyclic structure is generated by the program before the snapshot, or before the start of the interval in which the sliding view is read, then the garbage cycle may be easily identified on this view. When a cycle collection is executed on top of this sliding view, this cycle is guaranteed to be reclaimed.

We further incorporate the new algorithm into the more efficient age-oriented collector of Paz and Petrank [32]. The two algorithms seem a perfect match. Our cycle collector spends a large fraction of its time working on cycle candidates among newly allocated objects. The age-oriented collector, building on the weak generational hypothesis uses tracing to reclaim all newly allocated objects and reference counting for the rest of the objects. This eliminates a large fraction of the cycles as well as a large fraction of the cycle collector's work, when run only on older objects.

1.4 Implementation, measurements, discussion

We have implemented the new cycle collector as an add-on to the Levanoni-Petrank reference counting collector [26] and to the more efficient age-oriented collector of [32]. The implementation was done on the Jikes Java virtual machine [1], where the original cycle collector of Bacon and Rajan [4] is also implemented. To measure the effectiveness of the cycle collector, we compared some of its featured with the original cycle collector, showing that the number of traced objects is substantially reduced. Comparing the throughput is irrelevant in this case, since they are built on two different reference counting collectors (see [3, 26]).

We provide the first comparison of cycle collection to a backup tracing collector. This comparison is a most interesting one, since these are the main two options provided to an implementer of a reference counting algorithm. Our measurements contain a comparison of the throughput of a JVM that uses the cycle collector with a JVM that uses a backup tracing collector to collect unreachable cyclic structures. We used the SPECjbb2000 benchmark and the SPECjvm98 benchmark suites. These benchmarks are described in detail in SPEC's Web site [34].

It turns out that cycle collection still falls behind a backup tracing collector by 5-10% (application overall throughput). This small cost allows using pure reference counting without adding a backup tracing collector. However, with the direction modern computing is taking, we believe that the cycle collector may become much more effective comparing to the a backup tracing collector. Reference counting is currently inferior to tracing with respect to throughput on modern benchmarks [2]. However, as heaps grow larger, it is possible that reference counting will start winning. While tracing must trace the live objects in the heap, reference counting needs only account for reference counts updates and reclaiming dead objects. Actually, when the heap is tight and collections are frequent, reference counting is already winning over tracing the whole heap [2]. If future benchmarks use a large live heap with few updates, mostly to few young objects, then reference counting may become the best collector. When that happens, a companion cycle collector will be required. In that case, our cycle collector may be an excellent companion and we expect it to outperform a backup tracing collector.

Interestingly, when the cycle collector was used with the more efficient age-oriented collector, it performed as well as the backup tracing collector. It turns out that if the work on young objects is spared of the cycle collector, then it becomes highly efficient. Detailed measurements are provided in Section 5.

In the sequel, we assume that the reader is familiar with memory management standard terminology and algorithms. For a more detailed introduction to garbage collection and memory management, the reader is referred to [22].

1.5 Contribution of this work

We improve over the previous on-the-fly cycle collection algorithm of Bacon and Rajan by significantly improving the efficiency of the collector and by solving its theoretical liveness problem. Our contributions are:

- Combining the sliding-views technique of Levanoni and Petrank with Bacon and Rajan's cycle collector. This combination enables employing the simpler synchronous cycle collection algorithm on the obtained sliding view. It hence results in less tracing (and less buffering) and solves the liveness theoretical problem.

- One of the main technical challanges addressed in this paper is the execution of cycle collection in spite of having only partial information on reference counting decrements. The sliding views algorithm does not keep track of all decrements. This full list of decrements has previously been the list of candidates for cycle collection. We are able to use sliding views with their fast write barrier and still be able to collect all cycles. This is a second source of efficiency improvement. All previous cycle collectors had to explore all objects whose reference-count were decremented to a non-zero value. We do not need to go over all these objects.

- We propose a better scheduling strategy for cycle collection which allows reducing the number of candidates even further. Our scheduling only traverses candidates that have been in the system long enough and did not show any "life signs" recently.

- We develop new filtering techniques to further reduce the number of traced objects.

- We provide the first comparison between running a backup tracing collector and running a cycle collection with reference counting.

- We provide an intergation with an age-oriented collector that does not use reference counting on the young generation. It turns out that this greatly reduces the load on the cycle collector.

1.6 Organization

We start with an overview of the previous collectors in Section 2. The new cycle collector employs techniques from collectors described in this section. An overview of the new cycle collector is provided in Section 3. Implementation and results are given in Sections 4 and 5. Related work is discussed in Section 6 and we conclude in Section 7. The details of the cycle collector including the pseudo-code is given in the Appendix A.

2. REVIEW OF PREVIOUS COLLECTORS

In this section, we review relevant previous work. The algorithmic ideas presented in the section are then used to describe the new collector in Section 3 below. We start by reviewing the (synchronous) algorithms for cycle collection on a uniprocessor [28, 27, 4], we then review the previous concurrent cycle collector [4]. Finally, we explain ideas from the sliding views reference counting and tracing garbage collectors [26, 2] that aid in overcoming the disadvantages in state-of-the-art cycle collectors.

2.1 Collecting cycles on a uniprocessor

The algorithm of [28, 27] is based on two observations. The first observation is that garbage cycles can only be created when a reference count is decremented to a non-zero value. The second observation is that in a garbage cycle, all the reference counts are internal, i.e., the only pointers that reference the cycle's nodes reside on the cycle's nodes themselves. Thus, if internal references are subtracted from the reference counts, the reference counts of the nodes on a garbage cycle become zero and the unreachable cycle is identified.

The incorporation of the cycle collector of [28, 27] in a reference counting collector is as follows. Most unreachable objects are reclaimed when their reference count is decremented to zero. To reclaim cycles, the reference counting algorithm detects candidate unreachable cyclic structures and runs the cycle collector on them. Specifically, whenever a reference count of an object O is decremented to a value different than zero, a cycle collector (described below) is applied on all objects reachable from O.

The general idea of the cycle collection algorithm is to perform three traversals over the graph of objects reachable from a candidate object O. These traversals update the reference counts to reflect only pointers that are external to the cycle. It then reclaims all objects whose reference count was decremented to zero and restores the reference counts of the surviving objects.

While traversing the objects, the algorithm uses three colors to mark the state of objects. The initial color of an object is black, signifying that this object is active; a possible member of a garbage cycle is represented by the gray color; an object is eventually marked white if it is found to be a member of a garbage cycle.

The algorithm, given a candidate object O whose reference count was decremented to a non-zero value, works in three stages:

- **The mark stage:** traces the graph of objects reachable from O, subtracting counts due to internal references and marking nodes as possible garbage (by coloring them gray). At the end of this (first) traversal, the reference counts in the sub-graph will only reflect external pointers to nodes in the sub-graph.

- **The scan stage:** scan the sub-graph of objects reachable from O and restores the reference counts of objects which are reachable from external pointers. All such reachable nodes are re-colored black. All other nodes in the sub-graph (objects which remain with a zero reference counts) are colored white (these objects are identified as forming a garbage cycle).

- **The collect stage:** scan the sub-graph again and reclaim all garbage (white) objects.

To increase efficiency, instead of invoking the cycle collector each time a new candidate object is detected, the algorithm is lazily applied. Hence, the above traversals are postponed by saving the values of the deleted pointers in a buffer. Each such value is a candidate to be a root of a garbage cycle. Later on, at a suitable point, the buffer's objects are traversed. The benefit obtained by this laziness, is that at the later time, when the buffer is traversed, we have further indication on whether a value stored in the buffer is garbage. Most buffer's values are irrelevant by the time the buffer is traversed because their reference count either drops to zero (their other references were meanwhile deleted) or is incremented. Moreover, lazy treatment of candidates often prevents re-traversals of the same object. To summarize, delaying the traversal decreases the number of overall candidates that are traversed.

An additional important efficiency improvement (recently proposed by [4]) is to run each of the scans on all candidates simultaneously instead of applying the algorithm on each candidate separately. That strategy achieves a practical efficiency improvement and reduces the theoretical worst case complexity from $O(n^2)$ to $O(n)$ (where n stands for the transitive closure of the candidates).

2.1.1 Cycle candidates

Previously, we have mentioned that garbage cycles can only be created when a reference count is decremented to a non-zero value, and hence whenever a reference count of an object is decremented to a value different than zero, it is considered as a candidate of belonging to a garbage cycle.

Note, that a garbage cycle could be created while an application root de-reference an object. Consider the next scenario occurring: Two new objects are created and their pointers form a cycle. Next, local pointers to these objects are erased (the application roots referencing those two new objects are modified to point to other objects). Thus, a garbage cycle was formed as these two objects have reference count one, but they are not reachable from the program roots. Note, that if one does not employ a write-barrier on the application roots, the two decrements (corresponding to local pointers to the cycle's objects being erased) are unnoticed, and hence the garbage cycle could be left undetected. One simple solution to this problem is employing a write-barrier also on the application roots. This solution indeed catched all candidates, but yields an un-efficient collector.

Hence, to solve this problem, the usual employed solution is to consider all objects which have been created since the last cycle collection (and whose reference-count is not zero) and objects referenced by the program roots during the previous cycle collection, as candidates for a garbage cycle. Doing that covers all possible cycles that could have been created by modifying a root. Note, that by the weak generational hypothesis that "most objects have short lifetimes", we do not expect to get too many false candidates to be

considered. Most objects are reclaimed by the reference counting collector and the few remaining are candidates for cycle collection.

