

An Efficient On-the-Fly Cycle Collection

Harel Paz^{*1}, David F. Bacon^{**2}, Elliot K. Kolodner^{***3}, Erez Petrank^{†1}, and V. T. Rajan^{‡2}

¹ Dept. of Computer Science, Technion - Israel Institute of Technology, Haifa 32000, Israel.

² IBM T.J. Watson Research Center

³ IBM Haifa Research Lab

Abstract. A reference counting garbage collector cannot reclaim unreachable cyclic structures of objects. Therefore, reference counting collectors either use a backup tracing collector infrequently, or employ a cycle collector to reclaim cyclic structures. We propose a new *concurrent* cycle collector, i.e., one that runs concurrently with the program threads, imposing negligible pauses (of around 1ms) on a multiprocessor.

Our new collector combines the state-of-the-art cycle collector [5] with the sliding views collectors [26, 2]. The use of sliding views with cycle collection obtains two advantages. First, it drastically reduces the *number* of cycle candidates, which in turn, drastically reduces the *work* required to record and trace these candidates. This yields a large improvement in cycle collection efficiency. Second, it eliminates the theoretical termination problem that appeared in the previous concurrent cycle collector. There, a rare race may delay the reclamation of an unreachable cyclic structure forever. The proposed cycle collector guarantees reclamation of all unreachable cyclic structures.

We have implemented the proposed collector on the Jikes RVM and we provide measurements including a comparison between the use of backup tracing and the use of cycle collection with reference counting. To the best of our knowledge, such a comparison has not been reported before.

1 Introduction

Reference counting is a classical garbage collection algorithm. Systems using reference counting were implemented starting from the sixties [13]. However, reference counting garbage collectors cannot reclaim cyclic structures of objects. 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 motivated attempts to design a cycle collector [10, 12, 28]. This effort culminates in the state-of-the-art on-the-fly cycle collector of Bacon and Rajan [5].

* Email: pharel@cs.technion.ac.il.

** Email: dfb@us.ibm.com

*** Email: kolodner@il.ibm.com

† 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.

‡ Email: vtrajan@us.ibm.com

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. [36, 37, 14] and continued in a series of papers [20, 7, 8, 23, 24, 16, 15, 26, 17, 18, 4]. The advantage of an on-the-fly collector over a parallel collector and other types of concurrent collectors [6, 19, 30, 34, 11], is that it avoids the operation of stopping all the program threads. Such an operation usually increases the pause times. Today, on-the-fly collectors typically achieve pauses as short as a couple of milliseconds, and sometimes less [22].

1.2 The challenge

Bacon and Rajan [5] propose two cycle collectors. The simpler *synchronous* collector is the most efficient cycle collector known today. It runs in a stop-the-world context. Their more involved *asynchronous* collector is the only *on-the-fly* cycle collector known today. However, the asynchronous collector adds a lot of work in order to make the collection safe in the presence of concurrent program threads.

To understand why, one should note that a typical cycle collector traces cycle candidates repeatedly to discover which cycles are unreachable (Typically, each candidate structure is traced three times.) A crucial problem with repeated scanning arises when concurrent program threads modify the objects graph during the scan. This means that the collector cannot trust a scan to repeat the very same structure that a previous scan has traversed. Furthermore, as modifications occur concurrently with the scan, each specific scan cannot be guaranteed to view a consistent snapshot of the objects graph at any specific point in time. This problem that concurrency creates is the source of the two drawbacks of Bacon and Rajan's on-the-fly cycle collector: a practical drawback and a theoretical one.

The practical problem is that in order to achieve safety, the algorithm in [5] makes many repeated scans over the candidates. This reduces the overall efficiency of the reference counting collector. The theoretical problem is that completeness cannot be guaranteed (*completeness* of the algorithm here is used in the standard sense of *liveness* in distributed computing). A rare race condition may prevent an unreachable cyclic structure from being ever reclaimed.

1.3 The solution

In this work, we propose an algorithm for on-the-fly cycle collection which solves these drawbacks, by employing the sliding views techniques recently developed for concurrent garbage collection in [26, 2] and using them with the cycle collector of [5]. 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* (more efficient) algorithm of [5] 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 ([4, 26, 2]).

The theoretical completeness 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 in this view. When a cycle collection is executed on top of this sliding view, this cycle is guaranteed to be reclaimed.

Unfortunately, the solution described above does not work. The problem oddly stems from the celebrated efficiency of the sliding views reference counting collector. All previous cycle collectors require as input a list of all decrements of reference counts in order to work correctly. Missing decrements may lead to missed garbage cycles that will never be reclaimed. However, the sliding views reference counting collector does not keep track of all decrements. A large fraction of all reference counts updates are ignored by the sliding views reference counting collector and it is shown in [26] that objects may be correctly reclaimed even when only a small fraction of the reference count updates are recorded and executed. To solve this mismatch, we extend the analysis of the sliding views collector to show that the cycle collector may base its candidates on the decrements that are being recorded plus a special treatment of newly created objects. This is an interesting new property of the sliding views collector which is magically applicable to cycle collection.

From the practical point of view, the use of the simpler synchronous algorithm implies more efficient execution. Furthermore, making only a small fraction of the decrements (because of using the sliding views collector) implies recording fewer candidates for cyclic structures, which, in turn, implies less work on traversing these candidates. This yields a substantial reduction in the cycle collector work. In addition, we further improve the synchronous algorithm in [5] making it run even faster, by employing a better scheduling strategy and new filtering techniques that further reduce the number of traced objects.

