

# On-the-Fly Cycle Collection Revisited

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 collectors 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.

---

\*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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...\$5.00.

## 1. INTRODUCTION

Reference counting is a classical garbage collection algorithm. Systems using reference counting were implemented starting from the sixties ([11]). However, reference counting garbage collectors cannot reclaim cyclic structures of objects (as first noticed by McBeth [28]). 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 [8, 10]. The most popular algorithm for cycle collection was proposed by Martinez et al. [27]. 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 design 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. [34, 35, 13] and continued in a series of papers [19, 6, 7, 22, 23, 15, 14, 25, 16, 17, 3]. The advantage of an on-the-fly collector over a parallel collector and other types of concurrent collectors [5, 18, 29, 32, 9], 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 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, which 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.

### 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, 25, 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 [25]. Furthermore, if we use a sliding view instead of a snapshot (as in [25]) 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, 25, 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 age-oriented collector of Paz and Petrank [31]. 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 addition to the Levanoni-Petrank reference counting collector [25] and to the age-oriented collector of [31]. 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 features 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, 25]).

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 run times 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 [33].

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 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 4.

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 [21].

### 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 new cycle collector is provided in Section 3. Implementation and results are given in Section 4. Related work is discussed in Section 5 and we conclude in Section 6.

## 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 [27, 26, 4], we then review the previous concurrent cycle collector [4]. Finally, we explain ideas from the sliding views reference counting and tracing garbage col-

lectors [25, 2] that aid in overcoming the disadvantages in state-of-the-art cycle collectors.

## 2.1 Collecting cycles on a uniprocessor

The algorithm of [27, 26] 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 [27, 26] 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 marked white when 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).

## 2.2 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  $CRC$ , leaving 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 nodes. 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 more scanning of objects and 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 object 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 [25, 2]. The idea is to use a snapshot of the heap or a sliding-view of the heap ([25]). 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 copied 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 [25] 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 [25] 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 [25].

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 [25].

## 3. CYCLE COLLECTOR OVERVIEW

In this section, we describe the new cycle collector. The collector details are omitted for lack of space. The full details and more measurements appear in our technical report [30].

As mentioned above, adding the sliding views techniques from [25, 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 reach-

able, 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 [25]. 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, 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.

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 [24, 25]. 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  $o_0, o_1, o_2, \dots, o_n$  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.

It turns out that for all old objects (those who were created before the last sliding view) there is no problem. The reason is that an unreachable cyclic structure cannot be created by assigning a pointer  $p$  one of its objects and then changing  $p$ . If this cycle was reachable during the previous sliding view and is unreachable now, then a pointer 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.

However, young objects are more of a problem. We may fail to notice (and to collect) a candidate cycle comprising solely of young objects (objects created since the last collection). Consider the next scenario occurring between two garbage collections: Two new objects are created and their pointers form a cycle. Note that changing these pointers does not create any logging operation. New objects are known to point to nulls in the previous collection and are not logged. Next, an old object is modified to point to one of them and local pointers to these objects are erased. This updated may cause the old object's previous descendants to be logged in the mutation log (as  $o_0$  in the notation above). Next, the pointer is overwritten, leaving this cycle unreachable. However, this second update of this object is not logged. Thus, this two objects have reference count one, but they are not noticed for the cycle collection, since the decrement was unnoticed.

To solve this problem, we consider all young objects as candidates for a garbage cycle. Doing that covers all possible cycles that contain a young object. As explained above, cycles containing only old objects are accounted for properly. Thus all garbage cycles are properly noticed. 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.

To summarize, using the Levanoni-Petrack 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 [31]. 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. The idea is 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 perform 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 portions 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 scans 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 (live) object is reached during the scan phase (i.e., it was modified after it was colored gray), then this object is colored black (if not black already) 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.

## 4. IMPLEMENTATION AND RESULTS

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.

**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 [33]. 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, 16]), 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 ([25]), and the age-oriented collector ([31]). 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].

### 4.1 Server performance

Our major benchmark is the SPECjbb2000 benchmark. SPECjbb2000 requires multi-phased run with an increasing

number of warehouses. Each phase lasts for two minutes with a ramp-up period of half a minute before each phase. 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 the SPECjbb2000 benchmark produces almost no garbage cycles (only some dozens of objects turn to be cyclic garbage). Hence, a collector gets only minor benefit from applying our algorithm (over the SPECjbb2000 benchmark), while suffering from the overhead it produces in each collection. Note, that 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. Thus, SPECjbb2000 can serve as measuring the overhead our algorithm produces when applied on a program which does not produce any cycles.