2.2 Collecting cycles on-the-fly

A concurrent cycle collection algorithm is more complex. It should deal with the fact that the object graph may be modified while the collector is scanning it. Another problem concerns the accuracy of reference counts: they may be outdated due to mutator activity.

Bacon and Rajan designed and implemented the first concurrent cycle collection algorithm ([4]) building on and extending the synchronous algorithm described above. Since the object graph may be modified while the cycle collection algorithm scans it, they cannot rely on repeated traversal of the graph reading the same set of nodes (and edges).

Their algorithm consists of two phases. In the first phase, they use a variant of the synchronous algorithm to obtain a candidate set of nodes believed to be garbage cycles. Namely, they run a variant of the above three stages algorithm, but instead of collecting the white nodes, those nodes are added to a set of possible garbage nodes. Due to concurrent mutator activity, the algorithm may produce incorrect results. In particular, the presence of concurrent mutator activity can cause live nodes to be included in this set of candidates cycles. Thus, a second phase is required to prevent the collection of false cycles. The second phase is executed in the next garbage collection. In this phase, each candidate cycle (as detected in the previous phase) is tested against the information available at this later time to ensure that it is indeed a garbage cycle. Only then, candidate cycles are reclaimed.

In the synchronous algorithm, the mark stage subtracts reference counts of traced objects, while the scan stage increments reference counts (of objects which do not belong to garbage cycles), so that when the algorithm terminates each object holds its exact reference count. Since the concurrent collector can not rely on the repeated traversal of the graph to form the same set of nodes (and edges), then it is not possible to certify that the second traversal (scan stage) would restore the original reference count of an object correctly. To solve this issue, the algorithm does not modify the actual reference count field of an object. Instead, a second reference count field is kept for each object, denoted *CRC* (cyclic reference count), and is used by the cycle collection algorithm. The mark stage of their algorithm initializes the CRC of each black object reached (to its reference count). The following stages (the mark, scan and collect stages) operate solely upon the CRC, living the reference count field unmodified. Thus, they do not need to be restored.

2.2.1 Two disadvantages

The concurrent garbage collector presented above poses two drawbacks. One is theoretical and the other is practical. These drawbacks initiated this work, which solves them both, resulting in an efficient, non-intrusive, and live collector. We describe these two drawbacks, starting from the theoretical one.

A garbage collector is called *live* if it eventually collects all reachable objects. Rare race conditions may prevent the above cycle collector from collecting garbage cycles. An example of a garbage cycle that is never collected follows. The output of the first phase is a set of candidate cycles. Each such candidate cycle is a set of nodes that may form a garbage cycle. Due to mutator activity, a candidate cycle may include an actual garbage cycle but also some additional live nodes. Such a candidate cycle is bound to fail the tests of the second phase (at the next collection). When this happens, the algorithm reconstructs this candidate cycle using a chosen node from the cycle. However, if a similar race condition occurs again, the reconstructed set of nodes may, again, contain some live modes. Such a course of events may prevent the algorithm from collecting this garbage cycle, no matter how many collections are run. To sum up, this collector cannot guarantee collecting all unreachable objects.

We now turn to the practical problem. Since the objects graph is being modified by the program during the activity of the cycle collector, some care need be taken to ensure safety of the collection. This extra care is translated into the second phase of the algorithm and into more scanning of objects, hence causing a substantial reduction in efficiency. For example, the algorithm colors objects concurrently with the execution of the mutators. Therefore, it is possible that the mutators eliminate a reference and cause an arbitrary set of gray or white objects to be invisible to the next collector scan. These improperly colored objects may later fool a naive algorithm into making incorrect reclamations. The concurrent algorithm in [4] handles this problem by adding an action to each increment or decrement of a reference count. In particular, whenever the reference count of a gray or white object is changed, this object and all objects that are reachable from it and are gray or white, are colored black. This excessive scanning of objects pose an efficiency problem. The number of objects scanned as reported in [4] is high.

2.3 Incorporating sliding views

As explained in Section 2.2.1 above, there are two disadvantages to the concurrent cycle collector of [4]. One is the reduced efficiency due to repeated scanning of objects and the other is the inability of the collector to guarantee liveness. Both problems stem from the fact that the concurrent cycle collector cannot rely on being able to re-trace the same graph.

In this work, we present a modified cycle collection algorithm that extends the above techniques from [4] and adds recently developed techniques from [26, 2]. The idea is to use a snapshot of the heap or a sliding-view of the heap ([26]). Given a fixed view of the heap (as reflected by a snapshot or the sliding-views mechanism), it is possible to eliminate much of the redundant tracing and to guarantee liveness. Before describing the new algorithm, we provide an overview of the sliding views reference counting collector.

A simple version of the Levanoni-Petrank collector may be described when allowing a point in time in the beginning of the collection in which all mutators are halted. Using such a halt, it is possible to get a virtual snapshot of the heap using a copy-on-write mechanism. Each object is associated with a dirty bit which is cleared during the halt. Then, whenever a pointer is modified, the dirty bit of the object holding this reference is probed. If the object is dirty (i.e., has been modified previously) then the pointer assignment may proceed with no further action. Otherwise, the object is copied to a local buffer before the assignment is executed.

This allows a reference counting or a tracing collector to access a view of a heap snapshot as taken during the initial halt: objects that are not dirty may be read from the heap to find their values being equal to those existing during the initial halt and dirty objects have a replica in designated buffers with their actual values during the initial halt. To deal with multithreaded programs, a carefully designed write barrier is presented in [26] allowing the above write barrier to operate on concurrent threads without using any synchronization. It is also noted that only non-null references should be recorded as the other object fields are not read by the collector, and therefore updates to newly created objects do not need to be monitored.

The collector in [26] updates the reference counts due to the values of all modified pointers between the previous snapshot to the current one. It is observed that for each such pointer only two up-

dates are necessary, which buys a substantial reduction in the number of required updates. Details may be found in the original paper [26].

The algorithm described so far probably obtains short pause times, but in order to get even shorter pause times, the sliding view mechanism is proposed. Here, the program threads are not halted simultaneously, but one at a time. As a snapshot view cannot be assumed anymore, correctness considerations dictate a *snooping* mechanism. During the time in which the mutators are being halted one by one, the snooping mechanism operates for each modified pointer via the write barrier. For each such modified reference, the object that has acquired a new reference is logged. These logged objects are considered roots for the current collection and are not reclaimed. The view of the heap used by the collector may be thought of as a sliding view: the heap objects are viewed in this view at slightly different points in time. The snooping mechanism makes sure that no reachable object is reclaimed. More details appear in [26].

3. CYCLE COLLECTOR OVERVIEW

As mentioned above, adding the sliding views techniques from [26, 2] to the state-of-the-art cycle collector of [4] achieves a synergy solving the disadvantages of the previous cycle collector and yielding a non-intrusive, efficient cycle collector that guarantees liveness.

We start by describing the new cycle collection algorithm assuming a snapshot of the heap. The main idea is that when applying the *synchronous* algorithm of [4] on a snapshot of the heap, it correctly identifies the garbage cycles in the heap as viewed at the snapshot. Now, combining the fact that the synchronous algorithm is efficient and the fact that being a garbage cycles is a stable property, i.e., program activity cannot make an unreachable object reachable, we get an efficient identification of garbage cycles. We then suggest more improvements on the obtained algorithm to improve the cycle collector further.

The main operation of the synchronous cycle collector on the heap is traversing a sub-graph of it. Thus, we first concentrate on specifying how to traverse the heap through a snapshot. The ideas are similar to those in [2]. We need to traverse an object according to its pointer values as existed at snapshot time (and not necessarily as currently existing in the heap). To do that, we employ the write barrier from [26]. When examining an object to be scanned, the collector examines the dirty bit of the object. If the object is not dirty (no pointer in the object has been modified since the snapshot was taken), then its current state is equal to its state during the snapshot and we may trace it by reading it from the heap. Otherwise, the object has been modified since the snapshot time and it is marked dirty. In this case, we trace its snapshot values as recorded in the threads local buffers. This way, objects are traced according to their state at the snapshot time, and as a consequence multiple traces are bounded to trace the same graph each time.

In terms of liveness, this means that once a garbage cycle is created, it must exist in the next snapshot, and thus is bounded to be collected by the synchronous algorithm of [4]. In terms of efficiency, this means that we may use the efficient synchronous algorithm and get rid of inefficiencies originating from the need to insure correctness in spite of program-collector races. For example, the second phase is redundant: there is no need to save identified garbage cycles to be validated during the next garbage collection (as discussed in Section 2.2.1 above), and there is no need to go over these cycles again in the next collection. In addition, there is no need to "fix" inaccurate colors during the updates of the reference counts.

We now proceed to using sliding views instead of snapshots, thus, removing the need for a simultaneous halt of all program threads. The cycle collector remains the same, except that it (obliviously) reads a sliding view of the graph rather than a snapshot. As in the previous sliding views collectors, the sliding view may find an object unreachable because the view does not represent the heap at a consistent point in time. However, the snooping mechanism (see Section 2.3) makes sure that these objects are not reclaimed, ensuring the safety property. For the cycle collector, this means that a set of objects may be incorrectly identified as being an unreachable cyclic structure. How can this happen? Inaccuracies of reference counts due to the sliding view are discussed in [25, 26]. Intuitively, if no pointer is written to the heap during the beginning of the collection (when all mutators are halted one by one) then the sliding view represents a snapshot of the heap taken at the time the first mutator is stopped, denote this time by t_1. However, as pointers are being written in the heap, this snapshot gets distorted. Such a distortion appears only with respect to modified pointers that may replace the pointers existing at time t_1. If such a modified pointer creates a falsely determined unreachable garbage cycle, then in particular, it means that a pointer implying the reachability of one of the objects in the cycle is missing from the sliding view of the heap. In this case, it is guaranteed that the object that falsely seems unreachable in the sliding view must be snooped and therefore, we will not reclaim the cyclic structure that contains it. Thus, the safety of the cycle collector may be reduced to the safety of the tracing collector in [2].