Finally, in addition to testing the collector with pure reference counting, we also test it with an efficient variation of generational collection that uses reference counting (and cycle collection) on the old generation only. (We use the age-oriented collector of Paz et al. [33].) Cycle collectors spend a large fraction of their time working on cycle candidates among newly allocated objects. The age-oriented collector eliminates a large fraction of the cycles as well as a large fraction of the cycle collector's work, as it uses mark and sweep on the young objects and it runs the cycle collector on the older objects only.

1.4 Implementation, measurements, discussion

We have implemented the new cycle collector with the Levanoni-Petrunkin reference counting collector [26] and with the (more efficient) age-oriented collector of [33]. The implementation was done on the Jikes RVM [1] and compared against the original cycle collector of Bacon and Rajan [5]. We measured various features of these collectors showing that the amount of work has decreased significantly with the new cycle collector.

One of the more interesting measurements we provide is the first comparison of cycle collection to a backup tracing collector. This comparison is important since these are the main two options provided to an implementer of a reference counting algorithm. Unfortunately, there is no prior report comparing these two options. We measure the throughput of a JVM that uses the cycle collector with a JVM that uses a backup tracing collector to collect unreachable cyclic structures.

It turns out that backup tracing wins by 5-10% (application overall throughput). But, when reference counting was used on the old generation only, the new cycle collector performed equally to the backup tracing solution and even outperformed it on tight heaps. Note that in both cases we compared apples to apples: the same scenario was run once with a backup tracing and once with a cycle collector. Detailed measurements are provided in Section 5.

Discussion. At first glance, the reader may conclude that cycle collection is redundant. Backup tracing always does almost as well. However, we remind the reader that modern platforms and benchmarks also rule out reference counting as a method to reclaim garbage, as it is inferior to tracing in most cases with respect to throughput [2]. Should we give up on reference counting and cycle collection? To our minds, the answer is no. With the direction modern computing is taking, we believe that the cycle collector may become much more effective compared to a backup tracing collector. As heaps grow larger, reference counting may become the preferred method of choice. While tracing must traverse the live objects in the heap, reference counting needs only account for reference counts updates and reclaiming dead objects⁴. If future benchmarks use a large live heap or even a large old generation, then reference counting may become the best collector. When that happens, a companion cycle collector will be required. In that case, the cycle collector proposed here is an effective companion and we expect it to outperform a backup tracing collector.

The best way to use reference counting today is to run it on the old generation only as proposed in [3, 9, 33]. In that case, running cycle collection with the reference counting is the right choice.

1.5 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

⁴ Actually, when the heap is tight and collections are frequent, reference counting is already winning over tracing the whole heap [2]. But, we don't expect heaps to be tighter in the future.

new cycle collector stressing the main new ideas 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. New techniques which reduce the number of traced objects and the details of the cycle collector (including pseudo-code) are provided in our technical report ([32]) and in Appendices A and B for the reviewers.

2 Review of previous collectors

In this section, we review relevant previous work. We start by reviewing the algorithms for cycle collection [28, 27, 5] and then we review the sliding views collectors [26, 2].

In this paper the term *cycle* or *cyclic structure* refers to a strongly connected component in the objects graph. A strongly connected component is a maximal subgraph of a directed graph such that for every pair of vertices u, v in the subgraph, there exists a directed path from u to v and a directed path from v to u .

2.1 Collecting cycles on a uniprocessor

We start with the synchronous cycle collector of [5] (building on [28, 27]) that runs in a stop-the-world manner on a uniprocessor. Garbage cycles can only be created when a reference count is decremented to a non-zero value ([28, 27]). The reference counting collector records all objects whose reference count is decremented to a non-zero value. The cycle collector uses this list as a set of candidates that may belong to a garbage cycle. Three colors are used to mark the state of objects. The initial color of all objects is black. A possible member of a garbage cycle is marked gray. The white color signifies an object that is identified as part of an unreachable cycle. The cycle collector runs three traversals on all objects reachable from the candidate set as follows.

- **The mark stage:** traces the graph of objects reachable from the candidates, subtracting counts due to internal references and marking traversed nodes gray. At the end of this traversal, all nodes of each unreachable cyclic structure have zero reference counts, whereas each reachable cyclic structure has at least one node with positive reference count.
- **The scan stage:** scans the subgraph of (gray) objects reachable from the candidates. All objects reachable from external pointers (those with positive reference counts) and all their descendants are marked black. Also reference counts are restored to reflect all outgoing pointers from black objects. All other nodes in the subgraph are colored white (these objects are identified as forming a garbage cycle).
- **The collect stage:** scans the subgraph again and reclaims all white objects.

2.2 Collecting cycles on-the-fly

The first concurrent cycle collection algorithm was proposed in [5]. Their algorithm consists of two phases, each running several scans on the subgraph reachable from the candidates. In the first phase, a variant of the above synchronous algorithm is used, but instead of reclaiming the white nodes these nodes are recorded as potential unreachable

cyclic structures. Due to concurrent mutator activity, some of the white objects may have been incorrectly identified and may actually be reachable. The second phase is executed after waiting for the next (reference counting) collection and it then re-examines the potential cycles identifying which of them are indeed unreachable and reclaiming their objects.

