CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2006; **18**:959–995

Integrating Generations with Advanced Reference Counting Garbage Collectors[†]

Hezi Azatchi[1] and Erez Petrank[2,*]

[1] *IBM Haifa Research Labs, Haifa University Campus, Mount Carmel, Haifa 31905, Israel. Email* hezia@cs.technion.ac.il *Work done while the author was at the Technion - Israel Institute of Technology.*
[2] *Dept. of Computer Science, Technion - Israel Institute of Technology, Haifa 32000, Israel. Email:* erez@cs.technion.ac.il

SUMMARY

We propose the use of generations with modern reference counting. A reference counting collector is well suited to collect the old generation, containing a large fraction of live objects that are modified infrequently. Such a collector can be combined with a tracing collector to collect the young generation, typically containing a small fraction of live objects. We have designed such a collector appropriate for run on a multiprocessor. As our building blocks, we used the sliding-views on-the-fly collectors.

We have implemented the new collector on the Jikes Research Java virtual machine (Jikes RVM) and compared it against the concurrent reference counting collector supplied with the Jikes RVM package. Our measurements demonstrate short pause times, retaining those of the original on-the-fly collectors and a gain in application throughput time. It turns out that a modern reference counting collector may benefit from the use of generations.

KEY WORDS: Runtime systems, Memory management, Garbage collection, Generational Garbage Collection.

*Correspondence to: Dept. of Computer Science, Technion - Israel Institute of Technology, Haifa 32000, Israel. Phone: +972-4-8294942, fax: +972-829-3900. Email: erez@cs.technion.ac.il
†An abridged version of this work has appeared in [5].
Contract/grant sponsor: This research was supported by the Coleman Cohen Academic Lectureship Fund and by the Technion V.P.R. Fund - Steiner Research Fund.

Received 5 December 2003
Revised 9 August 2004
Accepted 29 April 2005

1. Introduction

Automatic memory management is well acknowledged as an important tool for fast development of large reliable software. However, the garbage collection process has an important impact on the overall runtime performance. See, for example, measurements in [24, 9] demonstrating how collector modifications in a production JVM influence the overall throughput of the JVM on standard benchmarks, and the discussion on the impact of garbage collection on performance in [21]. Thus, clever design of efficient memory management and garbage collection is an important goal in today's technology.

In the context of modern SMP platforms and modern applications there is a growing need for a memory manager that is especially targeted at large server configurations: 64-bit shared-memory multiprocessors running multithreaded applications on a multi-gigabyte heap. Such applications include, for example, web application servers, which must provide relatively fast responses to client requests and scale to support thousands of clients.

Tracing garbage collectors [33, 13] traverse the live objects in the heap, starting from root pointers directly available to the application and tracing their descendants reachable via heap references. The traced objects are either marked or copied to a different area. In the first case, all unmarked objects may be reclaimed in a second phase denoted *sweep*. Such a collector is called a *mark and sweep* collector. In the latter case, after all live objects have been copied to another area, the previously active area may be reclaimed as a whole. Such a collector is called a *copying* collector. The performance bottle-neck for tracing collectors is the traversal of all live objects in the heap. (The sweep phase is usually run using auxiliary bitmaps that make it very efficient [12].) Tracing collectors cannot scale to very large heap sizes (with large spaces of live objects) since they must visit all the live objects in the heap.

Reference counting is a most intuitive method for automatic storage management, known since the sixties [14]. The main idea is to keep for each object a count of the number of references to the object. When this number becomes zero for an object o, we know that o can be reclaimed. With the spread of the 64-bit architectures and the increase in usage of multi-gigabyte heaps, reference counting seems very promising to future garbage collected systems. While tracing collectors's performance depend on the space occupied by live objects, reference counting is different. The amount of work required for reference counting is proportional to the amount of work done by the user program between collections plus the amount of space that is actually reclaimed. This method seems appropriate for large heaps.

Many well studied garbage collection algorithms are not suitable for multiprocessors. In particular, many collectors run on a single thread after all program threads have all been stopped. This causes bad processor utilization, and hinders scalability. In order to make better use of a multiprocessor, concurrent collectors have been presented and studied. A concurrent collector is a collector that does most of its collection work concurrently with the program without stopping the program threads. Most concurrent collectors need to stop all program threads at some point during the collection, in order to initiate and/or finish the collection, but the time the mutators must be halted is short. Using the notation of [24], *on-the-fly collectors* never stop all the program threads simultaneously. Instead, each thread cooperates with the collector at its own pace through a mechanism called (soft) handshakes. Such collectors are especially useful for systems in which stopping all the threads together for synchronization is relatively long and costly.