With respect to liveness, it holds that any unreachable cyclic structure that is formed before the collection begins, must be collected. The reason is that these objects are not modified during the time the sliding view is taken and in particular, no new pointers are being written to objects in this cycle. Thus, none of the objects in the cyclic structure is snooped and the view of all pointers into and in between these objects appears in the sliding view exactly as it would have appeared had we taken a real snapshot at time t_1. Thus, such an unreachable cyclic structure must be reclaimed.

3.1 Cycle candidates

Previous cycle collection algorithms use as a candidate member of an unreachable cycle, each object whose reference count has been decremented to a non-zero value. This is not possible with the Levanoni-Petrank write-barrier that eliminates many redundant reference count updates (see Section 2.3 above). When a pointer p takes the values $o_0, o_1, o_2, \ldots, o_n$ between two sliding views, the only required reference count updates are a decrement to $rc(o_0)$ and an increment to $rc(o_n)$. However, the fact that not all increments and decrements of the objects $o_1, o_2, \ldots, o_n$ are executed, might prevent noting that one of the decrements creates a new unreachable cycle. Moreover, not only we are able to collect all garbage cycles, but we also manage to do it while considering less objects as candidates. In our cycle collector, when a pointer p takes the values $o_0, o_1, o_2, \ldots, o_n$ between two collections, at most o_0 would be considered as a candidate (if its reference count was decremented to a non-zero value), whereas previous cycle collectors have considered at most n objects as candidates ($o_0, o_1, o_2, \ldots, o_{n-1}$).

We shall next provide a non-formal proof showing that at least one object from each garbage cycle, would be considered as a candidate, and hence each garbage cycle would be collected. We divide the proof into 2 cases: garbage cycles comprising solely of old objects and garbage cycles containing at least one young object. We show that each of those two kinds of garbage cycles are properly noticed.

First, we focus on garbage cycles which include at least one

young object (object created since the last collection). As mentioned in 2.1.1, all young objects (surviving the reference-counting collection) are considered as candidates for a garbage cycle. Doing that covers all possible cycles that contain a young object.

Next, we'll show that garbage cycles containing only old objects (those who were created before the previous sliding view) would be considered. If this cycle was reachable during the previous sliding view and is unreachable now, then an object that referenced an object in this cycle in the previous sliding view must have been modified during the time between the two sliding views. The change of this pointer is logged in a mutation buffer and a decrement to the reference count of the object previously referenced must be executed. At that point, this object must become a candidate for cycle collection. Hence, cycles containing only old objects are accounted for properly.

To summarize, using the Levanoni-Petrank write-barrier we consider any object whose reference count is decremented to a non-zero value, as well as any young object to be a candidate.

3.1.1 Using the age-oriented collector

Our measurements show that the cycle collector spends a substantial fraction of its time checking cycle candidates among newly allocated objects. An alternative collector that ameliorates this problem is the age-oriented collector [32]. The age-oriented collector is an on-the-fly collector which uses tracing to reclaim young objects and reference counting for the old objects. This collector turns out to be a perfect match to our algorithm. It eliminates a large fraction of the cycles as well as a large fraction of the cycle collector's work since it does not need to consider the young objects as candidates. We have incorporated our cycle collector into the age-oriented collector and provide measurements for the resulting collector as well as for the standard reference counting collector.

3.2 Reducing the number of traced objects

The effectiveness of the cycle collection algorithm heavily depends on the number of objects traced by the cycle collector. Therefore, finding strategies to reduce the number of objects traced is important for the algorithm performance. An example of such strategy is trying to reduce the number of objects which are candidate roots of garbage cycles. Our algorithm applies several strategies. Some of them have been used before and others are new.

Two techniques for reducing objects' tracing are employed in previous collectors and were already mentioned in 2.1. The first idea is to record the values of the candidates in a buffer hoping that with time most of them will be identified as non-candidates (for example, if they are reclaimed by the reference counting collector). The second idea is to performs each of the three stages (mark, scan and collect) in its entirety on all the candidates simultaneously.

Another strategy for eliminating candidates we use (introduced by [4]) concerns acyclic object. Some objects are inherently acyclic, (i.e., can not be a part of a cycle), e.g., an array of scalars. Such objects are statistically determined. Acyclic objects are not considered as candidates roots of the algorithm. Moreover, an acyclic object reached during the algorithm traversals would be ignored. Since usually a considerable portion of the objects are acyclic (sometimes even a majority), ignoring acyclic objects significantly reduces the overhead of cycle collection.

It is reported in [3], that the combination of the above candidate filtering strategies is highly effective reducing the number of possible candidates by at least a factor of seven (and reducing the worst case algorithm complexity from quadratic to linear). However, their measurement show that only a small percentage of the non-filtered candidates indeed belong to garbage cycles. We feel that filtering techniques are a key component in improving the efficiency of cycle collection algorithms, and hence we propose a couple of additional elimination strategies which reduce both the number of candidates, as well as the number of objects actually traced during a candidates traversal.

The cycle collector of Bacon and Rajan is triggered every fixed number of garbage collections, or within an earlier collection if the candidates buffer has exceeded a threshold size. Our new collector runs cycle collection with each garbage collection. However, we let the candidates "mature" before actually testing them for membership in an unreachable cyclic structure. As discussed above, while candidates wait to be checked, many of them are already reclaimed and may be removed from the candidate list. Our strategy is to check only the candidates which were accumulated k collections ago and were not filtered. Thus, instead of having one large candidates buffer, we employ $k+1$ smaller candidates buffers, each containing candidates accumulated in different collections, and in the current collection we check the oldest buffer. Whenever an object is added to the current candidates buffer, it is removed from an older buffer, if recorded in any. (This removal is executed in a short processing of the buffers in the end of each collection.)

Note that this is a more structured way to let candidates mature. If we run the cycle collector each k collections on all candidates, then we also run it on candidates that were recently discovered and have not yet matured. Employing the new strategy, we trace only candidates that have not been filtered out (and have not died) throughout the last k collections.

Next, we note that several objects are known to be alive in the beginning of the current collection. This includes objects that were directly reachable from the roots, objects that were snooped and objects that are dirty since they were modified after the sliding view was taken. In addition, there are objects that are currently held as candidates in newer candidate buffers. All such objects get special treatment during the trace. In particular, when the cycle collector reaches any such object during the mark stage, it ignores it (if it is black there is no need to mark it gray and trace it; if it is gray-no need to decrement its CRC). In addition, if such a gray object is reached during the scan phase (i.e., it was modified after it was colored gray), then this object is colored black and so are all its (sliding-view) descendants.

Finally, to save more scanning time, we add a small additional stage between the mark phase and the scan phase. The scan stage may sometimes color white an object together with its sub-graph, only to find out further in this stage that this object was referenced from another gray object which is externally referenced. In this situation, the object (and sub-graph) would be colored black, using a second traversal in the scan stage. Such repeated traversals can be saved, if we start by coloring black the externally referenced objects together with their subgraph, and only then color white all the rest. Indeed, we cannot find all externally referenced objects, but we may have an approximate list, collected during the mark phase, which includes all objects encountered that are known to be alive (modified object, snooped object, object directly referenced by the system roots, and candidate located in a newer candidate buffer). After running the mark phase and before starting the scan phase, we color all their descendants black.

One disadvantage of the above filtering, is that we must use the additional *CRC* field as in the asynchronous cycle collector of Bacon and Rajan. (See Section 2.2 for explanations on the *CRC* field.) The reason we must use it is that the set of live objects (actually, only the subset of dirty objects) is not fixed during the algorithm. Thus, it is safe not to trace the sub graphs that we do not trace, but it is not clear that we do not eliminate more tracing as the stages

proceed. This necessitates the use of the *CRC* field. It is possible to avoid using it, if we avoid using the dirty bit to identify living objects (or by employing an additional mechanism to handle the "problematic" objects).

4. AN IMPLEMENTATION FOR JAVA

We have implemented our algorithm in Jikes [1], a Java virtual machine (upon Linux Red-Hat 7.2). The entire system, including the collector itself is written in Java (extended with unsafe primitives available only to the Java Virtual Machine implementation to access raw memory). Jikes uses *safe-points*: rather than interrupting threads with asynchronous signals, each thread periodically checks a bit in a condition register that indicates that the runtime system wishes to gain control. This design significantly simplifies implementing the handshakes of the garbage collection. In addition, rather than implementing Java threads as operating system threads, Jikes multiplexes Java threads on *virtual-processors*, implemented as operating-system threads. Jikes establishes one virtual processor for each physical processor.

4.1 Memory allocator

Our implementation employs the non-copying allocator of Jikes, which is based on the allocator of Boehm, Demers, and Shenker [10]. This allocator is well suited for collectors that do not move objects. Small objects are allocated from per-processor segregated free-lists build from 16KB pages divided into fixed-size blocks. Large objects are allocated out of 4KB blocks with first-fit strategy. This allocator keeps the fragmentation low and allows efficient reclamation of objects.

4.2 Implementation issues

We would like to point out some implementation choices that we made. First, we use the mechanism of Bacon et al [3] to keep the reference counts in a field of several bits and cache them in an auxiliary table in case of overflow.