Two disadvantages The above concurrent garbage collector has a theoretical drawback and a practical drawback. A garbage collector is called *complete* if it eventually collects all unreachable objects. The first problem of this cycle collector is that it is not complete. Rare race conditions may prevent it from collecting garbage cycles. An example appears in [5]. The second problem is practical. The algorithm traces the candidate cycles a couple of times in the second phase to ensure that no false garbage cycle is reclaimed. These extra scanings cause a substantial reduction in efficiency, especially for (typical) benchmarks which contain many garbage cycles or many false cycle candidates. Moreover, the concurrent cycle collection algorithm enforces additional overhead on the execution of the reference-counting algorithm as it must fix subgraphs that were left gray or white due to improper re-traversals.

2.3 Incorporating sliding views

Both drawbacks of the asynchronous collector in [5] stem from the fact that the concurrent cycle collector cannot rely on being able to re-trace the same graph. In this work, we propose a new concurrent cycle collector that ameliorates both problems. To that end, we exploit the 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 completeness. Before describing the new algorithm, we provide an overview of the sliding views reference counting collector.

A simple version of the Levanoni-Petrack sliding-views collector may be described by 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. If an object is not dirty, then its value in the heap is equal to its value at the snapshot time. The snapshot value of dirty objects may be obtained from the local buffers. 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 requiring synchronized operations.

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 updates 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 (short) 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 modified reference, the snooping mechanism logs the address of the object that has acquired a new reference in a local buffer. These logged addresses are considered roots for the current collection and so such objects are not reclaimed. The view of the heap used by the collector may be thought of as a view that is sliding in time: the heap objects are viewed 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

In this section we provide an overview of the new collector with its main ideas stressed. A full description including the pseudo-code is provided in Appendix B and it also appears in our technical report [32]. As mentioned, the new collector eliminates the disadvantages of the previous cycle collector yielding a non-intrusive, efficient cycle collector that guarantees completeness.

We start by describing the new cycle collection algorithm assuming two inputs. First, a snapshot of the heap. Second, a list of all objects whose reference count has been decremented to a positive value since the last cycle collection. A first observation is that given these two inputs we may apply the *synchronous* algorithm of [5] on the given snapshot and 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 cycle is a stable property, i.e., program activity cannot make an unreachable object reachable, we get an efficient identification of garbage cycles.

Next, we need to specify how to obtain the inputs efficiently. We first concentrate on the first input: the snapshot. The second input cannot be obtained efficiently, but we will find ways to use a restricted version of it.

3.1 Obtaining a snapshot (or a sliding view)

Looking at the cycle collector, the way it uses the snapshot is repeatedly traversing several subgraphs of the snapshot heap. To obtain a snapshot that can be traversed, we may use the mechanism of [26] described in Subsection 2.3. Traversing a subgraph is done as in [2]. We employ the write barrier of [26]. Then, to traverse an object according to its pointer values as existed at snapshot time we scan each object in the following manner. First, the dirty bit of the object is examined. If the object is not dirty (no pointer in the object has been modified since the snapshot was taken), then its current state in the heap is equal to its state during the snapshot and the collector may trace it by reading its pointers from the heap. Otherwise, the object has been modified since the snapshot time

and it is marked dirty. In this case, the collector traces 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, repeated traces are bound to trace the same graph each time.

In terms of completeness, this means that once a garbage cycle is created, it must exist in the next snapshot, and thus is bound to be collected by the synchronous algorithm of [5]. 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 entire second phase of the asynchronous algorithm of [5] is redundant: there is no need to *store* identified garbage cycles and validate them during the next garbage collection, and there is no need to *traverse* these cycles again in the next collection.

We now proceed to using sliding views instead of snapshots. The goal is to eliminate the need for a simultaneous halt of all program threads and to obtain extremely short pauses. The cycle collector remains the same, except that it (obviously) 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. In particular, the view may contain values of pointers that were updated after t_1 . If such a modified pointer creates a false unreachable garbage cycle in the view, then it must happen that a pointer is added to this cycle during the interval in which the sliding view is taken. 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 completeness, 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.

3.2 Obtaining the list of candidates

It remains to obtain the second input to the synchronized cycle collector of [5]. This collector described above and all previous collectors used a candidate set consisting of all newly created objects plus all objects whose reference count is reduced to a positive value by any pointer modification since the previous cycle collection. However, the

sliding views reference counting collector of Levanoni and Petrank [26] does not maintain such a list. In fact, it is oblivious to most of the pointer updates and this is what buys its efficiency. A naive solution is to add the recording of such a list to the reference counting collector of [26]. We could not accept this solution as it would undermine the efficiency of the reference counting. Instead, we analyze what is really required to collect cycles and find out that the reduced set of candidates suffices. This preserves the efficiency of the reference counting collector and also significantly improves the efficiency of the cycle collector as fewer candidates need to be recorded and less work is required to traverse their descendants.

Newly created objects Let us review one technicality that exists also in prior art. The assertion that it is enough to consider only reference count decrements as candidates is accurate but not relevant for all modern collectors. The reason is that reference counts are not updated for root pointers. Thus, all known cycle collectors use as candidates more than the set of objects whose reference count was decremented to a positive value. The set of candidates also includes all objects created since the last collection and all objects referenced directly from the roots during the previous collection.

Think, for example, of two new objects that point to one another only (forming a cycle) and a root pointer points to one of them. If the root pointer is modified, then a cycle of garbage is formed, but it is not noted from reference count decrements. The extended candidate set as above is enough to detect any such garbage cycle. We do not elaborate on this as this solution is used by all previous collectors.