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 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. Note, nevertheless, that we gave the collector (in this case) 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. 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

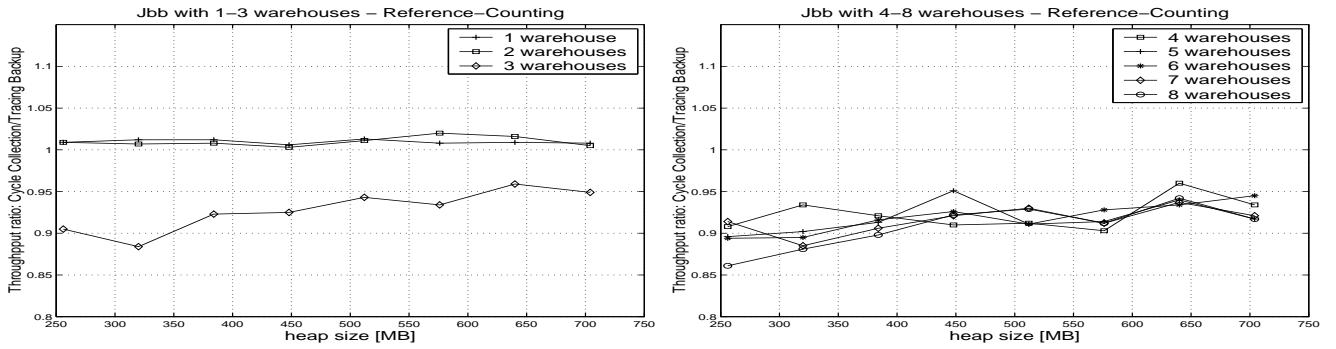


Figure 1: SPECjbb2000 on a multiprocessor: throughput ratio for the Levanoni-Petrank collector

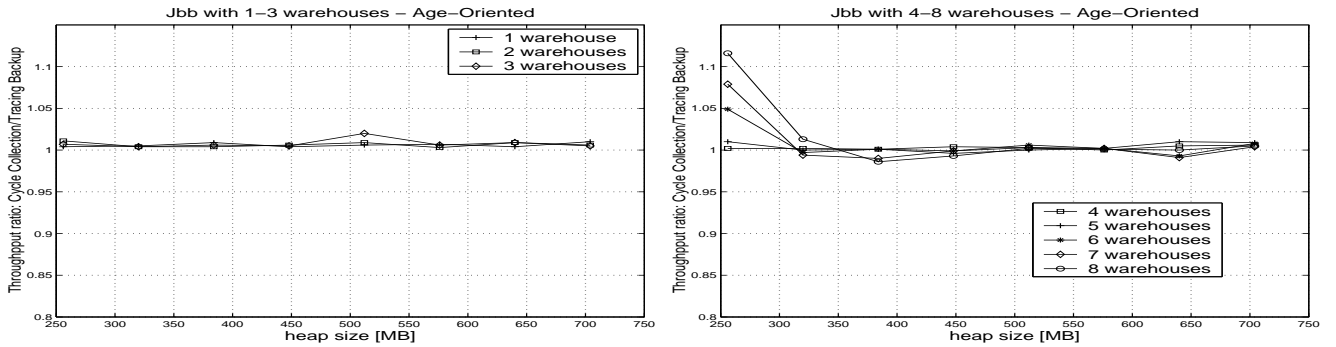


Figure 2: SPECjbb2000 on a multiprocessor: throughput ratio for the age-oriented collector

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.

More measurements appear in our Technical Report ([30]).

## 4.2 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 and the results appear in figure 3. 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.

## 4.3 Collector characteristics

### 4.3.1 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 to 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 object. Our new algorithm does not run this extra processing, and so, the saving of the new algorithm is even better than what is conservatively measured in these figures.

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 cycle 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 of the number of traced candidates and objects are introduced in figure 4. One may see that all 4 configu-

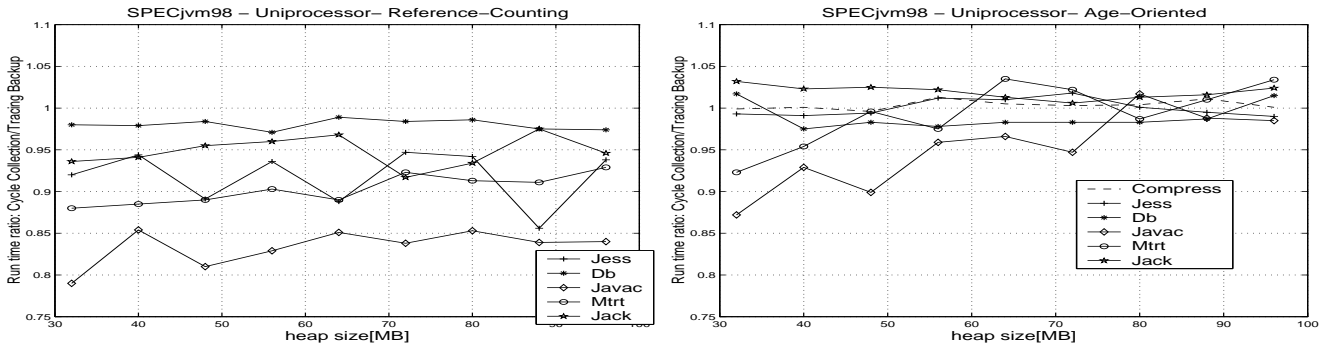


Figure 3: SPECjvm98 on a uniprocessor: throughput ratio.