In our implementation both reference counts, the color and the buffer status are stored in a single 32-bit word in the object header. As we use 3 colors, 2 bits are necessary for color representation. 2 bits are also needed to represent the 4 buffered status. The rest is devoted to representing the RC and CRC. Each count includes an overflow bit. When the overflow bit is set, the excess count is stored in a hash table. In practice, this hash table never contains more than a few entries.

During a collection, the set of objects referenced by the system roots is determined. The objects belonging to this set which have a zero reference count, are checked in the next collection, to see whether they have become garbage. The objects belonging to this set which have a non-zero reference count, and are not referenced by the system roots in the next collection, should be pushed into the candidate buffer of the next collection: they were de-referenced between the two collections.

In their collector, Bacon and Rajan [4] perform all reference count increments of the last epoch, and then all reference count decrements of the epoch before. Thus, when a reference count of a candidate is incremented, it is no longer considered as a candidate (unless its reference count would be decremented later). Since the sliding views reference counting collector interleaves decrements and increments of reference counts, we cannot ignore candidates whose reference counting was recently incremented, but only consider them as newly-buffered (and therefore delay their tracing). However, Bacon and Rajan report that this filtering technique removes a very small amount of candidates (and often none). Our measurements agree with this finding and so the loss is not noticeable.

5. MEASUREMENTS

u

Platform and benchmarks. We have run our measurements on a 4-way IBM Netfinity 8500R server with a 550MHz Intel Pentium III Xeon processor and 2GB of physical memory. The benchmarks we used were the SPECjvm98 benchmark suite and the SPECjbb2000 benchmark. These benchmarks are described in detail in SPEC's Web site [34]. We feel that the multithreaded SPECjbb2000 benchmark is more interesting, as the SPECjvm98 are more appropriate for clients and our algorithm is targeted at servers. We also feel that there is a dire need in academic research for more multithreaded benchmarks. In this work, as well as in other recent work (see for example [3, 17]), SPECjbb2000 is the only representative of large multithreaded applications.

Testing procedure. We used the benchmark suite using the test harness, performing standard automated runs of all the benchmarks in the suite. Our standard automated run runs each benchmark five times for each of the JVM's involved (each implementing a different collector).

Finally, to understand better the behavior of our collector under tight and relaxed conditions, we tested it on varying heap sizes. For the SPECjvm98 suite, we started with a 32MB heap size and extended the sizes by 8MB increments until a final large size of 96MB. For SPECjbb2000 we used larger heaps, starting from 256MB heap size and extending by 64MB increments until a final large size of 704MB.

The compared collectors. We have incorporated the cycle collection algorithm into two collector: the Levanoni-Petrank reference counting collector ([26]), and the more efficient age-oriented collector ([32]). Both collectors are also implemented in Jikes and are accompanied by a backup mark and sweep collector which is run infrequently to collect garbage cycles. For performance measurements, we ran both collectors accompanied with our cycle collection algorithm against both collectors when using the backup mark and sweep algorithm. In addition, we have compared characteristics of our cycle collection algorithm (with both collectors), against the characteristics of the previous on-the-fly cycle collector of Bacon and Rajan [3].

5.1 Server performance

Our major benchmark is the SPECjbb2000 benchmark. SPECjbb2000 requires multi-phased run with an increasing number of warehouses. The benchmark provides a measure of the throughput and we report the throughput ratio improvement when applied with the proposed cycle collection algorithm (compared to the same collector with the backup mark and sweep algorithm). Thus, the higher the ratio, the better our algorithm behaves, and in particular, any ratio larger than 1 implies that the cycle collector outperforms the tracing auxiliary collector.

We would like to stress that our algorithm is applied in each collection, and hence incurs overhead in each collection. However, applying the mark and sweep algorithm, does not produce any special overhead, since it is applied infrequently instead the original collector, and not as an add-on.

The measurements are reported for a varying number of warehouses and varying heap sizes. The measurements of the cycle collector with the Levanoni-Petrank reference counting collector are presented in Figure 1. The behavior of the algorithm should be separated into two cases.

The first case is with 1-3 warehouses. In this case, since our

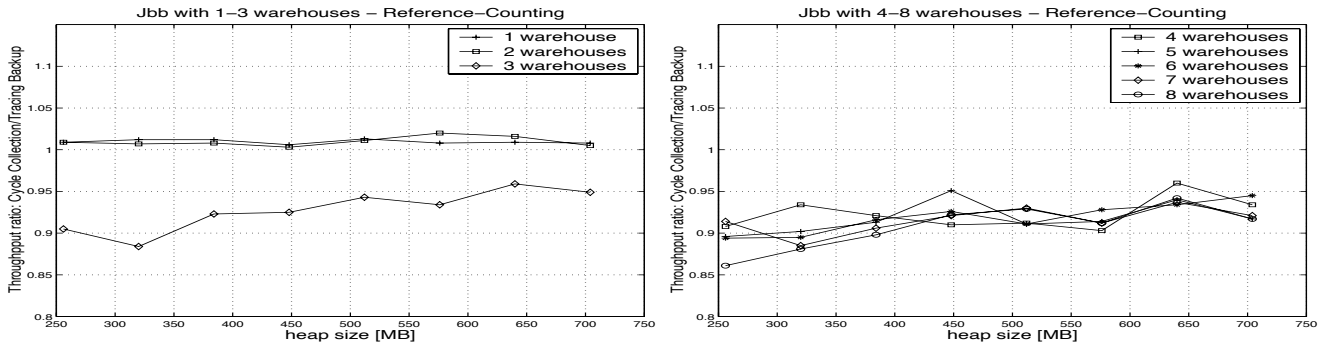

Figure 1: SPECjbb2000 on a multiprocessor: throughput ratio for the Levanoni-Petrank collector

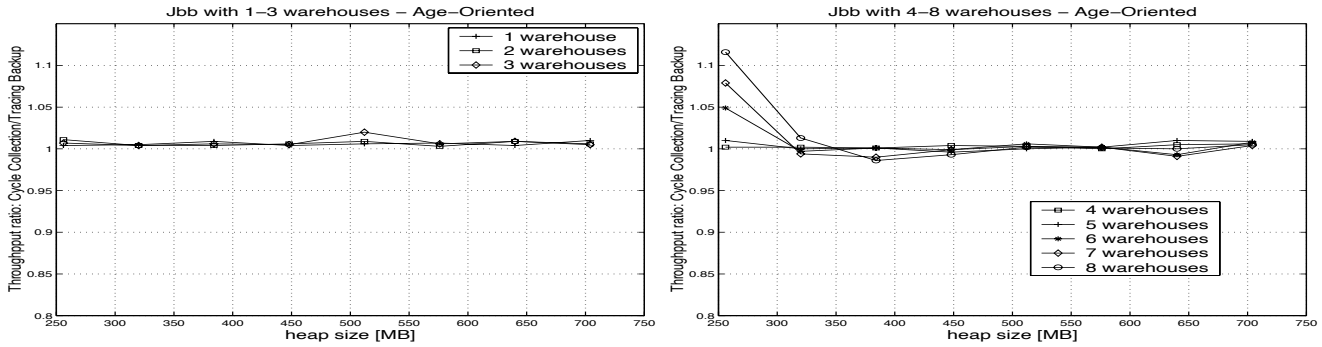

Figure 2: SPECjbb2000 on a multiprocessor: throughput ratio for the age-oriented collector

machine has four processors, both two on-the-fly compared collectors run on a spare processor. In this setting, if the collectors could handle all their work while mutators are running (except for handshakes), the collectors achieve the same throughput (as they share similar allocator and write-barrier). This is indeed what we get with 1-2 warehouses. However, with 3 warehouses the backup mark and sweep collector performs better (by usually between 5%-10%). The reason is that our collector does not make it in releasing space fast enough for allocations. Thus, the program sometimes is delayed waiting for the collector to finish. As expected, the overhead of our algorithm is more noticeable on tight heaps, where collections are run more frequently.

The second case refers to 4-8 warehouses, where collectors do not run on a spare processor but rather share a processor with the program threads. (Nevertheless, we gave the collector the highest priority, so that when a collection is triggered the collector always gets a dedicated processor.) Thus, when the number of warehouses is four and up, the efficiency of the collector becomes more important: a collector should not only be able to handle all its work while mutators are running, but also as the collector becomes more efficient, a collection would consume less time, thus letting mutators use a larger fraction of the fourth processor (and therefore increasing the throughput). As with 3 warehouses, the results show that with 4-8 warehouses, the backup mark and sweep collector also performs better (by usually between 6%-10%). As before, on tighter heaps, where the collector efficiency is more significant, the overhead of the cycle collector is larger.

The same measurements have also been run on the age-oriented collector. This is a more efficient collector that runs mark-and-sweep on new objects and reference-counting on old objects ([32]). The results are presented in Figure 2. When using our cycle collector with the age-oriented collector, a large fraction of the cycle collector's work is eliminated, as newly allocated objects are not considered cycle candidates. Our measurements show that we achieve similar performance to the one achieved when using the backup mark and sweep algorithm. Hence, not only the age-oriented collector accompanied by our cycle collector is able to handle all its work while mutators are running, but also its collection time is hardly affected by the added cycle collection activity.