Obtaining the candidates. The sliding views collector can yield almost for free a list of newly created objects and a list of objects that were referenced by the roots during the previous collection. We now concentrate on the more problematic set of objects whose reference count was decremented.

The Levanoni-Petrank reference-counting collector [26] uses a shortcut to reduce a large fraction of the reference count updates. The idea is that when a pointer p takes the values $o_0, o_1, o_2, \dots, o_n$ 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, \dots, o_{n-1} are executed might prevent noting that one of the decrements creates a new unreachable cycle.

We now claim that we are able to collect all garbage cycles, even though we record and consider many fewer objects as candidates, i.e., those supplied by the Levanoni-Petrank reference-counting collector. To be more precise, when a pointer p takes the values $o_0, o_1, o_2, \dots, o_n$ between two collections, only o_0 is considered as a candidate (if its reference count is decremented to a non-zero value) by the new cycle collector. The objects o_1, o_2, \dots, o_{n-1} that were considered by previous collectors as candidates are ignored. Additional relevant decrements are decrements that are executed by the reference counting collector itself. When an object is reclaimed, the collector decrements the reference counts of all its descendants. These decrements may also produce candidates (if the descendant's reference count is not decremented to zero).

To show that the collector does not miss a garbage cycle, we divide the argument into 2 cases: garbage cycles comprising solely of old objects and garbage cycles containing at least one young object, where a young object is an object that has been created

after the previous sliding-view (or snapshot). We show that each of these two cases is properly handled.

The easy case is when a garbage cycle includes a young object. As mentioned earlier in this section, all young objects (surviving the reference-counting collection) are considered candidates. Thus, this cycle will not be missed.

The more interesting part is to note that garbage cycles containing only old objects (those who were created before the previous sliding view) are not missed. If this cycle was reachable during the previous sliding view and is unreachable in the current sliding view, then there exists a pointer to one of the cycle's objects in the previous sliding view, but this pointer does not exist in the current sliding view. If this was a root pointer, then the cycle is considered by the fact that all root pointers from previous collection are candidates. Otherwise, this is a heap pointer that has been modified during the time interval between the two sliding views. This scenario is depicted in Figure 1. The pointer modification results either from the application modification of the pointer (as in Figure 1), or because the object containing this pointer was reclaimed and then the memory manager deleted the pointer. In the first case, the change of this pointer is logged in a local buffer causing a decrement to the reference count of the object previously referenced. In the latter case, the delete operation of the collector implies a similar reference count decrement. In each of these cases, this object becomes a candidate for cycle collection. Hence, cycles containing only old objects are accounted for properly.

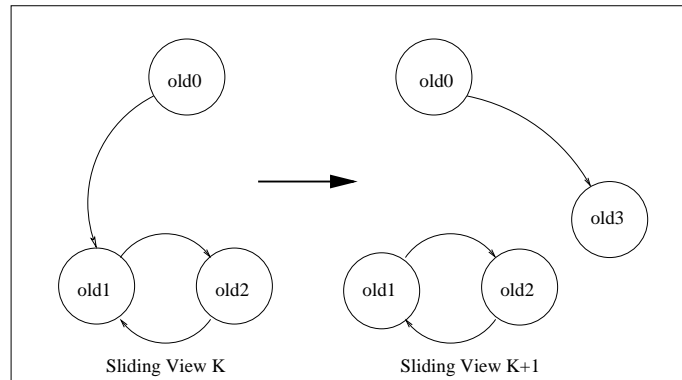


Fig. 1. A garbage cycle comprising solely of old objects is created between the K^{th} and the $K + 1^{\text{st}}$ sliding-views. In this example, the cycle was reachable from *old0* and it became unreachable because *old0* was modified. Since *old0* is modified between the sliding views, *old0* (and in particular, the pointer to *old1* in it) must be logged to a local buffer that is later used by reference counting collector. Therefore, the reference count of *old1* gets decremented in the $K + 1^{\text{st}}$ collection, and it is then considered as a candidate.

To summarize, we may employ the efficient write barrier of Levanoni-Petrack and collect cycles correctly using as candidates all objects whose reference count is decremented to a non-zero value, as well as all young objects.

3.3 Using the age-oriented collector

From the description above, it seems that newly created objects add a substantial burden on the cycle collector. Our measurements show that this is indeed correct. Therefore, we also tried to use the proposed cycle collector with a collector that runs reference counting and cycle collection on the old generation only. We chose an age-oriented collector that runs concurrent reference counting on the old generation and concurrent mark and sweep on the young objects [33]. The age-oriented collector is not a standard generational collector. It is a twist on generational collection that is adequate for concurrent collection. Using the age-oriented collector it was possible to eliminate 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. Indeed cycle collection was more effective in this setting. Let us say a few words about the age-oriented collector. For a full description see [33].

The age-oriented collector keeps generations, but it does not run frequent young generation collections. The reason is that short pauses are obtained by concurrency already and do not need to be obtained by short young collections. The heap is collected only when it gets full. When that happens, the age-oriented collector uses a reference counting collector to reclaim objects in the old generation and mark and sweep collector to reclaim objects in the young generation. Since these collections always happen together, there is no need to record inter-generational pointers. It is important to note that the age-oriented collector is an efficient collector, in particular, it is more efficient than the reference counting algorithm as a stand-alone.

3.4 Reducing the number of traced objects

New techniques which reduce the number of traced objects are provided in our technical report ([32]) and in Appendix A for the reviewers.

