An Efficient On-the-Fly Cycle Collection

Harel Paz[1], Erez Petrank[1,⋆], David F. Bacon[2], Elliot K. Kolodner[3], and V. T. Rajan[2]

[1]Dept. of Computer Science, Technion - Israel Institute of Technology, Haifa 32000, Israel
{erez, pharel}@cs.technion.ac.il
[2]IBM T.J. Watson Research Center
{dfb, vtrajan}@us.ibm.com
[3]IBM Haifa Research Lab
{kolodner}@il.ibm.com

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 [20, 2]. The use of sliding views for cycle collection yields 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. Therefore, a large improvement in cycle collection efficiency is obtained. 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 sliding-views cycle collector guarantees reclamation of all unreachable cyclic structures.

The proposed collector was implemented 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 [11]. 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 unreachable 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 [8, 10, 23]. This effort culminated in the state-of-the-art on-the-fly cycle collector of Bacon and Rajan [5].

⋆ Research supported by the Bar-Nir Bergreen Software Technology Center of Excellence and by the IBM Faculty Partnership Award.

R. Bodik (Ed.): CC 2005, LNCS 3443, pp. 156–171, 2005.
© Springer-Verlag Berlin Heidelberg 2005

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 are usually 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. [30, 31, 12] and continued in a series of papers culminating in [13, 4, 17, 20, 2]. The advantage of an on-the-fly collector over a parallel collector and other types of concurrent collectors [6, 28, 9, 14, 15, 18] 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 achieve pauses as short as a couple of milliseconds, and sometimes less [17].

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 *concurrent* cycle collector known today.

A typical stop-the-world cycle collector traces cycle candidates two or three times to discover which cycles are unreachable. A concurrent cycle collector must deal with concurrent program threads that modify the objects graph during the scan. Thus, a concurrent 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 concurrency problem is the source of the two drawbacks of Bacon and Rajan's on-the-fly cycle collector. A practical drawback is the reduced efficiency: the asynchronous collector employs additional checks (which add substantial additional work) in order to make the collection safe in the presence of concurrent program threads. A theoretical drawback is that completeness cannot be guaranteed[4]. A rare race condition may prevent an unreachable cyclic structure from being ever reclaimed.

1.3 The Solution

We present an on-the-fly cycle collector which solves these drawbacks, by employing the sliding-views techniques [20]. The idea is to obtain a fixed view of the heap (via the sliding-views mechanism), and then run the more efficient *synchronous* (i.e., stop-the-world) cycle collector of [5] on this obtained view. The theoretical completeness problem is immediately solved. Any unreachable cyclic structure generated before the view of

[4] *Completeness* of a concurrent garbage collector stands for the standard *liveness* term in distributed computing. A collector is complete if all unreachable objects are eventually reclaimed.

the heap is created can be identified in the view and reclaimed. From the practical point of view, the use of the simpler and more efficient synchronous algorithm implies a more efficient execution.

But there are more efficiency benefits. All previous cycle collectors required as input a list of all reference-count decrements, in order to reclaim all garbage cycles correctly. However, the sliding-views reference-counting collector keeps track of only a small fraction of reference-count updates (and in particular decrements). This problem is solved by improving the analysis of the cycle collector to show that the small number of decrements recorded by the sliding-views mechanism suffices to reclaim all garbage cycles. Note, that fewer decrements implies recording fewer candidates for cyclic structures, which, in turn, implies less work on traversing these candidates and a reduction in the cycle collector work. Finally, we improve the efficiency of the synchronous algorithm of [5] by employing a better scheduling strategy and new filtering techniques that further reduce the number of traced objects.

In order to check the behavior of the cycle collector in a different environment, we also incorporated it into the age-oriented collector [27]. The age-oriented collector is an efficient variation of generational collection that uses reference counting to collect the old generation and tracing to collect the young generation. 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 only on the older objects.

Organization. An overview of previous cycle collectors and the sliding-views collectors is presented in Section 2. An overview of the new cycle collector appears in Section 3. Results are given in Section 4. Related work is discussed in Section 5 and we conclude in Section 6.