An on-the-fly reference-counting collector was proposed by Levanoni and Petrank [31]. Concurrency and reference counting seem adequate for handling large heaps on multiprocessors with very short pauses. Our goal is to increase the efficiency of the Levanoni-Petrank concurrent reference-counting collector while maintaining its very short pause times. To that end, we employ generational garbage collection. Generational garbage collectors rely on the assumption that most objects die young. The heap is partitioned into two parts: the young generation and the old generation. New objects are allocated in the young generation, which is collected frequently. Young objects that survive one or more collections are “promoted” to the older generation. If the generational assumption (i.e., that most objects die young) is indeed correct, we get several advantages. Pauses for the collection of the young generation are short; collections are more efficient since they concentrate on the young part of the heap where we expect to find a high percentage of garbage; and finally, the working set size is smaller both for the program (because it repeatedly reuses the young area) and for the collector (because most of the collections trace over a smaller portion of the heap).

In this paper we propose the use of reference counting for the old generation and mark and sweep for the young generation. Each of these collectors seems most suitable for the typical behavior of objects in the generation it is associated with. In the young generation, we expect most object to die. Thus, a tracing collector would be very efficient quickly traversing the small fraction of objects that are reachable from the roots. On the other hand, the old generation has a lower percentage of death, and thus, a reference counting collector, which does not trace the live objects best fits the collection of these objects. As building blocks, we use the modern on-the-fly sliding-views collectors: the reference counting collector [31] and the mark-and-sweep collector [4]. Our new generational collector is on-the-fly and like its building blocks, it employs a write barrier that uses no synchronization operation. This collector retains the non-obtrusiveness of the original on-the-fly collector and improves its efficiency.

In addition to achieving the efficiency improvement, the new study is interesting for showing that it is advantageous to use generations with reference counting and it is advantageous to use generations with an on-the-fly collector. It is not always clear that generational collection gains efficiency. For example, IBM’s garbage collector in their production JVM did not employ generations [21] as it was not clear it would help. Thus, it is interesting to see that generations help in various environments. The only other work that we are aware of that uses generations with an on-the-fly collector is the work of Domani, Kolodner, and Petrank [24][†].

[†]A partial incorporation of generations with an on-the-fly collector, used only for immutable objects was used by Doligez, Leroy and Gonthier [23, 22]. The whole scheme depends on the fact that many objects in ML are immutable. This is not true for Java and other imperative languages. Furthermore, the collection of the young generation is not concurrent. Each thread has its own private young generation (used only for immutable objects), which is collected while that thread is stopped.

Cycle collection. A major disadvantage of reference counting is that it does not collect cycles. In order to reclaim cyclic structures, we also use the on-the-fly mark-and-sweep collector of [4] occasionally to collect the full heap and reclaim garbage cycles.

1.1. Implementation and results

We have implemented our algorithm on the Jikes RVM - a Research Java Virtual Machine [2] version 2.0.3 (upon Linux Red-Hat 7.2). The entire system, including the collector itself is written in Java (extended with unsafe primitives to access raw memory). 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. These benchmarks are described in detail in SPEC's Web site[42].

We tested our new collectors against the Jikes RVM concurrent collector distributed with the Jikes RVM package. This collector is a reference counting concurrent collector developed at IBM and reported by Bacon et al. [6]. Our collector achieves excellent performance measures. The throughput is improved by up to 67% for the SPECjbb2000 benchmark. The pauses are also smaller. These results hold for a variety of heap sizes that were measured. We also measure the generational collector against the original reference counting collector from which it was constructed. Results show generally a nice improvement in throughput, while the pause times do not change.

1.2. Organization

In Section 3 we review reference counting developments in recent years and mention related work. In section 4 we present the sliding-views collectors we build on. In section 5 we present the generational algorithms. In section 6 we discuss our implementation and in section 7 we present our measurements. We conclude in section 8.

2. Related work

Generational collection. Generational garbage collection was introduced by Lieberman and Hewitt [32], and the first published implementation was by Ungar [45].

Concurrent collection. Concurrent collectors have been presented and studied in many previous papers (see for example, [7, 43, 44, 20, 3, 16, 11, 23, 22, 37, 40, 36, 9]). On-the-fly collectors have been proposed in [20, 23, 22, 25, 24, 6, 31, 4].

Historically, the study of concurrent reference counting for modern multithreaded environments and multiprocessor platforms has not been as extensive and thorough as the study of concurrent and parallel tracing collectors. However, recently, we have seen several studies and implementations of modern reference counting algorithms on modern platforms [39, 6, 31] building on and improving on previous work. Bacon et al. [6] and Levanoni and Petrank [31] following DeTreville [16] have presented on-the-fly reference counting algorithms that overcome the concurrency problems of reference counting. Levanoni and Petrank have completely eliminated the need for synchronization operations in the write barrier. In addition, the algorithm of Levanoni and Petrank drastically reduces the number of counter updates (for common benchmarks).