4 An Implementation for Java

We have implemented our algorithm in Jikes RVM (research virtual machine) [1]. 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.

More implementation details are provided in our technical report ([32]).

5 Measurements

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. We have used the SPECjvm98 benchmark suite and the SPECjbb2000 benchmark (both described in SPEC's Web site [35]). 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 [4, 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 (in Jikes) 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 collectors: the Levanoni-Petrank reference counting collector ([26]), and the more efficient age-oriented collector ([33]). 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 [4].

5.1 Performance

Our major benchmark is SPECjbb2000. SPECjbb2000 requires a 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 a 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.

Figure 2 depicts the throughput ratio between using the cycle collector and a backup tracing collector when both are used with the Levanoni-Petrank collector on a varying number of warehouses and heap sizes. Note that with 1-3 warehouses the collector has a spare processor to run on, since the platform has four processors. In this case, throughput differences occur only when the collector is not efficient enough to free enough space for program threads with on-going allocations. This is more noticeable on tight heaps. With 4-8 warehouses, the collector does not have a spare processor and

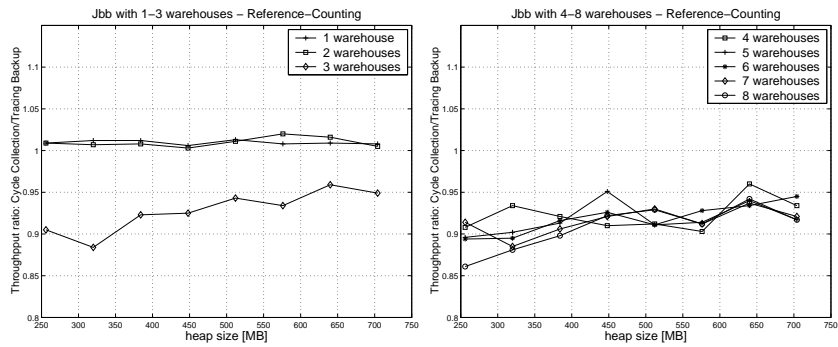


Fig. 2. SPECjbb2000 on a multiprocessor: throughput ratio for the Levanoni-Petrack collector

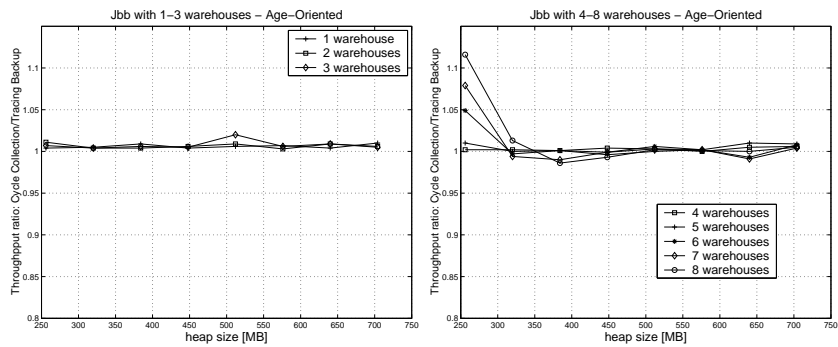


Fig. 3. SPECjbb2000 on a multiprocessor: throughput ratio for the age-oriented collector

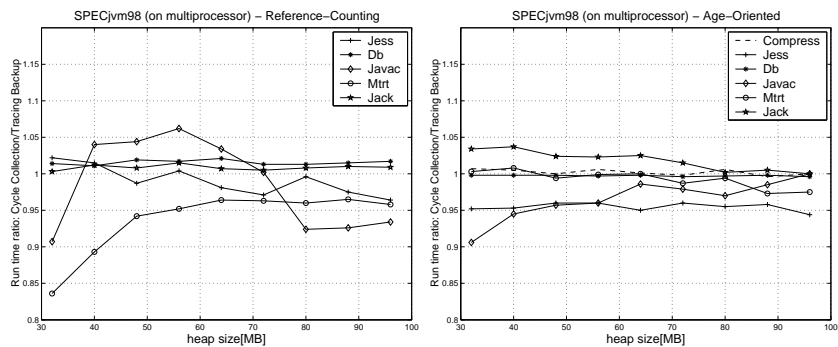


Fig. 4. SPECjvm98 on a multiprocessor: run-time ratio .

its use of CPU directly affects the throughput. The tracing backup collector outperforms the cycle collector usually by 5%-10%.

The same measurements have been run when the cycle collector and the backup tracing collector were used with the age-oriented collector [33], see Figure 3. Here, only old objects are collected with reference counting, and thus the cycle collector runs only on old candidates. In this case, there is not much difference between the two options to collect cycles, except for tight heaps. As already seen in [2] reference counting has an advantage on tight heaps over tracing. Here it is seen that cycle collection is also preferable on tracing (as an add-on to reference counting) when the heap is tight.

When running the SPECjvm98 benchmarks on a multiprocessor the collector runs concurrently with the program thread(s) on a spare processor. Figure 4 depicts the results both with the Levanoni-Petrank reference counting collector as well as with the age-oriented collector. The results do not point to a clear winner. Each application behaves somewhat differently and most of the differences are below 5%. The only clear noticeable difference, is 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 induce a large amount of work on the cycle collector. During the (single) long collection, the mutators 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.