2 Review of Previous Collectors

This section reviews relevant previous work. We start by reviewing the algorithms for cycle collection [23, 21, 5] and then we review the sliding-views collectors [20, 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 in the Stop-the-World Setting

We start with the synchronous cycle collector of [5] (building on [23, 21]) that runs in a stop-the-world manner on a single thread. Garbage cycles can only be created when a reference count is decremented to a non-zero value ([23, 21]). 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 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 (objects 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 on-the-fly cycle collection algorithm of [5] consists of two phases. 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 only at the next (reference-counting) collection. The potential unreachable cycles are re-examined and those found still unreachable are reclaimed.

This 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 (practical) problem is that 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 scannings cause a substantial reduction in efficiency, especially for (typical) benchmarks which contain many garbage cycles or many false cycle candidates. Moreover, additional work is required to fix subgraphs that were not recolored black on time due to improper re-traversals.

2.3 The Sliding-Views Reference-Counting Collector

A simple version of the Levanoni-Petrank sliding-views collector is one that allows stopping all program threads (mutators) simultaneously in the beginning of the collection. 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 thread-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 simultaneous halt. If an object is not dirty, then its value in

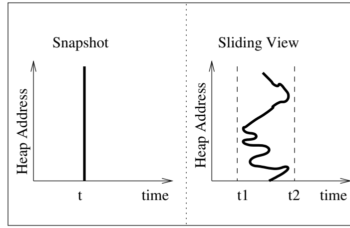

Fig. 1. A snapshot view at time t vs. a sliding view at interval [t1,t2]

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 [20] allowing the above write barrier to operate on concurrent threads without requiring synchronized operations.

The collector in [20] eliminates many of the reference-count updates by updating the reference counts according to the change in pointer values between the previous snapshot to the current one. Consider a pointer slot that, between two garbage collections is assigned the values $o_0, o_1, o_2, \ldots, o_n$. All previous reference-counting collectors execute $2n$ reference-count updates for these assignments: RC(o_0)−−, RC(o_1)++, RC(o_1)−−, RC(o_2)++, ..., RC(o_n)++. However, it is observed that only two are required: RC(o_0)−− and RC(o_n)++, which buys a substantial reduction in the number of required updates. The "o_0" value of a modified slot (previous snapshot value) is exactly the value recorded by the write-barrier when the slot is modified. The "o_n" value of a modified slot (current snapshot value) is obtained according to the dirty flag, as explained above. Note, that for pointers in newly created objects the previous referent o_0 is always null. However, the reference count of current child (o_n) of newly created objects should be incremented.

The algorithm described so far probably obtains short pause times, but in order to get even shorter pause times, the sliding-views mechanism is proposed. Here, the program threads are not halted simultaneously, but one at a time. The obtained view of the heap is not a snapshot but a sliding view. A snapshot of the heap at time t is a copy of the content of each object in the heap at time t. A sliding view of the heap is associated with a time interval [t1,t2] (rather than a single point in time). It provides the content of each object in the heap at an arbitrary time t, satisfying $t_1 \leq t \leq t_2$. In contrast to a snapshot, objects are not all recorded at the same time. Figure 1 depicts the difference between a sliding view and a snapshot.

As a snapshot view cannot be assumed anymore, correctness considerations dictate a *snooping* mechanism. During the (short) time interval $[t_1, t_2]$ in which the program threads 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 records in a local buffer the address of the object that has acquired a new reference. 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 [20].

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 our technical report [26].

We first observe that if we were given a snapshot of the heap with all reference counts updated and a list of all objects whose reference counts have been decremented to a positive value since the last cycle collection, then we would have been able to run the *synchronous* algorithm of [5] on the given snapshot and correctly identify the garbage cycles in the heap as viewed at the snapshot. This is good news because being a garbage cycle is a stable property and such a cycle remains unreachable, no matter how the application behaves, until the collector reclaims its objects. Thus, unreachable cycles can be reclaimed based on a snapshot and a list of decrements.