Generational reference counting. Concurrently with this work [5], Blackburn and McKinley [10] have designed and implemented a generational combination of reference counting for the old generation and copying collector for the young generation. In contrast to this work, they did not deal with concurrency, but suggested a novel control of pause times: each collection runs a full young generation collection and part of the old generation collection, as affordable with the pause time constraint. This yielded short controllable pause times. Also, by not dealing with concurrent mutations, some of the mechanisms used here are not required and a throughput improvements may be gained. Our collector, by using concurrency, achieves an order of magnitude shorter pause times when run on a multiprocessor.

Generational collection without moving objects. Usually, on-the-fly garbage collectors do not move objects[‡]. Demers, et al. [15] presented a generational collector that does not move objects. Their motivation was to adapt generations for conservative garbage collection. Here we exploit their ideas: instead of partitioning the heap physically and keeping the young objects in a separate area we partition the heap logically. For each object, we keep one bit indicating if it is young or old.

Further background on garbage collection can be found in [29]

3. An overview of reference counting algorithms

The traditional method of reference counting was first developed for Lisp by Collins [14]. The idea is to keep a reference count field for each object telling how many references exist to the object. Whenever a pointer is updated the system invokes a *write barrier* that keeps the reference counts updated. In particular, if the pointer is modified from pointing to O_1 to pointing to O_2 then the write barrier decrements the count of O_1 and increments the count of O_2. When the counter of an object is decreased to zero, it is reclaimed. The reference counts of all its children are then decremented as well and the reclamation may continue recursively.

This method was later used in SmallTalk-80 [27], the AWK [1] and Perl [46] programs. Improvements to the naive algorithm were suggested in several subsequent papers. Weizenbaum [47] studied ameliorating the delay introduced by recursive deletion. Several works [41, 50] use a single bit for each reference counter with a mechanism to handle overflows. They built on the fact that most objects are singly-referenced, except for the duration of short transitions.

[‡]Standard incremental collectors [7, 3, 34] require either a high efficiency overhead or a substantial pause time. The only exception is the Sapphire collector [28], which is an involved on-the-fly collector with very short pause times.

Deutsch and Bobrow [18] noted that most of the overhead on counter updates originates from the frequent updates of local references (in stack and registers). They suggested using the write barrier only for pointers on the heap. Now, when a reference count decreases to zero, the object can not be reclaimed since it may still be reachable from local references. Instead of reclaiming an object whose reference count is decremented to zero, such an object is recorded in a table denoted *the Zero Count Table*, ZCT. To collect objects, a garbage collection is invoked. During the collection one can reclaim all objects in the ZCT that still have zero heap reference count and are not accessible from local references. This method is called *deferred reference counting* and it yields a great saving in the write barrier overhead. It is used in most modern reference counting collectors. In particular, this method was later adapted for Modula-2+ [16]. Further study on reducing work for local variables can be found in [8] and [38].

Reference counting seemed to have an intrinsic problem with multithreading. First, the updates on the counters may be executed concurrently by more than one thread and so the updates must be atomic. Using a synchronization operation in the write barrier seems to be too costly. Since write barriers are run very often (with each pointer update), then even a compare-and-swap operation is too costly for the overall program throughput. An even more problematic issue is the fact that, if the user program allows races, then the collector may fail to operate correctly. For example, suppose Thread T_1 executes $O.next \leftarrow B$ while T_2 concurrently executes $O.next \leftarrow C$. Suppose also, that before these operations started concurrently, $O.next$ pointed to the object A. Using a naive write barrier, both threads start by reading the old value of $O.next$ (so that they can later decrement its reference count). They both read A as the referenced object. Then, both perform the assignment. Due to the race, only one assignment prevails. Finally, both T_1 and T_2 decrement the counter of the old object and increment the counter of the new one. Thus, both will reduce the reference count of A and they will increment the counts of B and C. Note that the counters are now wrong. First, A's count has been decremented twice (which is wrong since only one pointer was moved) and second, the counters of both B and C were incremented while only one of them obtained a new reference.

The first system to work with reference counting in a multithreaded environment was presented by DeTreville [16]. He described a concurrent multiprocessor reference counting collector for Modula-2+. DeTreville's algorithm adopted Deutsch and Bobrow's ideas of deferred reference counting and added the idea of a local transaction log. Each thread records its modifications in a local transaction log that requires no synchronization. During the collection, the collector uses all transaction logs to actually update the counters. This eliminates the races on the reference counts since only the collector modifies them, but leaves the problem with program races. To solve that, DeTreville used a single central lock for each update operation (on heap references). This implies that only a single update can occur simultaneously in the system, placing a hard bound on its scalability.