When running the SPECjvm98 benchmarks on a multiprocessor, we allow a designated processor to run the collector thread. Here again the collector runs concurrently with the program thread(s) and good concurrency is the main factor in the comparison. Results show that both the reference counting collector and the age-oriented collector with the cycle collection algorithm usually perform similarly to the reference counting (or age-oriented) collector applied with the backup mark and sweep (Figure 3). The only clear noticeable difference, is in the case of reference counting collector using the backup mark and sweep, with the _227_mtrt benchmark. The reason for this difference is that for this benchmark there exists an initial phase in which many objects are created and kept alive till the end of the run. These newly created objects require a large amount of work of the cycle collector. After this initial phase, the cycle collector does not incur a large overhead. However, in this single long collection, the mutator needs to halt waiting for free space. The performance difference is noticeable only with the reference counting collector, and not with the age-oriented collector. There, the cycle collector is not run on this pack of young objects that are all alive.

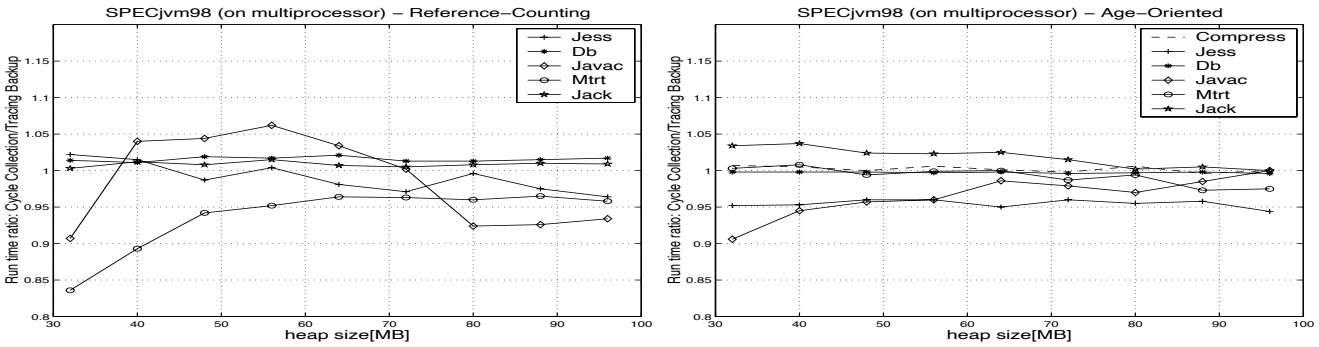

Figure 3: SPECjvm98 on a multiprocessor: run-time ratio .

Bench-marks	RC		AO	
	cyclic objects reclaimed	Cyclic bytes reclaimed	cyclic object reclaimed	Cyclic bytes reclaimed
compress	108	84.08 MB	0	0
jess	24	0.15 MB	0	0
db	16	0.09 MB	0	0
javac	1 M	67.64 MB	0.57 M	37.02 MB
mtrt	66052	5.78 MB	66042	5.66 MB
jack	8976	1.72 MB	3360	0.62 MB
jbb	146	0.88 MB	0	0

Table 1: Cyclic garbage collected for each benchmark by our cycle collector, when incorporated with the reference counting and the age-oriented collectors.

5.2 Client performance

Although our cycle collection algorithm is targeted at servers running on SMP platforms, as a sanity check, we have also measured its performance against the same collector with the backup mark and sweep algorithm on a uniprocessor. The behavior of the collector on a uniprocessor may demonstrate its efficiency. We measured the cycle collection algorithm on a uniprocessor with the SPECjvm98 benchmark suite. Results appear in Figure 4. When using reference counting collector, the incorporation of the cycle collector is usually within 10% of the incorporation of a backup tracing collector. When using the age-oriented collector, the cycle collector performs similarly to the backup mark and sweep collector. The only bad exception is encountered with the _213_javac benchmark, which produces hundreds of thousands of objects in garbage cycles. Dealing with these objects is time consuming.

5.3 Collector characteristics

5.3.1 Amount of cyclic garbage

We have measured for each benchmark, the number of garbage cycle objects that were reclaimed and the space they consume. These measurements are reported in Table 1. The amount of cyclic garbage handled by our cycle collection algorithm varies according to the collector it is incorporated into (reference counting or age-oriented), and hence measurements are separated into the 2 cases. As the age-oriented collector reclaims garbage cycles of the young generation without using the cycle collector, the cycle collector reclaims sometimes a smaller set of garbage cycles when run with the age-oriented collector.

_213_javac and _201_compress are the only benchmarks containing garbage cycles that constitute a substantial amount of space. In _201_compress, there are some dozens of garbage cycles comprising of huge objects, and it hence requires only a small amount of tracing. _213_javac however, contains thousands of garbage cycles (some of them containing thousands of objects), thus requiring a large amount of tracing by our cycle collection algorithm.

5.3.2 Amount of tracing

Several new strategies have been suggested in this paper to reduce the number of candidates and the number of objects traced in order the reduce the overhead introduced by the cycle collection algorithm. We have measured the effectiveness of those strategies, by measuring the reduction in the number of candidate objects and traced objects. We report the ratio of the candidates traced and objects traced when compared to those of [4]. Note that we only measure the number of objects traced. In the original algorithm of [4], objects are processed again to verify safety. That processing is not included in the measurements of traced objects. Our new algorithm does not run this extra processing, and so, the saving of the new algorithm is even better than what we report in these measures: only the improved filtering is measured and not the overall improved algorithm.

In these graphs, the lower the ratio, the better our algorithm behaves, and any ratio smaller than 1 implies that it traced less candidates and objects. As before, we have measured our algorithm with both the reference counting collector (denote *RC*) and the age-oriented collector (denote *AO*). Each collector characteristics was measured with 2 and 3 candidates buffers. In the first case, the implication is that candidates gathered until the current collection will be considered only at the next collection, i.e., there is a delay of one collection before handling the collected candidates. In the latter case, there is a delay of two collections. We thus denote these two cases by *delay1* and *delay2*. Hence, we have measured our algorithm in four configurations depending on the delay and the underlying collector.

The results are introduced in Figure 5. One may see that all 4 configurations trace less candidates compared to the previous cycle collector (of [4]) with all benchmarks. Also, when checking how many objects were traced, the new cycle collector traces substantially less objects except for one case: the _227_mtrt benchmark with the reference counting configurations. The superiority of the age-oriented configurations over the reference counting configurations is also very clear, and demonstrates the work saved by not having to collect cycles from the young object and not having to consider young objects as candidates.

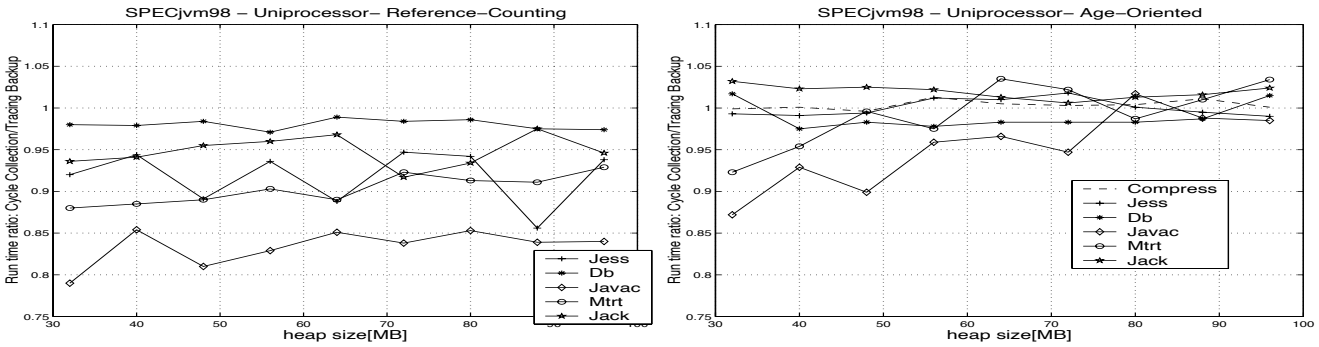

Figure 4: SPECjvm98 on a uniprocessor: run-time ratio.

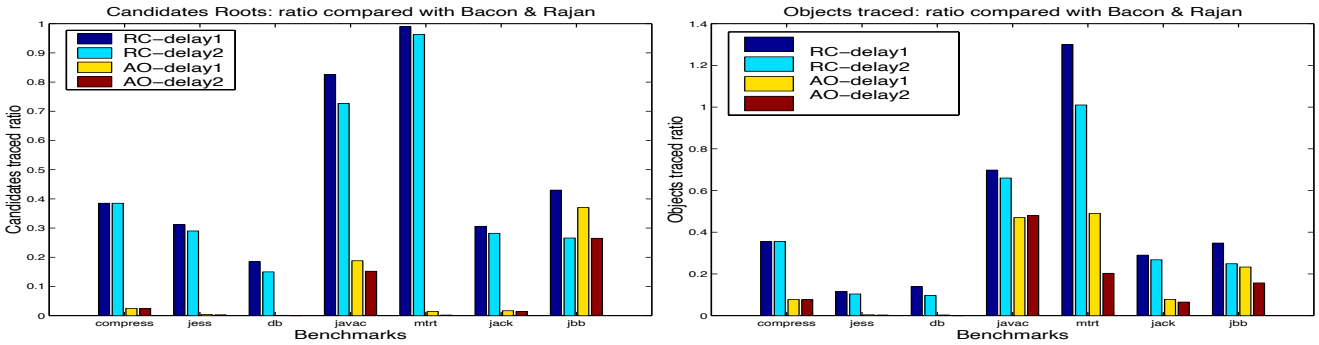

Figure 5: saving in tracing work and number of candidates.