Next, we explain how we obtain the snapshot and the list of decrements efficiently. We first concentrate on the first issue: obtaining the snapshot. The full list of reference-count decrements cannot be obtained efficiently, but we will show that it is possible to use a partial list and how that partial list can be obtained efficiently.

3.1 Obtaining a Snapshot (or a Sliding View)

The cycle collector uses the heap (or a snapshot of it) by repeatedly traversing several subgraphs of it. To obtain a virtual snapshot of the heap that may be used for such traversals, we use the mechanism of [20] described in Subsection 2.3. Traversing a subgraph is done as follows (following [2]). The write barrier of [20] is employed by the program threads. To traverse an object according to its pointer values at snapshot time, we obtain these values 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 finds its snapshot values in the threads local buffers. After obtaining the snapshot values, objects can be traced according to their state at the snapshot time, and thus, repeated traces are bound to trace the same graph repeatedly.

In terms of completeness, once a garbage cycle is created, it must exist in the next snapshot, and thus it is bound to be reclaimed by the synchronous algorithm of [5]. We also improve efficiency, since we can use the efficient synchronous algorithm of [5] instead of using their less efficient concurrent collector. Inefficiencies originating from the need to insure correctness in spite of program-collector races are eliminated. For example, the entire second phase of the asynchronous algorithm of [5] is redundant: there is no need to *store* identified garbage cycles and there is no need to re-examine them during the next garbage collection by more traversals.

We now extend the discussion to using sliding views instead of snapshots in order to obtain an on-the-fly collector. The on-the-fly collector does not halt all program threads simultaneously, but stops each of them separately to obtain their roots and read their buffers. This creates a sliding view of the heap associated with a short time interval $[t_1, t_2]$ instead of a snapshot.

The cycle collector remains the same, except that it (obliviously) uses a sliding view of the heap rather than a snapshot. A sliding view may incorrectly indicate that an object is unreachable because the view does not represent the heap at a consistent point in time. The snooping mechanism makes sure that such objects are not reclaimed, ensuring the safety property. The snooping mechanism is explained in [20]. Let us review it shortly.

How can objects be seen unreachable in the sliding view while they are actually reachable at all times? Suppose the sliding view is read during the interval $[t_1, t_2]$. If no pointer is written to the heap during this time, the sliding view represents a snapshot of the pointers in the heap taken at the time t_2. However, as pointers are being written in the heap, this snapshot gets distorted, and the view may contain values of pointers that were updated between t_1 and t_2. It can be shown that if such a modified pointer creates a false unreachable garbage cycle in the view, then a pointer must have been written pointing to an object in this cycle during the time interval $[t_1, t_2]$. The snooping mechanism records all objects that acquire a new reference. Thus, the object that falsely seems unreachable in the sliding view must be snooped. Snooped objects are considered roots, and therefore, cyclic structures containing snooped objects cannot be reclaimed.

With respect to completeness, any unreachable cyclic structure formed before the collection begins, must be collected. The reason is that these objects are not modified during the time interval $[t_1, t_2]$ 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_2. Thus, such an unreachable cyclic structure must be reclaimed.

3.2 Obtaining the List of Candidates

It remains to explain how the list of objects whose reference counts was decremented is obtained. All cycle collectors use 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 [20] does not maintain such a list. In fact, it is oblivious to most of the pointer updates and this obliviousness is what buys its efficiency. A naive solution is to make the reference-counting collector record all the extra required updates. This solution is unacceptable as it undermine the efficiency of the reference-counting collector. Instead, we improve the analysis of the cycle collector and show that the reduced set of candidates obtained from the Levanoni-Petrank collector suffices. This way, we can preserve 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. Taking only reference-count decrements as candidates is not enough when the write barrier is not used with the roots. This is the case with all modern collectors, as a write barrier on the roots is too costly. Since decrements of roots are not accounted for, cycle collectors also include in the set of candidates all objects created since the last collection and all objects referenced directly from the roots during the previous collection.

To see that this is indeed required for all modern collectors, consider two newly created objects that point to each other only (forming a cycle) and a root pointer that references one of them. If the root pointer is modified, then a cycle of garbage is formed, but it cannot be noticed from reference-count decrements. The extended candidate set as above is enough to detect any such garbage cycle.