Plakal and Fischer in [39] proposed a collection method based on reference counting for architectures that support explicit multi-threading on the processor level. Bacon et al. [6] provided a solution to the entire garbage collection problem with a pure reference counting approach, showing that this is feasible. Their contributions include a novel on-the-fly cycle collector, an improvement upon Deutsch and Bobrow's Deferred Reference Counting algorithm that does not require use of Zero Count Tables, and an improvement over DeTreville's algorithm for concurrent collection by introducing epochs that eliminate the requirement for a single shared buffer for recording information on required increments and decrements of reference counts. In terms of synchronization, they have reduced the need for the central lock and suggested one compare-and-swap instead. This is an improvement over the naive approach which would require at least three compare-and-swap's.

Concurrently with Bacon et. al., Levanoni and Petrank [31] presented a reference counting collector that completely eliminated the need for a synchronization operation in the write barrier. Levanoni and Petrank also build on Deutsch and Bobrow and on DeTreville, using deferred reference counting with local transaction buffers. However, they suggest a carefully designed write barrier and a careful analysis showing that program races can also be overcome without synchronization. Using their write barrier, a program race would result in multiple copies of transaction records rather than in foiling the counts. Their collector can easily overcome multiple records in the transaction buffers.

In addition to the improvement in synchronization, Levanoni and Petrank also presented a significant reduction in the overhead on transaction logging and counter updates. 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 $2n$ updates of reference counts for these assignments: RC(o_0)- -, RC(o_1)++, RC(o_1)- -, RC(o_2)++, ..., RC(o_n)++. However, only two are required: RC(o_0)- - and RC(o_n)++. Furthermore, in modern collectors in which the program threads log updates in a transaction buffer, only one transaction needs to be logged, whereas other collectors log n transactions. Furthermore, records for new objects do not need to be taken, since objects are created with null pointers. As most modifications are performed on new objects (in typical benchmarks), measurements have shown that this improvement reduces the number of logs and counter updates by a factor of 100-1000 for standard Java benchmarks.

4. The sliding-views collectors

In this section we provide a short overview of the sliding-views collectors. In the later description of the generational collectors, we will adopt a convention of adding an asterisk to any line in the generational algorithm pseudo code that differs from the original collector. A more detailed description of the original algorithms appear in the original papers [31, 4].

4.1. The sliding-views reference counting algorithm

Levanoni and Petrank [31] have presented an on-the-fly reference counting algorithm (the sliding views algorithm) with two interesting algorithmic properties: it saves many of the reference count updates, and it does not require synchronization in the write barrier. Thus, they obtain a non-disruptive efficient on-the-fly reference-counting collector.

As discussed above, the first contribution of the Levanoni-Petrank collector is the elimination of many redundant reference count updates. Consider a pointer slot that, between two garbage collections is assigned the values $o_0, o_1, o_2, \ldots, o_n$. Only two updates are required out of the $2n$ implied updates: RC(o_0)- - and RC(o_n)++. Furthermore, modern reference counting collectors let the program threads log updates in a transaction buffer. Using the above observation, we note that only one transaction needs to be logged, whereas other collectors log all the n transactions.

The write barrier checks whether an object has already been logged. If not, its non-null pointer values are logged in a local buffer before the actual modification takes place. The collector employs a carefully designed write barrier that allows running this check in parallel among the running program threads without any synchronization operation (even not a compare-and-swap type of operation). A careful analysis of this write barrier shows that a program race may result in multiple copies of object recording but not contradicting ones. The collector may then discover the valid pointer values from one of those buffers. The Levanoni-Petrank collector is the first collector that allows concurrent reference counting with no synchronization in the write barrier.

Using the write barrier described above, the mutators record all heap objects whose pointer slots are modified between one collection to the next. The recorded information is the address of the modified object as well as the values of the object's pointer slots before the current modification. As mentioned, a dirty flag is used to let only one record be kept for any modified object. The analysis shows that (infrequent) races may cause more than one record be created for an object, but all such records contain the same information[§]. The records are written into a local buffer with no synchronization. The dirty flag is actually implemented as a pointer, being either null when the flag is clear, or a pointer *o.LogPointer* to the logging location in the local buffer if the flag is set. The pseudo code appears in figures 5-17 (Lines with asterisk in their numbers should be ignored, these lines represent modifications to the Levanoni-Petrank algorithm).

All created objects are marked dirty during creation (figure 4), by setting the *o.LogPointer* pointer to their logging location. There is no need to log their slots' (children) values as they are all null at creation time (and thus, also during the previous collection). But objects that will be referenced by these slots during the next collection must be noted and their reference counts must be incremented.