When using three buffers the algorithm traces less then when using two buffers, which means that by delaying the handling of a candidate, its traversal may sometimes be spared. However, when measuring benchmarks' throughput in these two cases, there was mostly no clear throughput superiority. This is due to the fact that the handling of candidates buffer (filtering candidates each collection) also consumes time. Hence, one should tune the number of candidates buffers used according to its collector and its collection triggering policy. For our collectors, using two candidates buffers was usually enough, but for collectors that use frequent collections, we believe that using more buffers would improve efficiency.

5.3.3 Candidates filtering

We have examined the exact effectiveness of each candidate filtering strategy we have added, by measuring the fraction of candidates it has filtered. To do that, we have measured these strategies with the base reference counting collector (of [26]) using two or three candidates buffers, i.e., with a delay of one or two collections. In Figure 6, the fraction of candidates which were filtered out is presented for delays of one and two cycles. In each such collection, we filter each candidate that was reclaimed or buffered again. The fraction of the rejected candidates due to a specific filter is presented in by the partition of each bar.

We stress that these filtering techniques are executed on the candidates that have survived the filtering techniques used by Bacon and Rajan (in [4]). These techniques include filtering candidates which were reclaimed during the collection in which they were buffered, ignoring acyclic objects, and preventing duplicate candidates in the same candidate buffer. Those techniques were reported to be highly effective (see [3]). The fraction we present is the *addi-*

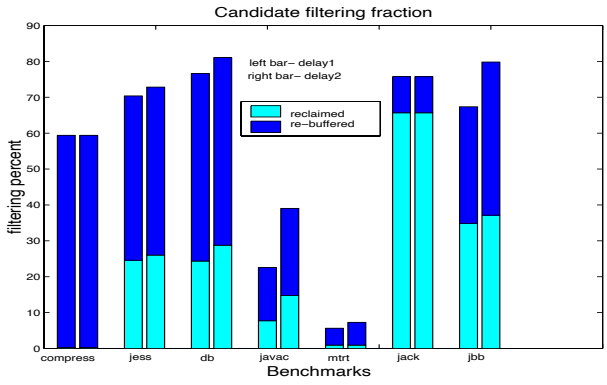

Figure 6: Candidates filtering: percentage of candidates filtering partitioned to objects that were reclaimed during the delay and objects that were re-buffered during the delay (and thus were filtered). For each benchmark, the left bar represents a delay of one collection cycle. The right bar represents a delay of two collection cycles.

tional benefit obtained by the new filtering methods.

The results show that these new techniques usually filter between 40%-80% of the surviving candidates. The only benchmark for which these techniques are not effective is the _227_mtrt benchmark, where only 7.26% of the candidates were eliminated. This explains why _227_mtrt is the only benchmark for which the number of candidates traced is similar to that of [4].

It can be seen that candidates are effectively eliminated both because they are reclaimed and because they are newly buffered. In terms of delaying the check, the first cycle delay is far more effective than the delay of the second cycle. Nevertheless, for some benchmarks (such as _213_javac and the SPECjbb2000 benchmarks) the filtering benefit obtained at the second cycle delay is non-negligible.

6. RELATED WORK

The traditional method of reference counting applicable in the realm of uniprocessing was first developed for Lisp by Collins [12]. In its simplest form, it allowed immediate reclamation of garbage in a localized manner, yet with a notable overhead for maintaining the space and semantics of the counters. Weizenbaum [37] showed how the delay introduced by recursive deletion (which is the only non-constant delay caused by classic reference counting) can be ameliorated by distributing deletion over object creation operations. Deutsch and Bobrow [13] eliminated most of the computational overhead required to adjust reference counters in their method of deferred reference counting. According to the method, local references are not counted thus the need to track fetches, local pointer duplication and cancellation are deemed unnecessary. Only stores into the heap need be tracked. However, the immediacy of reference counting is lost in a certain extent, since garbage may be reclaimed only after the mutator state is scanned. Bacon et al [3] and Levanoni and Petrank [26] have extended the reference counting algorithm to running concurrently with the program threads. Both achieve extremely low pauses times (or around 2ms).

The inability of reference-counting to reclaim cyclic garbage structures was first noticed by McBeth [29]. Christopher [11] developed an algorithm, whose primary method is reference counting, yet a tracing collector is called periodically to reclaim nodes in the heap that have a non-zero reference counts but are not externally reachable. The algorithm of Martinez et al. [28] reclaims cells, which were uniquely referenced when their count drops to zero, while when a pointer to a shared object is deleted, a local depth-first search is applied on it. This search subtracts reference counts due to internal pointers. If a collection of objects with zero reference counts is found, then a garbage cycle has been found, and is collected. Lins [27] extended this algorithm by postponing the above traversals while saving the values of the deleted pointer in a buffer (each such value is a candidate to be a root of a garbage cycle) and traversing the buffer at a suitable point. Delaying the traversal decreases the number of actual candidates traversals since most buffer's values are irrelevant by the time the buffer is traversed (because their reference counts either drop to zero or is incremented). Bacon et al [4] extended Lins algorithm to a concurrent cycle collection algorithm. They also improved Lins' algorithm by performing the tracing of all candidates simultaneously, reducing the number of traced objects.

7. CONCLUSIONS

We presented a new cycle collector for reference counting which is on-the-fly and is based on combining techniques from the sliding views collectors of [26, 2] and the on-the-fly cycle collector of [3, 4]. We gain efficiency and simplicity by running the simpler synchronous cycle collector of Bacon and Rajan [4] on sliding views of the heap. This eliminates work previously required to ensure correctness when running concurrently with the program. In addition, we add more filtering techniques that manage to filter out a large fraction of the cycle candidates. The resulting collector is an efficient cycle collector which retains the very short pauses of the on-the-fly reference counting collectors in [3, 26]. Finally, the new collector is guaranteed to reclaim all garbage cycles, whereas the previous on-the-fly collector has an (extremely rare) sequence of events that prevents it from collecting an unreachable cyclic structure forever.

We provide the first direct comparison of running a cycle collector against running a backup tracing collector. With current benchmarks, the backup tracing collector seems to have a small advantage. We expect reference counting to win with future large heaps, and as such, an accompanying cycle collector will be required. This work provides such a companion cycle collector. When incorporating the new cycle collector into the age-oriented collector to save the work on collecting young generation cycles, the cycle collector performs equally to the backup tracing collector.

8. ACKNOWLEDGEMENTS

Ram Natahniel initiated our discussion on this problem by suggesting to use algorithms for strongly connected components to efficiently locate garbage cycles. Our attempts to follow this direction failed, but this paper has evolved. We thanks Ram for many interesting discussions.

9. REFERENCES

[1] Bowen Alpern, C. R. Attanasio, Anthony Cocchi, Derek Lieber, Stephen Smith, Ton Ngo, John J. Barton, Susan Flynn Hummel, Janice C. Sheperd, and Mark Mergen. Implementing Jalapeño in Java. In *OOPSLA'99 ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 34(10) of *ACM SIGPLAN Notices*, pages 314–324, Denver, CO, October 1999. ACM Press.

[2] Hezi Azatchi, Yossi Levanoni, Harel Paz, and Erez Petrank. An on-the-fly mark and sweep garbage collector based on sliding view. In OOPSLA [31].

[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. In *Proceedings of SIGPLAN 2001 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Snowbird, Utah, June 2001. ACM Press.

[4] David F. Bacon and V.T. Rajan. Concurrent cycle collection in reference counted systems. In Jørgen Lindskov Knudsen, editor, *Proceedings of 15th European Conference on Object-Oriented Programming, ECOOP 2001*, volume 2072 of *Springer-Verlag*, Budapest, June 2001. Springer-Verlag.

[5] Henry G. Baker. List processing in real-time on a serial computer. *Communications of the ACM*, 21(4):280–94, 1978. Also AI Laboratory Working Paper 139, 1977.

[6] Mordechai Ben-Ari. On-the-fly garbage collection: New algorithms inspired by program proofs. In M. Nielsen and E. M. Schmidt, editors, *Automata, languages and programming. Ninth colloquium*, pages 14–22, Aarhus, Denmark, July 12–16 1982. Springer-Verlag.

[7] Mordechai Ben-Ari. Algorithms for on-the-fly garbage collection. *ACM Transactions on Programming Languages and Systems*, 6(3):333–344, July 1984.

[8] Stephen M. Blackburn and Kathryn S. McKinley. Ulterior referene counting: Fast garbage collection without a long wait. In OOPSLA [31].

[9] Daniel G. Bobrow. Managing re-entrant structures using reference counts. *ACM Transactions on Programming Languages and Systems*, 2(3):269–273, July 1980.

[10] Hans-Juergen Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. *ACM SIGPLAN Notices*, 26(6):157–164, 1991.

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

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

[13] L. Peter Deutsch and Daniel G. Bobrow. An efficient incremental automatic garbage collector. *Communications of the ACM*, 19(9):522–526, September 1976.

[14] Edsgar W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):965–975, November 1978.

[15] Damien Doligez and Georges Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *Conference Record of the Twenty-first Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices. ACM Press, January 1994.

[16] Damien Doligez and Xavier Leroy. A concurrent generational garbage collector for a multi-threaded implementation of ML. In *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, pages 113–123. ACM Press, January 1993.

[17] Tamar Domani, Elliot Kolodner, and Erez Petrank. A generational on-the-fly garbage collector for Java. In *Proceedings of SIGPLAN 2000 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Vancouver, June 2000. ACM Press.

[18] Tamar Domani, Elliot K. Kolodner, Ethan Lewis, Elliot E. Salant, Katherine Barabash, Itai Lahan, Erez Petrank, Igor Yanover, and Yossi Levanoni. Implementing an on-the-fly garbage collector for Java. In Hosking [21].