5.2 Pause times

We have measured the maximum pause times of the Levanoni-Petrank reference counting collector accompanied by our cycle collection algorithm. The maximum pause times for the runs of the SPECjvm98 benchmarks and the SPECjbb2000 benchmark are reported in table 1. The SPECjvm98 benchmarks were run with a 64MB heap size and the SPECjbb2000 (with 1,2,3 warehouses) were run with a 256MB heap size. In these measurements, the number of program threads is smaller than the number of CPU's. Note that if the number of threads exceeds the number of processors, then large pause times appear because threads lose the CPU to other mutators or the collector. The length of such pauses depends on the operating system scheduler and is not relevant to the collector. Hence we report only settings in which the collector runs on a separate spare processor.

The maximum pause time measured for all benchmarks was 1.7 ms. The maximum pause time of the Levanoni-Petrank reference counting collector does not depend on whether it is accompanied by a tracing backup or by a cycle collector. The operation that determines the length of the pause time is the scanning of the roots of a single thread, which occurs in one of the handshakes.

5.3 Collector characteristics

Amount of cyclic garbage Table 2 provides, for each benchmark, the number of garbage cycle objects reclaimed and the space they consume. As the age-oriented collector only employs cycle collection on old objects, it needs to reclaim a smaller set of

Benchmarks	Maximum pause time (milliseconds)
compress	1.0
jess	1.3
db	0.7
javac	1.7
jack	1.0
mtrt	0.9
jbb-1	0.8
jbb-2	0.6
jbb-3	1.1

Table 1. Maximum pause time in milliseconds

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

Table 2. Cyclic garbage collected for each benchmark by our cycle collector, when incorporated with the reference counting and the age-oriented collectors.

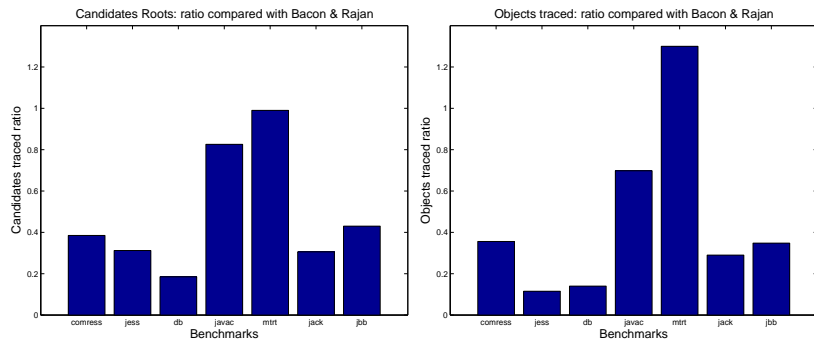


Fig. 5. Saving in the tracing work and in the number of candidates compared the collector of Bacon and Rajan.

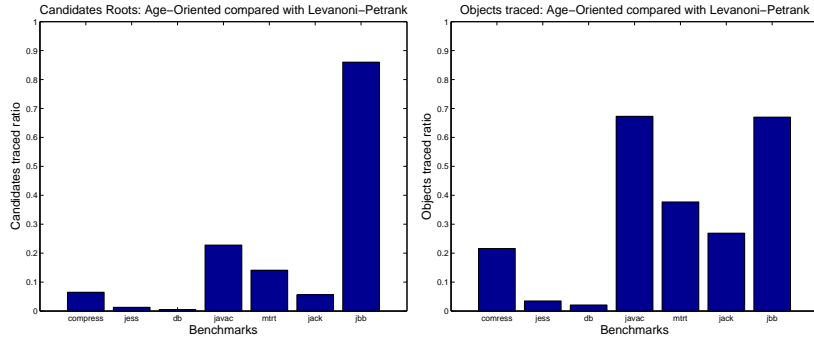


Fig. 6. Saving in the tracing work and in the number of candidates when the age-oriented collector is used compared to the reference counting collector.

garbage cycles than the reference counting collector. The cycles in the young generation are reclaimed by the tracing collector.

The only benchmarks that produce a substantial amount of space in garbage cycles are `_213_javac` and `_201_compress`. In `_201_compress`, there are some dozens of garbage cycles comprised of huge objects, and it hence requires only a small amount of tracing. The benchmark `_213_javac` however, contains thousands of garbage cycles, thus requiring a large amount of work of the cycle collector.

Amount of tracing To check that the proposed collector indeed traces much fewer objects than previous collectors, we compare it to the cycle collector of Bacon and Rajan [5]. We report the ratios of the *candidates examined* and the ratio of *objects traced* when compared to those of [5]. To be extremely conservative we did not include the objects scanned during the additional verification phase of [5]. Thus, the actual advantage of the new collector is even higher than reported. We did not count the additional phase since in this additional phase some of the objects were not actually *traced*. For these objects the actual operation only included work on their colors.

In the graphs presented in Figure 5, the lower the ratio, the better the new algorithm behaves, and any ratio smaller than 1 implies that it traced fewer candidates and objects. To make the comparison fair, the new collector was measured only with the reference counting collector. Figure 5 shows that the new cycle collector trace fewer candidates compared to the previous cycle collector (of [5]) over all benchmarks. It is usually also traces substantially fewer objects except for one case: the `_227_mtrt` benchmark (which was discussed above).

In Figure 6, we report the additional saving when the cycle collector is used with the age-oriented collector (on the old generation only). As can be seen, the further reduction in tracing is substantial for most benchmarks.