A collection begins by taking a sliding view of the heap (figures 7-9). A sliding-view is essentially a non-atomic snapshot of the heap. It is obtained incrementally, i.e., the mutators are not stopped simultaneously. Such a non-atomic snapshot introduces a correctness danger: objects reachability may be wrongly computed by the collector because pointers keep changing in the heap. A solution to this problem is the *snooping* mechanism. While the view is being read from the heap, the snooping mechanism (via the write-barrier) records (locally) any object to which a new reference is created in the heap. These objects are marked as *Snooped* by ascribing them to the threads' local buffer: $Snooped_i$, thus, preventing them from being collected in this collection cycle mistakenly. It is shown in [30] that this mechanism ensures correctness.

To achieve cooperation between the collector and the program threads, handshakes are used. During a handshake, each thread is halted (separately, not simultaneously) for a short pause

[§]This actual property requires some care, see the original paper [31] to cooperate with the collector. During a halt, data may be exchanged between the collector and the program threads.

The Levanoni-Petrank collector employs four handshakes during the collection cycle (see figures 7-10). The collection starts with the collector raising the $Snoop_i$ flag of each thread, signaling to the mutators that it is about to start computing a sliding-view. During the first handshake, mutator local buffers are retrieved and then are cleared. There are two such buffers for each mutator. $YoungObjects_i$ contains all objects created since the last collection by Mutator i and $Updates_i$ contains all objects modified by Mutator i since the last collection together with their previous sliding view non-null pointer slot values. Next, the dirty flags of the objects listed in the buffers (both the *YoungObjects* and the *Updates* buffer) are cleared while the mutators are running (figure 8). Most of the clearing operations in this stage clear dirty bits created during the previous collection cycle, as intended. But some clearing is done to objects that have been dirtied concurrently by the running program threads. Such dirty bits should not be cleared. The collector proceeds by fixing this extra clearing. It reads the new local buffers from all mutators in a second handshake and sets the dirty bits for the (small number of) objects that have been dirtied and logged in the buffers concurrently with the collector clearing process (figure 9). A third handshake with no specific operation is executed to make sure that the reinforced dirty bits are visible to all mutators. A fourth handshake is used to scan threads local states and objects directly reachable from the roots are marked as *Roots*.

After the fourth handshake the collector proceeds to adjust *rc* fields due to differences between the sliding views of the previous and current cycle (figure 15). Each object which is logged to one of the mutator's local buffers has been modified since the previous collection cycle, thus we need to decrement the *rc* of its children (as reflected by its pointer slot values) as appearing in the previous sliding-view and increment the *rc* of its slots values in the current sliding-view. The *rc* decrement operation of each modified object is done using the objects' information obtained from the retrieved local buffers. This information contains the object non-null pointer slots' value at the previous sliding-view.

Deciding which objects *rc* to increment, i.e., what are the children of an object O in the current sliding view is a bit more involved, since we cannot assume that all these children appear in some given list. If O has not been modified since the beginning of this collection then its pointer slots values may be read from the heap. If it has been modified, then the values of its pointer slots in the current sliding view must be obtained from the new *updates* buffers currently being written by the threads. Note that care is required here since threads are modifying the dirty bit while the collector is trying to determine if the object is currently dirty or not. This possible race is solved by the special way the collector checks these objects. The collector first checks if the object's dirty flag, *o.LogPointer*, is not null. If it is not null, then the values of this object may be obtained from the current local buffers (via the *o.LogPointer* pointer). Otherwise, we take a temporary replica of the object and check again if it is dirty. If not, then we may use the replica for finding the slots values. Otherwise, the object got dirty while the replica was taken, then we must read the buffer values.

A collection cycle ends with reclamation which recursively free any object with zero *rc* field which is not marked as *local* (figure 16).

4.2. The sliding-views tracing algorithm

"Snapshot at the beginning" [51, 48, 49, 26] mark&sweep collectors exploit the fact that a garbage object remains garbage until the collector recycles it. i.e., being garbage is a stable property. A snapshot at the beginning garbage collector virtually freezes a view of the heap at the beginning of the collection time and then identifies unreachable objects in this snapshotted heap. All objects that are identified as unreachable must still be unreachable in spite of mutators' further activity, as mutators have no access to unreachable objects.

The tracing collector of [4] collects the heap based on a sliding view of the heap. Using the same mechanism proposed for the reference counting collector, the tracing collector implicitly computes a sliding view of the heap and traces it (see figures 7-10). After the Mark-Roots stage, the collector starts tracing according to the sliding view associated with the cycle (figures 10, 12). When in need to trace through an object the collector tries to determine its value in the sliding view as was done in the previous algorithm, i.e. by checking if the object's *LogPointer* (the dirty flag) is set. If it is set each object's slot sliding-view value can be found directly from the committed already (by some mutator) replica which is pointed by the object's *LogPointer*. If it is not set, a temporary replica of the object is taken and is confirmed by checking again if the object's dirty flag is still not set. If the replica is committed the collector continues by tracing through the object's replica. Finally, the collector proceeds to reclaim garbage objects by sweeping the heap (figure 14).