[19] John R. Ellis, Kai Li, and Andrew W. Appel. Real-time concurrent collection on stock multiprocessors. Technical Report DEC–SRC–TR–25, DEC Systems Research Center, Palo Alto, CA, February 1988.

[20] David Gries. An exercise in proving parallel programs correct. *Communications of the ACM*, 20(12):921–930, December 1977.

[21] Tony Hosking, editor. *ISMM 2000 Proceedings of the Second International Symposium on Memory Management*, volume 36(1) of *ACM SIGPLAN Notices*, Minneapolis, MN, October 2000. ACM Press.

[22] Richard E. Jones and Rafael D. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, July 1996.

[23] H. T. Kung and S. W. Song. An efficient parallel garbage collection system and its correctness proof. In *IEEE Symposium on Foundations of Computer Science*, pages 120–131. IEEE Press, 1977.

[24] Leslie Lamport. Garbage collection with multiple processes: an exercise in parallelism. In *Proceedings of the 1976 International Conference on Parallel Processing*, pages 50–54, 1976.

[25] Yossi Levanoni and Erez Petrank. A scalable reference counting garbage collector. Technical Report CS–0967, Technion — Israel Institute of Technology, Haifa, Israel, November 1999.

[26] Yossi Levanoni and Erez Petrank. An on-the-fly reference counting garbage collector for Java. In *OOPSLA'01 ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 36(10) of *ACM SIGPLAN Notices*, Tampa, FL, October 2001. ACM Press.

[27] Rafael D. Lins. Cyclic reference counting with lazy mark-scan. *Information Processing Letters*, 44(4):215–220, 1992. Also Computing Laboratory Technical Report 75, University of Kent, July 1990.

[28] A. D. Martinez, R. Wachenchauzer, and Rafael D. Lins. Cyclic reference counting with local mark-scan. *Information Processing Letters*, 34:31–35, 1990.

[29] J. Harold McBeth. On the reference counter method. *Communications of the ACM*, 6(9):575, September 1963.

[30] David A. Moon. Garbage collection in a large LISP system. In Guy L. Steele, editor, *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 235–245, Austin, TX, August 1984. ACM Press.

[31] *OOPSLA'03 ACM Conference on Object-Oriented Systems, Languages and Applications*, ACM SIGPLAN Notices, Anaheim, CA, November 2003. ACM Press.

[32] Harel Paz and Erez Petrank. Age-oriented garbage collection. Technical Report CS-2003-08, Technion, Israel Institute of Technology, October 2003. http://www.cs.technion.ac.il/users/wwwb/cgi-bin/tr-info.cgi?2003/CS/CS-2003-08.

[33] Tony Printezis and David Detlefs. A generational mostly-concurrent garbage collector. In Hosking [21].

[34] SPEC Benchmarks. Standard Performance Evaluation Corporation. http://www.spec.org/, 1998,2000.

[35] Guy L. Steele. Multiprocessing compactifying garbage collection. *Communications of the ACM*, 18(9):495–508, September 1975.

[36] Guy L. Steele. Corrigendum: Multiprocessing compactifying garbage collection. *Communications of the ACM*, 19(6):354, June 1976.

[37] J. Weizenbaum. Symmetric list processor. *Communications of the ACM*, 6(9):524–544, September 1963.

APPENDIX

A. THE GARBAGE COLLECTOR DETAILS

In this section, pseudo-code and explanations regarding the new cycle collection algorithm are provided.

A.1 The log-pointer

The original reference counting algorithm requires maintaining a dirty bit signifying whether an object has been modified since the most recent collection started. During the first modification of an object in a cycle, its pointers are recorded in the *updates* buffer and its dirty bit is set. We follow [26, 2] by choosing to dedicate a full word to keep the dirty bit. Indeed, this consumes space, but it allows keeping information about the dirty object. In particular, this word is used to keep a pointer into the thread's local buffer where this object's pointers have been logged. A zero value (a null

```
Procedure Update(o: object, offset: int , new: object)
begin
1.      if o.LogPointer=NULL then // object not dirty
2.          TempPos := CurrPos
3.          foreach field ptr of o which is not NULL
4.              Buffer[++TempPos] := ptr
5.          // is it still not dirty?
6.          if o.LogPointer=NULL then
7.              // add pointer to object
8.              Buffer[++TempPos] := address of o
9.              //committing values in buffer
10.             CurrPos := TempPos
11.             // set dirty
12.             o.LogPointer = address of Buffer[CurrPos]
13.     write( o, offset ,new)
14.     if Snoop and new != NULL then
15.         Snooped := Snooped ∪ { new }
end
```

Figure 7: Mutator code: Update Operation

```
Procedure Add-Candidate(cand: object)
begin
1.      if cand.status = Newly-Buffered ∨ is-Acyclic(cand) then
2.          return
3.      cand.status := Newly-Buffered
4.      push cand to newCandidatesBuffer
end
```

Figure 8: Add-Candidate

```
Procedure Process-Cycles
begin
1.      Mark-Candidates
2.      Scan-Black-Live-Stack
3.      Scan-Candidates
4.      Collect-White
5.      Process-Buffers
end
```

Figure 9: Process-Cycles

pointer) signifies that the object is not dirty (and not logged). We call this word the `LogPointer`. Justification to this choice is provided in the original sliding views papers and is not repeated here.

Tracing a sliding view makes good use of the `LogPointer` field. When an object is scanned, the `LogPointer` is checked. If it is null, then the current state of the object may be used. Otherwise, it provides a pointer to the log entry where the state of the object in the sliding view is recorded.

Procedure Update (Figure 7) is activated at pointer assignment and its main task is to record the object whose pointer is modified (i.e., log objects values at the sliding views). We stress that the write barrier (the Update protocol) is only used with heap pointer modification. Modifications of local pointers in the registers or stack are not monitored. Going through the pseudo-code, we see that each object's LogPointer is optimistically probed twice (lines 1 and 6) so that if the object is dirty (which is often the case), then the write barrier is extremely fast. If the object was not logged (i.e., the LogPointer of an object is NULL) then after the first probe, the objects values are recorded into the local $Updates_i$ (lines 2-4). The second probe at line 6 ensures that the object has not yet been logged (by another thread). If *LogPointer* is still NULL (in the second probe), then the recorded values are committed as the buffer pointer is modified (line 10). In order to be able to distinguish later between objects and logged values, in line 8 we actually log the object's address with the least significant bit set on (while values are logged with least significant bit turned off). Then, the object's *LogPointer* field is set to point to these values (line 12). After logging has occurred, the actual pointer modification happens. Finally, from the time a collection begins until marking the roots of the mutators, the snoop flag is on. At that time, the new target of the pointer assignment is recorded in the local $Snooped_i$ buffer. This happens in lines 14-15. The variables $Updates_i$, *CurrPos*, $Snoop_i$ and $Snooped_i$ are local to the thread.

We do not elaborate on the properties of the write-barrier, on why it works in a multithreaded environment, etc. A thorough discussion of the write barrier appears in the original paper [26].

A.2 General issues

A.2.1 candidate objects status

In order to process the candidates buffers, we keep a state with each object. An object is allocated in state *non-buffered*. When it is first buffered it is marked *newly-buffered*. During each collection, all buffers are processed. Each *newly-buffered* object is removed from all older buffers. An object that is a member of the oldest buffer (the buffer that is currently being checked for cycles) is marked *old-buffered*. All other buffered objects (in buffers that are not the youngest or oldest buffers) are marked *mid-buffered*. The candidates buffers are denoted, in a corresponding manner, *newCandidatesBuffer*, *midCandidatesBuffer* and *oldCandidatesBuffer*. Of-course, there may be several buffers of type *midCandidatesBuffer*.

A.2.2 Assumed procedures

In the pseudo code we assume the existence of some simple methods. These include:

- is-Acyclic: checks whether an object is inherently acyclic.
- is-Buffered-Not-Old: checks whether an object is buffered but not in the oldest buffer.
- is-Released: checks whether an objects was released in the current collection. As this method is called by the collector, the *current collection* is well defined.

A.3 Interface with the reference counting collector

Procedure Add-Candidate (Figure 8) is called by the reference counting collector in order to insert an object into the *newCandidatesBuffer*. If this object is already buffered in the *newCandidatesBuffer*, or if it is acyclic, the object is ignored. Else, it is buffered in *newCandidatesBuffer* after its state is modified into newly-buffered.

An interface in the opposite direction is the ability of the cycle collector to call the RC-Free procedure, which performs the recursive deletion of an object. It is invoked by Procedure Collect described below.