Obtaining the candidates. The sliding-views reference-counting collector yields 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 finding the more problematic set of objects whose reference counts were decremented.

As the sliding-views collector reduces a large fraction of the reference-count updates, we now claim that it is possible to collect all garbage cycles, even though we record and consider much fewer objects as candidates. To be more precise, when a pointer p takes the values $o_0, o_1, o_2, \ldots, o_n$ 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. Previous collectors considered also the objects o_1, o_2, $\ldots$, o_{n-1} as candidates but are ignored by us. Additional relevant decrements are treated by this collector in the same manner as previous collectors. These 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). Both cases are properly handled.

The easy case is when an unreachable 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 involved case is a garbage cycle containing only old objects (created before the previous sliding view). 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. The pointer modification could originate either from the application modifying a pointer (as in Figure 2), or from a reclamation of the object containing this pointer and the memory manager deleting the pointer. In the first case, the change of this pointer is logged in a local buffer causing a reference-count decrement to 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.

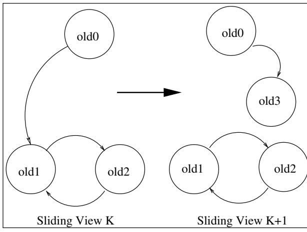

Fig. 2. A garbage cycle comprising solely of old objects created between the Kth and the $K+1^{\text{st}}$ sliding views. The cycle was reachable from *old0* and it became unreachable because *old0* was modified. Since *old0* is modified between the sliding views, *old0* (and its previous value *old1*) 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, even though the Levanoni-Petrank reference-counting collector executes only a small fraction of the reference-count updates, we may collect cycles correctly using as candidates only those objects whose reference counts are decremented by this collector to a non-zero value, plus the roots at the previous sliding view and all newly created objects.

3.3 Checking Behavior with the Age-Oriented Collector

Newly created objects add a substantial burden on the cycle collector. Therefore, we also used the proposed cycle collector with a collector that runs reference counting and cycle collection on the old generation only. We chose the age-oriented collector, a twist on generational collection that is adequate for concurrent collection. Our age-oriented collector runs concurrent reference counting on the old generation and concurrent mark and sweep on the young objects [27]. The age-oriented collector 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. 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 [27].

The age-oriented collector keeps generations, but it does not run frequent young generation collections. The reason for allowing entire heap collections 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. Therefore, it is relevant to check its performance with a cycle collector.

3.4 Reducing the Number of Traced Objects

New techniques for filtering and reducing the number of traced objects were designed and implemented in the proposed collector. For lack of space, these techniques are omitted. They are described in our technical report [26].

4 Measurements

An Implementation for Java. Our algorithm was implemented in Jikes RVM [1], a research Java virtual machine. 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).

Platform and benchmarks. We have taken measurements on a 4-way IBM Netfinity 8500R server with a 550MHz Intel Pentium III Xeon processor and 2GB of physical memory. The benchmarks used were the SPECjvm98 benchmark suite and the SPECjbb2000 benchmark (described in [29]). We feel that the multithreaded SPECjbb-2000 benchmark is more interesting, as the SPECjvm98 are more appropriate for clients and our algorithm is targeted at servers (multi-processors). SPECjbb2000 runs in a single JVM in which threads represent terminals in a warehouse. It is run with one terminal per warehouse, thus, the number of warehouses signifies the number of threads. 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 ([4, 13]) 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). The average of this 5 runs is used. Finally, each JVM was run on varying heap sizes. For the SPECjvm98 suite, we started with a 32MB[5] heap size and extended the sizes by 8MB increments until a final large size of 96MB. For SPECjbb2000 we started from 256MB heap size and extended by 64MB increments until a final large size of 704MB.

The compared collectors. The cycle collection algorithm was incorporated into two collectors: the Levanoni-Petrank reference-counting collector [20], and the more efficient age-oriented collector [27]. 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. This first ever reported comparison of cycle collection to a backup tracing collector is important since these are the main two options provided to an implementer of a reference-counting 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].