The algorithm can infer whether an object is garbage or not only if it has been allocated prior to the fourth handshake. Each thread has a local variable, denoted $AllocColor_i$ that holds the color the thread has to assign to the *color* field of newly allocated objects, (see figures 4, 10). The variable is toggled between two colors, *black* and *white*. During sweeping, the collector considers each object in the heap. If the object is *black*, then it is retained. If it is colored *blue*, then it is ignored (this is an object that is not allocated). Otherwise, the object is *white*. In that case the collector reclaims the object by coloring it as *blue* and passing it back to the allocator. Thus, when sweeping is over, the heap contains only *black* or *blue* objects since any object which had been *white* was turned *blue* and mutators color newly allocated objects *black*. Before starting the tracing of the next cycle the collector toggles the values of *black* and *white* variables, so all objects allocated prior to the next cycle's fourth handshake are considered "unmarked", thus can be reclaimed if they are garbage.

5. Collector Overview

In the following sections we describe the generational collector we have designed and implemented. This collector runs reference counting on the old generation and tracing on the young. We first discuss general issues and then describe the collector of each generation.

The generational collector uses the same four handshakes as the original collectors of [31] to take a sliding view of the heap. A sliding view is being taken by the procedures: Initiate-Collection-Cycle, Clear-Dirty-Marks, Reinforce-Clearing-Conflict-Set and Mark-Roots (pseudo-code appears in figures 7 - 10). The generational algorithm differs from the original collector of [31] in the way it handles the sliding view after it is (virtually) taken.

The promotion policy used is simple. The young generation holds all objects allocated since the previous collection and each object that survives a young (or full) collection is immediately promoted to the old generation. This naive promotion policy fits nicely into the algorithm we use. Generations are not segregated in the heap since we do not move objects in the heap. In order to determine efficiently if an object is young or old, we keep a bitmap (1 bit for each 8 bytes) telling which objects are old. All objects are created young and promotion modifies this bit.

5.1. The minor (mark and sweep) collection

The mark and sweep minor collection marks all reachable young objects at the current sliding view and then sweeps all the young and unmarked objects. The young generation contains all the objects that were created since the *previous* collection cycle. These objects were logged by the mutators to their *Young-Objects* local buffers. These local buffers hold addresses of all newly created objects (created since the last collection) and can be also viewed as holding pointers to all objects in the young generation to be processed by the next minor collection. In the first handshake of a collection, these buffers are retrieved by the collector and their union is taken and stored in a buffer called *Young-Objects*. This *Young-Objects* buffer will serve the collector as its young generation to be processed in this minor collection.

Recall that we are using the Levanoni-Petrank sliding view collectors as the basis for this work. The sliding-views algorithm uses a dirty flag for each object to tell if it was modified since the previous collection. All modified objects are kept in an *Updates* buffer (which is essentially the union of all mutator's $Updates_i$ local buffers) so that the *rc* fields of objects referenced by these objects' slots can later be updated by the collector. Since we are using the naive promotion policy, we may use these buffers also as our remembered set. The young generation contains only objects that have been created since the last collection, thus, it follows that inter-generational pointers may only be located in pointer slots that have been modified since the last collection. Clearly, objects in the old generation that point to young objects must have been modified since the last collection cycle, since the young objects did not exist prior to this collection. Thus, the addresses of all the inter-generational pointers must appear in the *Updates* buffer of the collector at this collection cycle. The scenario of creating a new inter-generational pointer is depicted in figure 1. When a pointer in the (old) object A is modified to point to a (new) object C, the address of A is added to the *Updates* buffer. The previous value of this pointer (the address of B) is also put in the buffer, but that is irrelevant for inter-generational pointers.

At first glance it may appear that this is enough. However, the collection does not use a snapshot, but a sliding view. Since it runs concurrently with the program threads and the program threads are never stopped simultaneously for the collection, then, referring to the time of the last collection cycle is not accurate. There are two cases in which inter-generational pointers are created but do not appear in the *Updates* buffer read by the collector in the first handshake.