6 Related work

The inability of reference-counting to reclaim cyclic garbage structures was first noticed by McBeth [29]. The algorithm of Martinez et al. [28] (inspired by [12]) 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. Lins [27] postponed 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 traversed the buffer at a suitable point. Bacon et al. [5] 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 Conclusion

We presented a new cycle collector adequate for use with a reference counting garbage collector. The new cycle collector runs concurrently with the program threads, achieving negligibly short pauses of less than 2ms. It uses the sliding views reference counting collector of Levanoni and Petrank [26] with the synchronous efficient cycle collector of Bacon and Rajan [5]. These algorithms do not fit together since the original cycle collector expects to get a list of all reference count decrements, whereas the original reference counting collector is oblivious to most of these decrements. However, we provide a finer analysis of cycle collection showing that the information gathered by the reference counting collector is enough to guarantee reclamation of all unreachable cycles.

Building on the sliding views mechanism obtains a drastic improvement in efficiency. Much of the work required to ensure concurrent correctness may be eliminated. We also add filtering techniques to further optimize the collector's performance. An additional theoretical contribution is the completeness of the collector. The resulting cycle collector is guaranteed to reclaim all garbage cycles, whereas the only available previously known concurrent collector [5] had an (extremely rare) sequence of events that prevents it from collecting an unreachable cyclic structure forever.

We implemented the proposed cycle collector and we provide the first direct comparison of running a cycle collector with reference counting against running reference counting with a backup tracing collector. Our results show that with contemporary benchmarks, the backup tracing collector outperforms the cycle collector, even though it is the most efficient cycle collector available today. However, we also measured the cycle collector when the reference counting was run only on objects in the old generation. In this case, the cycle collector performed equally to the backup tracing collector, and even better on tight heaps. This means that on today's platforms and benchmarks cycle collection is effective when applied to the old generation only. In the future, if heaps and live data become much larger, then the techniques described in this work may become a most effective method to reclaim garbage.

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.

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. Hezi Azatchi and Erez Petrank. Integrating generations with advanced reference counting garbage collectors. In *Proceedings of the Compiler Construction: 12th International Conference on Compiler Construction, CC 2003*, volume 2622 of *Lecture Notes in Computer Science*, pages 185–199, Warsaw, Poland, May 2003. Springer-Verlag Heidelberg.
4. 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.
5. 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.
6. 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.
7. 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.
8. Mordechai Ben-Ari. Algorithms for on-the-fly garbage collection. *ACM Transactions on Programming Languages and Systems*, 6(3):333–344, July 1984.
9. Stephen M. Blackburn and Kathryn S. McKinley. Ulterior reference counting: Fast garbage collection without a long wait. In *OOPSLA* [31].
10. Daniel G. Bobrow. Managing re-entrant structures using reference counts. *ACM Transactions on Programming Languages and Systems*, 2(3):269–273, July 1980.
11. Hans-Juergen Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. *ACM SIGPLAN Notices*, 26(6):157–164, 1991.
12. T. W. Christopher. Reference count garbage collection. *Software Practice and Experience*, 14(6):503–507, June 1984.
13. George E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 3(12):655–657, December 1960.
14. Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *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, Portland, OR, January 1994. ACM Press.

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 L. Hudson and J. Eliot B. Moss. Sapphire: Copying GC without stopping the world. In *Joint ACM Java Grande — ISCOPE 2001 Conference*, Stanford University, CA, June 2001.
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. Wachenchauser, 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, David F. Bacon, Elliot K. Kolodner, Erez Petrank, and V.T. Rajan. Efficient on-the-fly cycle collection. Technical Report CS–2003–10, Technion, 2003.
33. Harel Paz, Erez Petrank, and Stephen M. Blackburn. 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>.
34. Tony Printezis and David Detlefs. A generational mostly-concurrent garbage collector. In Hosking [21].

35. SPEC Benchmarks. Standard Performance Evaluation Corporation. <http://www.spec.org/>, 1998,2000.
36. Guy L. Steele. Multiprocessing compactifying garbage collection. *Communications of the ACM*, 18(9):495–508, September 1975.
37. Guy L. Steele. Corrigendum: Multiprocessing compactifying garbage collection. *Communications of the ACM*, 19(6):354, June 1976.

A Reducing the number of traced objects

Next, several methods are proposed to further reduce both the number of *candidates*, and the number of *objects traced* during a candidate traversal. First, we make use of the strategy of Bacon and Rajan [5] to reduce tracing. They proposed to ignore acyclic objects (objects that cannot be a part of a cycle, e.g., an array of scalars). Such objects are statically determined and are never considered as candidates. Any acyclic object reached during the algorithm traversals is ignored. Additional elimination strategies are proposed next.

Examining only mature candidates. The new collector runs cycle collection with each garbage collection. However, it lets the candidates “mature” before actually testing them for membership in an unreachable cyclic structure. While candidates “wait” to be examined, many of them are removed from the candidate list (as explained below). The collector examines only the candidates which were accumulated k collections ago and were not filtered. Technically, instead of having one large candidates buffer, we employ $k + 1$ smaller candidates buffers, each containing candidates accumulated in different collections. In the current cycle collection we use the oldest buffer. One reason for removal from the candidate list is that objects are simply reclaimed by the reference counting collector before reaching the oldest buffer. A second filtering technique employed is to remove any object, that is added to the most recent candidates buffer, from any older buffer it appears in. This means that an object appears only in one buffer and that if its reference count is decremented several times, it will be examined only once (if not reclaimed earlier). (The removal from older lists is executed in a short processing of the buffers in the end of each collection.)