4.1 Performance

SPECjbb2000. Figure 3 depicts the throughput ratio between using the cycle collector and a backup tracing collector when both are used with the Levanoni-Petrank collector. 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.

[5] This is a tight heap for Jikes as it is self-hosted.

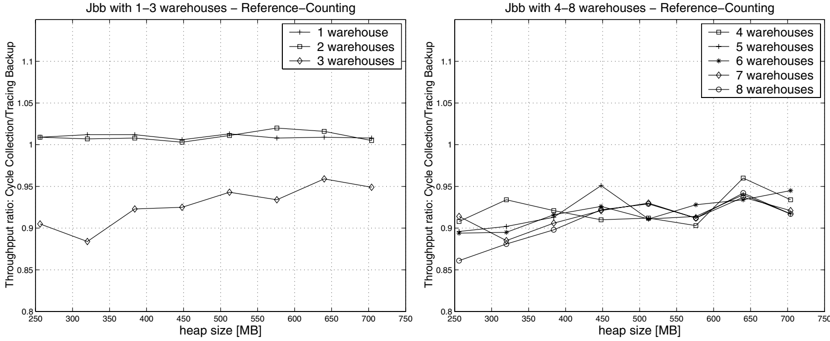

Fig. 3. SPECjbb2000 on a multiprocessor with the reference-counting collector. The higher the ratio, the better the cycle collector performs compared to the backup tracing algorithm

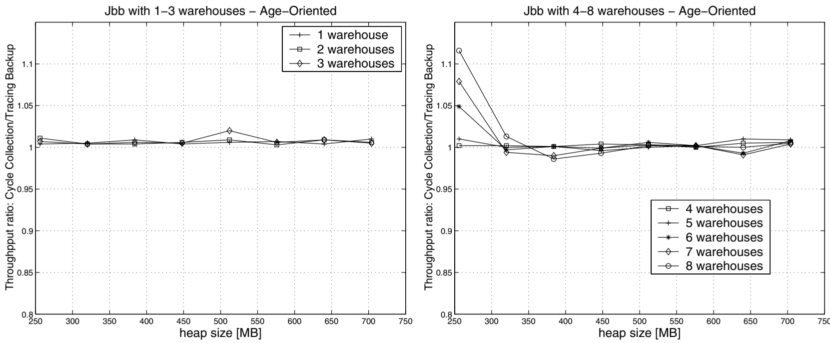

Fig. 4. SPECjbb2000 on a multiprocessor with the age-oriented collector. The higher the ratio, the better the cycle collector performs compared to the backup tracing algorithm

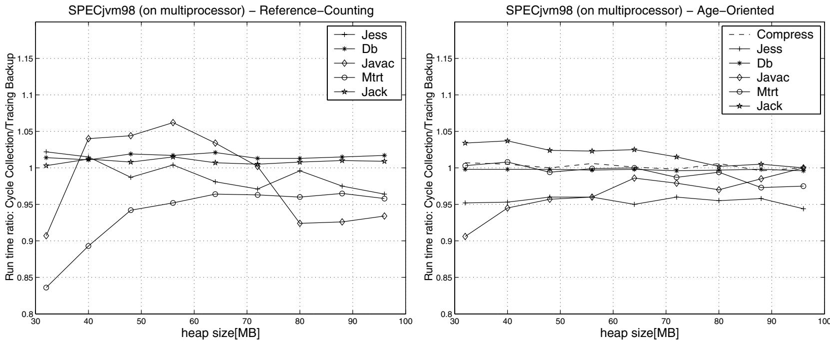

Fig. 5. SPECjvm98 on a multiprocessor. The higher the ratio, the better the cycle collector performs compared to the backup tracing algorithm

This is more noticeable on tight heaps. With 4–8 warehouses, the collector does not have a spare processor and 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 old generation of the age-oriented collector [27], see Figure 4. As only old objects are collected with reference counting and cycle collection, the behavior differs. Here, cycle collection performs usually as good as the backup tracing collector, whereas in tight heaps in which cycle collection wins. As already observed 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.