Case 1: Mutator M_j creates a new object O after responding to the first handshake. Later, Mutator M_i, who has not yet responded to the first handshake executes an update operation

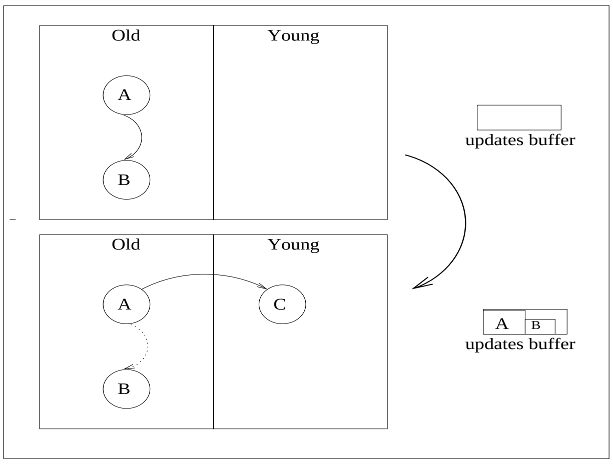

Figure 1. A standard creation of an inter-generational pointer

assigning a pointer in the old generation to reference the object O. In this case, an inter-generational pointer is created: the object O was not reported to the collector in the first handshake and thus, will not be reclaimed or promoted in the current collection. It will be reported as a young object to the collector only in the next collection. But the update is recorded in the current collection (the update was executed before the first handshake in the view of Mutator M_i) and will not be seen in the next collection. Thus, an inter-generational pointer will be missing from the view of the next collection. This scenario is depicted in figure 2.

Case 2: Some mutator updates a pointer slot in an object O to reference a young object. The object O is currently dirty because of the previous collection cycle, i.e., the first handshake has occurred, but the clear dirty flags operation was not yet executed for that object. In this case, an inter-generational pointer is created but it is not logged to the i-th mutator $Updates_i$ local buffer. Indeed, this pointer slot must appear in the $Updates$ buffer of the previous collection and correctness of the original algorithm is not foiled, yet, in the next cycle the $Updates$ buffer might not contain this pointer, thus an inter-generational pointer may be missing from the view of next collection. This scenario is depicted in figure 3.

In order to identify inter-generational pointers that are created in one of the above two manners correctly, each minor collection records into a special buffer called the *IGP-Buffer*, all the addresses of objects that had to do with updates to young objects in the uncertainty period of from before the first handshake has begun to after the clear dirty flags operation is over for all the modified (logged) objects, (see figures 5, 10). The next collection cycle will use the *IGP-Buffer* buffer that was appended in the *previous* collection cycle as its *PrevIGP-Buffer* buffer in order to scan the potential inter-generational pointers that might have not appeared

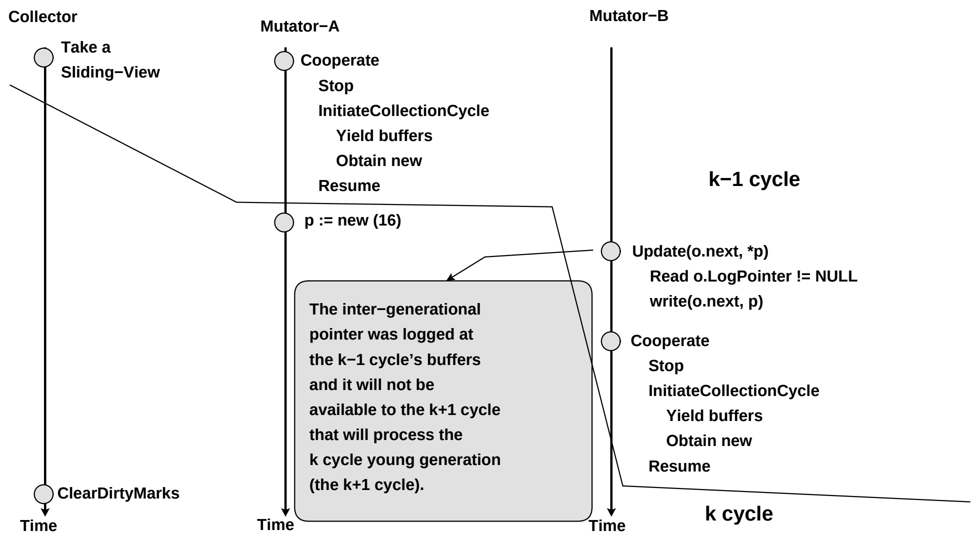

Figure 2. The first case: an inter-generational pointer is logged too early.