The $k + 1$ buffers method allows a structured control over candidates maturity and allows tracing only candidates that have not been filtered out (and have not died) throughout the last k collections. Previous collectors run the cycle collector each k collections on all candidates. Thus, they handle candidates that were recently discovered and have not yet matured.

Ignoring known to be alive objects. Next, we try to reduce the number of candidates and also the amount of tracing executed. Several objects are known to be alive in the beginning of a collection. These include objects that were directly reachable from the roots, objects that were snooped and objects that were marked dirty because they were modified after the sliding view was taken. Many such objects are traced by the cycle collector. The proposed collector treats these objects differently. First, if the cycle collector reaches one of these objects during the first (mark) stage, it ignores it. It does

not decrement its reference count, nor does it trace its subgraph. By avoiding the explorations of those objects, we do not loose any garbage cycles, since if an object is known (or assumed) to be alive, so do all the objects reachable from it.

Saving doubly scan stage. To save more scanning time, we add an additional stage between the mark stage and the scan stage. It is more efficient to preprocess the list of objects that are known to be alive and trace them to mark their subgraph black. If we do not do that, the scan stage may sometimes color white an object together with its subgraph, only to find out later that this object was referenced from another (gray) externally referenced object. In this situation, the object (and subgraph) would be colored black only during a second traversal of the same subgraph. Such repeated traversals are saved by the new stage.

Not all objects that are known to be alive, can be identified when collection commences. In particular, an object may be modified (and thus become dirty) by a mutator after being traced during the mark stage. The subgraph of such an object may be traced and colored gray (during the mark stage) and only later it becomes evident that the tracing was redundant. Thus, we also add a check to the second (scan) stage. When reaching a gray object in the scan stage, we check whether it is dirty. If it is, its subgraph is colored black immediately (as it is reachable from an object which was alive when the sliding-view was taken).

One disadvantage of using dirty objects for the above filtering methods, is that we must use the additional *CRC* (cyclic reference count) field as in the asynchronous cycle collector of Bacon and Rajan, in order to correctly restore the "real" reference counts. The reason we must use it is that the set of identified live objects (actually, only the subset of dirty objects) is not fixed during the collection. Recall that we do not scan this set of objects, implying that we also do not decrement their reference counts for internal references. However, since objects become dirty concurrently, we may decrement their reference counts several times before noting that they are alive. It is not possible later to tell how many times decrements were applied on dirty objects (before they became dirty) and thus it is not possible to restore the original reference counts. This necessitates the use of the *CRC* field. One must decide between using the dirty bit to identify reachable objects and avoiding using the *CRC* field. We chose to consider dirty objects and use the *CRC* field.

B The Garbage Collector Details

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

B.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 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 object's 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].

B.2 General issues

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*.

```

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

```

Fig. 7. Mutator code: Update Operation

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.

B.3 Interface with the reference counting collector

```

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

```

Fig. 8. Add-Candidate

Procedure Add-Candidate (Figure 8) is called by the reference counting collector in order to insert an object onto the *newCandidatesBuffer*. If this object is not already

buffered in the *newCandidatesBuffer*, and it is not acyclic, 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.

B.4 Cycle algorithm code

```

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

```

Fig. 9. Process-Cycles

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 stages:

- **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).
- **Scan black stage:** colors black the objects that the mark stage has considered as alive. Its purpose is to save redundant traversals as described in A.
- **Scan stage:** scans the subgraph of relevant candidates, and re-colors black objects which are reachable from external pointers. All other nodes in the subgraph are colored white.
- **Collect stage:** scans the white subgraphs 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 ⁵ and those that

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

```

Fig. 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 // reached during the recursion
6.       obj.CRC --
7.     // check whether object was witnessed living
8.     if obj.LogPointer != NULL  $\vee$  obj  $\in$  Roots
9.        $\vee$  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

```

Fig. 11. Mark

```

Procedure Read-Sliding-View(obj: 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

```

Fig. 12. Read-Sliding-View

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 candidates.

The Mark procedure performs a depth-first traversal over the candidates' sliding-view subgraph. 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-views 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 (as it was initialized before), but is decremented. Objects that are alive are pushed onto 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 *Log-Pointer*. 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* were all pushed during the mark stage.

```

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

```

Fig. 13. Scan-Black-Live-Stack

```

Procedure Scan-Black(obj: 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

```

Fig. 14. Scan-Black

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

```

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

```

Fig. 15. Scan-Candidates

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 are 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 cycles' 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

```

Procedure Scan (obj: object)
begin
1.   if obj.color = gray then
2.       if obj.CRC = 0 then // currently no evidence
3.           // 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 subgraph as alive
10.          Scan-Black(obj)
end

```

Fig. 16. Scan

```

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

```

Fig. 17. Collect-White

```

Procedure Collect (obj: object)
begin
1.   mark obj as released // so that the is-Released
2.   // procedure should identify it 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.              // child is not part of the garbage cycle.
12.              // The RC collector should free it.
13.              RC-Free(child) //recursive deletion
14.  obj.color = black
15.  return obj to the general purpose allocator.
end

```

Fig. 18. Collect

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) ⁶. Such recursive deletion can end-up reclaiming black candidates buffered in *oldCandidatesBuffer* (which motivates line 3 in the **Collect-White** procedure).

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

⁶ Note, that the recursive deletion, i.e., the **RC-Free** procedure, modifies the released object status to non-buffered

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*)
 \vee *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

Fig. 19. Process-Buffers

(current) collection and changing the status of 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.