SPECjvm98. When running the SPECjvm98 benchmarks on a multiprocessor the collector runs concurrently with the program thread(s) on a spare processor. Figure 5 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. In _227_mtrt 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. Performance difference on _227_mtrt is not noticeable with the age-oriented collector, where the cycle collector is not run on this pack of young objects.

Discussion. At first glance, it may seem that backup tracing is the right choice. However, it is worth noting that modern platforms and benchmarks also run more efficiently with tracing than with reference counting [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 objects [6]. If future benchmarks use a large live heap or even a large old generation, then reference counting may become the best collector, and 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. Note also, that the best way to use reference counting today is to run it on the old generation only as proposed in [3, 7, 27]. In that case, running cycle collection with the reference counting is the right choice.

4.2 Pause Times

Table 1 presents the maximum pause times of the Levanoni-Petrank reference-counting collector accompanied by our cycle collection algorithm. Pauses were measured with a 64MB heap for SPECjvm98 benchmarks, and a 256MB heap for SPECjbb2000 with 1, 2, and 3 warehouses. For this number of threads, no thread gets swapped out, and so pauses are due to the garbage collection only. If we run more program threads, large

[6] 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.

Table 1. Maximum pause time in milliseconds

Benchmarks	Maximum pause time (ms)
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 2. Cyclic garbage collected for each benchmark

	RC		AO	
Bench-marks	cyclic objects reclaimed	cyclic bytes (in MB)	cyclic objects 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

pause times (whose lengths depend on the operating system scheduler) appear because threads lose the CPU to other threads.

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 of the collector with the program threads.

4.3 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 garbage cycles than the reference-counting collector.

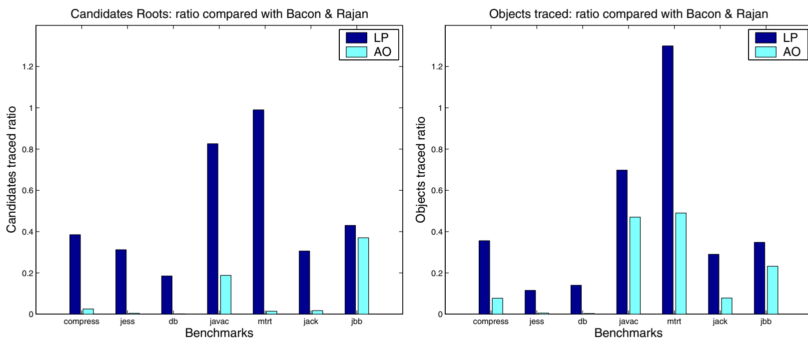

Fig. 6. Comparison between the new collector and the previous cycle collector of Bacon and Rajan. On the left, comparing the number of cycle candidates and on the right the number of traced objects. The lower the ratio, the better the new cycle collector algorithm behaves

The benchmarks producing a substantial amount of garbage cycles space are _213_javac and _201_compress. _201_compress creates dozens of garbage cycles comprised of huge objects, and thus requires only a small amount of tracing. _213_javac however, contains thousands of garbage cycles, thus requiring a large cycle collection work.

Amount of Tracing. Figure 6 reports the *candidates examined* and the *objects traced* ratios when the cycle collector is run with the Levanoni-Petrank collector (LP) and age-oriented collector (AO) compared to these of the cycle collector of Bacon and Rajan [5]. To be extremely conservative we did not include the objects scanned during the additional verification phase of [5] (since in this phase the actual operation on some objects only included work on their colors, i.e., they were not actually *traced*). Thus, the actual advantage of the new collector is even higher than reported.

Figure 6 shows that the new cycle collector with the Levanoni-Petrank collector traces fewer candidates compared to the previous cycle collector (of [5]) over all benchmarks[7]. It usually also traces substantially fewer objects except for one case: the _227_mtrt benchmark (discussed above). The additional saving when the cycle collector is used with the age-oriented collector is substantial for most benchmarks.