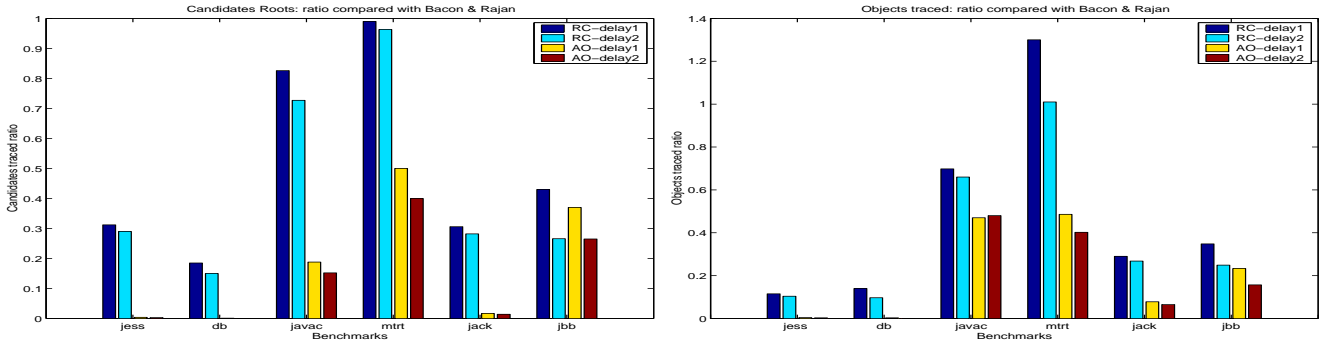


Figure 4: SPECjbb2000: saving in tracing work and number of candidates.

rations 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.

When using three buffers the algorithm traces less than 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 those 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. RELATED WORK

The traditional method of reference counting applicable in the realm of uniprocessing was first developed for Lisp by Collins [11]. 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 [36] 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 [12] 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 [25] 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 [28]. Christopher [10] 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. [27] 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 [26] extended this algorithm by post-

poning 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.

## 6. 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 [25, 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, 25]. 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.

## 7. 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.

## 8. 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'03 ACM Conference on Object-Oriented Systems, Languages and Applications*, ACM SIGPLAN Notices, Anaheim, CA, November 2003. ACM Press.
- [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] Daniel G. Bobrow. Managing re-entrant structures using reference counts. *ACM Transactions on Programming Languages and Systems*, 2(3):269–273, July 1980.
- [9] Hans-Juergen Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. *ACM SIGPLAN Notices*, 26(6):157–164, 1991.
- [10] T. W. Christopher. Reference count garbage collection. *Software Practice and Experience*, 14(6):503–507, June 1984.
- [11] George E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 3(12):655–657, December 1960.
- [12] L. Peter Deutsch and Daniel G. Bobrow. An efficient incremental automatic garbage collector. *Communications of the ACM*, 19(9):522–526, September 1976.
- [13] 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.
- [14] 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.
- [15] 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.
- [16] 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.
- [17] 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 [20].
- [18] 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.
- [19] David Gries. An exercise in proving parallel programs correct. *Communications of the ACM*, 20(12):921–930, December 1977.
- [20] 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.
- [21] Richard E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, July 1996. With a chapter on Distributed Garbage Collection by R. Lins.
- [22] 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.
- [23] 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.
- [24] Yossi Levanoni and Erez Petrank. A scalable reference counting garbage collector. Technical Report CS–0967, Technion — Israel Institute of Technology, Haifa, Israel, November 1999.
- [25] 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.
- [26] 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.
  - [27] A. D. Martinez, R. Wachenchauser, and Rafael D. Lins. Cyclic reference counting with local mark-scan. *Information Processing Letters*, 34:31–35, 1990.
  - [28] J. Harold McBeth. On the reference counter method. *Communications of the ACM*, 6(9):575, September 1963.
  - [29] 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.
  - [30] Harel Paz, David F. Bacon, Elliot K. Kolodner, Erez Petrank, and V. T. Rajan. On-the-fly cycle collection revisited. Technical Report CS-2003-10, Technion, Israel Institute of Technology, November 2003.
  - [31] Harel Paz and Erez Petrank. Age-oriented garbage collection. Technical Report CS-2003-08, Technion, Israel Institute of Technology, October 2003.
  - [32] Tony Printezis and David Detlefs. A generational mostly-concurrent garbage collector. In Hosking [20].
  - [33] SPEC Benchmarks. Standard Performance Evaluation Corporation. <http://www.spec.org/>, 1998,2000.
  - [34] Guy L. Steele. Multiprocessing compactifying garbage collection. *Communications of the ACM*, 18(9):495–508, September 1975.
  - [35] Guy L. Steele. Corrigendum: Multiprocessing compactifying garbage collection. *Communications of the ACM*, 19(6):354, June 1976.
  - [36] J. Weizenbaum. Symmetric list processor. *Communications of the ACM*, 6(9):524–544, September 1963.