A.4 Cycle algorithm code

The cycle algorithm's code for cycle *k* is presented in **Procedure Process-Cycles** (Figure 9). This procedure is applied in every cycle collection after the reference counting collector is done collecting the non-cyclic garbage. It consists of the following phases:

- **Mark stage:** traces the graph of relevant candidates, subtracting counts (*CRC*) due to internal references and marking nodes as possible garbage (by coloring them gray).

```
Procedure Mark-Candidates
begin
1.      for each cand in oldCandidatesBuffer do
2.          if is-Released(cand) ∨ cand.status = Newly-Buffered then
3.              remove cand from oldCandidatesBuffer
4.          else if cand.color = black then
5.          // was not reached during previous candidates' traversals
6.              Mark(cand,true)
7.          else
8.              // Gray objects, i.e., descendants of other candidates.
9.              cand.status := Non-Buffered
10.             remove cand from oldCandidatesBuffer
end
```

Figure 10: Mark-Candidates

```
Procedure Mark (obj: object, isCand: Boolean)
begin
1.      if obj.color != gray then
2.          // first time reached in this mark stage
3.          obj.color := gray
4.          obj.CRC := RC
5.          if !isCand then // was reached during the recursive call
6.              obj.CRC−−
7.          // check whether object was witnessed living
8.          if obj.LogPointer != NULL ∨ obj ∈ Roots
9.                  ∨ is-Buffered-Not-Old(obj) then
10.             push obj to LiveStack // it was alive
11.         else
12.             replica := Read-Sliding-View(obj)
13.             for each o in replica do
14.                 if !is-Acyclic(o) then
15.                     Mark(o,false)
16.     else // was reached before
17.         obj.CRC−−
end
```

Figure 11: Mark

- **Scan black stage:** colors black the objects that the mark stage have considered as alive. Its purpose is to save redundant traversals as described in 3.2.
- **Scan stage:** scans the sub-graph of relevant candidates, and re-colors black objects which are reachable from external pointers. All other nodes in the sub-graph are colored white.
- **Collect stage:** scans the white sub-graphs again and reclaims all garbage (white) objects.
- **Filter candidates from buffers:** iterates over the new and middle buffers, while filtering non-relevant candidates. In addition, it performs a cyclic swapping of buffer roles.

Procedure Mark-Candidates and Procedure Mark (Figures 10-11) perform the mark stage. This stage is performed on the *oldCandidatesBuffer*'s objects which have survived all filters of the previous collections. The Mark-Candidates procedure first filters more candidates: those that were released during this collection [1] and those that were re-added to the candidates buffer in this collection (and thus are newly-buffered). Note, that it also pops out of the buffer (and clears buffer statuses of) gray objects: those objects are reachable from other candidates that have already been traced during this stage, and thus they could only belong to a garbage cycle rooted from an already traced candidate. The Mark procedure is applied on all the other candidate.

[1] The deletion of the last pointer to a shared cell will recycle it immediately, regardless of whether there is a reference to it in a candidate buffer.

```
Procedure Read-Sliding-View(cand: object)
begin
1.      // Check if object has been modified
2.      if obj.LogPointer = NULL then
3.          // read its descendants from heap.
4.          replica := copy(obj)
5.          // Check again if copied replica is valid.
6.          if obj.LogPointer != NULL then
7.              // Object has been modified while being read.
8.              // Get replica from buffers.
9.              replica := getOldObject(obj.LogPointer)
10.     else // Object has been modified. Use buffers to obtain replica.
11.         replica := getOldObject(obj.LogPointer)
12.     return replica
end
```

Figure 12: Read-Sliding-View

```
Procedure Scan-Black-Live-Stack
begin
1.      while LiveStack is not empty
2.          obj := pop(LiveStack)
3.          Scan-Black(obj)
end
```

Figure 13: Scan-Black-Live-Stack

The Mark procedure performs a depth-first traversal over the candidates' sliding-view sub-graph. An object reached for the first time is colored gray, its *CRC* is initialized and if it is not considered as alive (was not modified, is not local nor is buffered in a younger candidate buffer), its sliding-vies descendants (which are not acyclic) are traced using the Read-Sliding-View procedure. A parameter to the function is a flag telling the function whether the current object is scanned due to a reference found in the heap (and therefore an inner reference is found and the *CRC* should be decremented) or it is scanned because it is a candidate in the buffer (and therefore its *CRC* value should not be decremented). If the object has been reached before, its *CRC* is not initialized, but is decremented. Objects that are alive are pushed into the *LiveStack*, and their descendants are later colored black (these objects are not traced at this stage).

Procedure Read-Sliding-View (Figure 12) serves for getting the sliding-view values of a given object. If the object was not modified since the sliding-view was taken, its current values are also its sliding-views values. Otherwise, its pointer slots at the recent sliding view can be found by looking at the log entry which is pointed by the *LogPointer*. Note, that an object may be modified by mutators while the replica is taken (lines 5-9).

Procedure Scan-Black-Live-Stack (Figure 13) colors black the non-black objects in *LiveStack* and their non-black sliding-view descendants. The objects in *LiveStack*, where all pushed during the mark stage.

```
Procedure Scan-Black(cand: object)
begin
1.      if obj.color != black then
2.          obj.color = black
3.          replica := Read-Sliding-View(obj)
4.          for each o in replica do
5.              Scan-Black(o)
end
```

Figure 14: Scan-Black

```
Procedure Scan-Candidates
begin
1.      for each cand in oldCandidatesBuffer do
2.          Scan(cand)
end
```

Figure 15: Scan-Candidates

```
Procedure Scan (obj: object)
begin
1.      if obj.color = gray then
2.          if obj.CRC = 0 then
3.          // currently no evidence of obj being externally reachable
4.              obj.color := white
5.              replica := Read-Sliding-View(obj)
6.              for each o in replica do
7.                  Scan(o)
8.          else
9.              // mark its relevant sub-graph as alive
10.             Scan-Black(obj)
end
```

Figure 16: Scan

Procedure Scan-Black (Figure 14) is the actual procedure that colors the sliding-view's sub-graph of an object as black.

Procedure Scan-Candidates and Procedure Scan (Figures 15-16) perform the scan stage. Each gray candidate in the oldCandidatesBuffer with a non-zero CRC is considered live (and so do all its sliding-view descendants), and thus the object and its descendants are colored black. Else, it is colored white, and the scan procedure is invoked on its children. Note, that although we use the Scan-Black-Live-Stack procedure as the second stage of the algorithm, still an object may be colored white and then re-colored black.

Procedure Collect-White and Procedure Collect (Figure 17- 18) perform the collect stage. Each white candidate is a root of a garbage cycle, and thus these cycle's objects (this candidate and all white objects reachable from it) are colored black and reclaimed. Since we are dealing with a garbage cycle, whose objects are reclaimed one by one, one object may still reference another object in the cycle that was just released. Thus, when iterating over the object's descendants, one should check if the descendant is already released (line 5 in the Collect procedure). For a similar reason, we also mark any cycle object as released (line 2 in the Collect procedure), before iterating over its descendant and actually freeing it.

While reclaiming a garbage cycle, the reference counts of objects referenced by this cycle are decremented. At first it seems that the reference count of such an object could not reach zero, since if it does, then its CRC should have reached zero (during the cycle collection), and it would have been colored white (and reclaimed as part of the cycle). However, there are objects that our algorithm does not trace, such as inherently acyclic objects and objects buffered in newer candidates buffers. Such objects could be solely referenced by garbage cycles, and thus when releasing a garbage cycle, their reference count reaches zero. Hence, such objects are released using the reference counting recursive deletion (line 13 in the Collect procedure) [2]. Such recursive deletion can end-up reclaiming black candidates buffered in *oldCandidatesBuffer* (which motivates line 3 in the Collect-White procedure).

```
Procedure Collect-White
begin
1.      for each cand in oldCandidatesBuffer do
2.          remove cand from newCandidatesBuffer
3.          if !is-Released(cand) then
4.          // was not released during previous cycles releases
5.              cand.status := Non-Buffered
6.              if cand.color = white then // garbage cycle root
7.                  Collect(cand)
end
```

Figure 17: Collect-White

```
Procedure Collect (obj: object)
begin
1.      // the is-Released procedure should identify it as released
2.      mark obj as released
3.      // no need to check LogPointer: obj is a cyclic garbage object
4.      for each child child of obj do
5.          if !is-Released(child) then
6.              if child.color = white then
7.                  Collect(child)
8.              else
9.                  child.RC−−
10.                 if child.RC = 0 then
11.                     // This child is not part of the garbage cycle.
12.                     // The reference counting collector should free it.
13.                     RC-Free(child) //recursive deletion
14.     obj.color = black
15.     return obj to the general purpose allocator.
end
```

Figure 18: Collect

```
Procedure Process-Buffers
begin
1.      // filter candidates and change statuses
2.      for each buff which is midCandidatesBuffer do
3.          for each cand in buff do
4.              if is-Released(cand) ∨ cand.status = Newly-Buffered then
5.                  remove cand from buff
6.              else if buff is the oldest midCandidatesBuffer buffer then
7.                  cand.status := Old-Buffered
8.      for each cand in newCandidatesBuffer do
9.          if is-Released(cand) then
10.             remove cand from newCandidatesBuffer
11.         else
12.             cand.status := Mid-Buffered
13.     // Swap-Buffers-Roles
14.     tempBuffer := oldCandidatesBuffer
15.     oldCandidatesBuffer := oldest midCandidatesBuffer buffer
16.     make newCandidatesBuffer a midCandidatesBuffer
17.     newCandidatesBuffer := tempBuffer
end
```

Figure 19: Process-Buffers

Procedure Process-Buffers (Figure 19) prepares the next invocation of the cycle collection algorithm, by filtering non-relevant candidates, and preparing the buffers for the next collection. It first iterates over all the middle buffers, while rejecting candidates which have either died during current collection or were newly-buffered during it. In addition, since the oldest buffer of this buffers set, would be the oldest buffer in the next collection, the status of its candidates is modified (from mid-buffered) to old-buffered. Next, it traverses the new buffer, while rejecting candidates which have died in the last (current) collection and changing the status of

[2] Note, that the recursive deletion, i.e., the RC-Free procedure, modifies the released object status to non-buffered

the remaining candidates to mid-buffered (as *newCandidatesBuffer* would be considered as a middle buffer in the next collection). Finally, it performs a cyclic swapping between the buffers' roles.