5 Related Work

The inability of reference counting to reclaim cyclic garbage structures was first noticed by McBeth [24]. Martinez et al. [23] (inspired by [10]) reclaim 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 [21] postponed these 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. Lins [22] showed the algorithm can employ 2 graph traversals (instead of 3) per candidate by using an extra data structure.

6 Conclusion

We presented a new non-intrusive, complete, and efficient 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 [20] with the synchronous cycle collector of Bacon and Rajan [5]. These algorithms do not naturally 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

[7] These measurements include the new techniques reducing the number of traced objects, reported in our technical report [26].

that the information gathered by the reference-counting collector is enough to guarantee reclamation of all unreachable cycles.

The use of the sliding-views mechanism yields a drastic improvement in efficiency. Much of the work required to ensure concurrent correctness may be eliminated. We have further added filtering techniques to 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 prevented 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, although it is the most efficient cycle collector available. However, when the reference-counting collector was run only on the old generation, the cycle collector performed equally to the backup tracing collector, and even better on tight heaps. Thus on today's platforms and benchmarks cycle collection is effective when applied to the old generation only. In the future, as heaps and live data become much larger, the techniques described in this work may become a preferred and most effective method to reclaim garbage.

Acknowledgement. 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, 1999.
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 [25].
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*, volume 2622 of *LNCS*, pages 185–199, 2003.
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.
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*, 2001.
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. Stephen M. Blackburn and Kathryn S. McKinley. Ulterior reference counting: Fast garbage collection without a long wait. In OOPSLA [25].

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. Edsgar W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):965–975, November 1978.
13. 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.
14. Toshio Endo, Kenjiro Taura, and Akinori Yonezawa. A scalable mark-sweep garbage collector on large-scale shared-memory machines. In *Proceedings of High Performance Computing and Networking (SC'97)*, 1997.
15. Christine Flood, Dave Detlefs, Nir Shavit, and Catherine Zhang. Parallel garbage collection for shared memory multiprocessors. In *Usenix Java Virtual Machine Research and Technology Symposium (JVM '01)*, Monterey, CA, April 2001.
16. Tony Hosking, editor. *ISMM 2000 Proceedings of the Second International Symposium on Memory Management*, volume 36(1) of *ACM SIGPLAN Notices*, 2000.
17. 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, 2001.
18. Elliot K. Kolodner and Erez Petrank. Parallel copying garbage collection using delayed allocation. In *Parallel Processing Letters*, volume 14, June 2004.
19. Yossi Levanoni and Erez Petrank. A scalable reference counting garbage collector. Technical Report CS–0967, Technion — Israel Institute of Technology, Haifa, Israel, November 1999.
20. 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.
21. Rafael D. Lins. Cyclic reference counting with lazy mark-scan. *IPL*, 44(4):215–220, 1992.
22. Rafael D. Lins. An efficient algorithm for cyclic reference counting. *IPL*, 83:145–150, 2002.
23. A. D. Martinez, R. Wachenchauzer, and Rafael D. Lins. Cyclic reference counting with local mark-scan. *Information Processing Letters*, 34:31–35, 1990.
24. J. Harold McBeth. On the reference counter method. *CACM*, 6(9):575, September 1963.
25. *OOPSLA'03 ACM Conference on Object-Oriented Systems, Languages and Applications*, ACM SIGPLAN Notices, Anaheim, CA, November 2003.
26. 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.
27. Harel Paz, Erez Petrank, and Stephen M. Blackburn. Age-Oriented Concurrent Garbage Collection. In *Proceedings of the 14th Int. Conference on Compiler Construction*, 2005.
28. Tony Printezis and David Detlefs. A generational mostly-concurrent garbage collector. In Hosking [16].
29. SPEC Benchmarks. Standard Performance Evaluation Corporation. http://www.spec.org/, 1998,2000.
30. Guy L. Steele. Multiprocessing compactifying garbage collection. *Communications of the ACM*, 18(9):495–508, September 1975.
31. Guy L. Steele. Corrigendum: Multiprocessing compactifying garbage collection. *Communications of the ACM*, 19(6):354, June 1976.