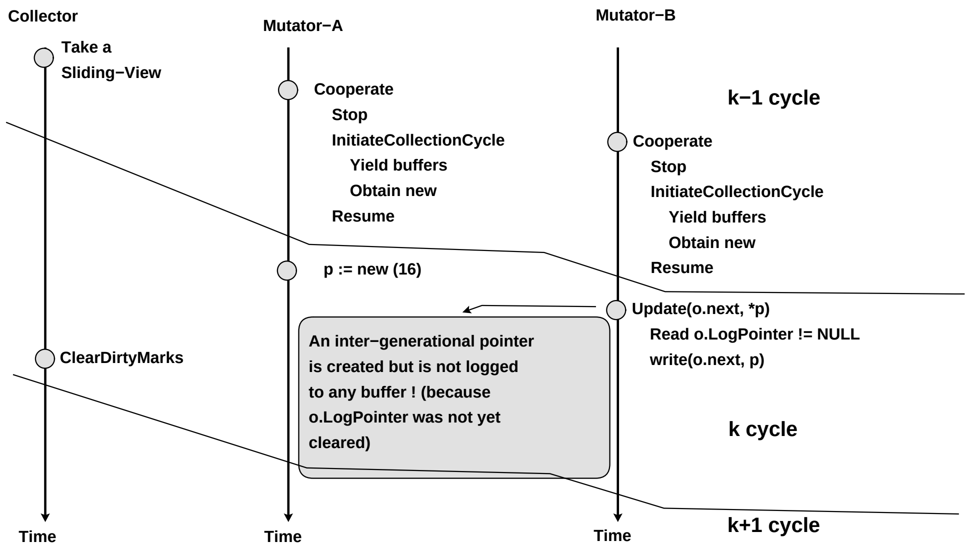

Figure 3. The second case: a pointer is updated before its dirty bit is cleared.

in the *Updates* buffer. In this way, we ensure that all inter-generational pointers are discovered for each minor collection.

Finally, we note that the sweep phase processes only young objects. It scans each object's color in the *Young-Buffer*, which is actually the young generation that is processed by this minor collection. Objects which are marked *white* are reclaimed, otherwise, they are promoted by setting their *old* flag. The pseudo-code for this operation can be seen in figure 14.

One issue to be handled is that while we are collecting, it is possible that the address of an object to be reclaimed appears in the *Updates* buffer that is going to be used by the next collection. This may happen since the collector runs concurrently with the mutators and the mutators are never stopped simultaneously to synchronize with the collector. For example, suppose mutator M_i creates an object after responding to the first handshake. It marks the new object dirty and puts it in the *Young-Objects* local buffer. Later, the collector clears the dirty bit of the object and before the dirty bit is restored, another (or the same) mutator writes a pointer to the object, puts it in the *updates* buffer and marks the object dirty. When this rare case happens and we later collect the object, we invalidate the log entry in the *updates* buffer and reclaim it. Since the log entry is invalidated, the next collection will not deal with this object.

5.2. The full (reference counting) collection

Two buffers denoted *Major-New-Objects* and *Major-Updates* buffers are used to aid the full collection. They correspond to the minor collection's *Young-Objects* and *Updates* buffers. These buffers are prepared by the minor collections to serve the full collection, their preparation is executed during the sweep (figure 14). They are then used by the reference counting algorithm as if they were prepared in the normal manner without generations (see figure 15).

Logging to the Major-Updates buffer. One bothering issue is that the minor collector may reclaim objects whose records appear in the *Major-Updates* or *Major-New-Objects* buffers. Recall that buffers' entries include object's addresses and values of their pointer slots during the previous sliding view. These old values of pointer slots may refer to objects that are being reclaimed by the young generation collector. However, the logged objects themselves (that appear in the *Updates* buffer) must be old objects as young objects are recorded in the *Young-Objects* buffer exclusively. The *Updates* buffers hold only old objects because we are in the middle of the collection after the Initiate-Collection-Cycle, except for the rare race discussed above. The objects logged in the *Updates* buffer are in the old generation and cannot be collected by the young generation collector. To summarize, when handling the *Updates* buffer, we can safely move recorded objects to the *Major-Updates* buffer but we check each pointer slot logged to make sure that the referent object survives the sweep phase.

Using deferred reference counting ([31] following [18]), we employ a *zero count table* denoted ZCT to hold each young object whose count decreases to zero during the counter updates. All these candidates are checked after all the updates are done. If their reference count is still zero and they are not referenced from the roots, then they may be reclaimed. Note that all newly created (young) objects must be checked since they are created with reference count zero (they are only referenced by local variables in the beginning.) Thus, all objects in the *Young-Objects*

```
Procedure New(size: Integer) : Object
begin
1.    Obtain an object $o$ of size $size$ from the allocator.
2.    $o$.color := $AllocColor_i$
3.    $Young\text{-}Objects_i[Young\text{-}Objects\text{-}CurrPos{+}{+}]$ := address of $o$
4.    $o$.LogPointer = address of $Young\text{-}Objects_i[Young\text{-}Objects\text{-}CurrPos]$
5.    return $o$
end
```

Figure 4. Mutator code: **Allocation Operation**

as well as in the *Major-New-Objects* buffer are appended to the ZCT that is reclaimed by the collector. The code can be seen in figures 16, 17.

The inability of reference counters algorithms to reclaim cyclic structures is treated with an auxiliary on-the-fly mark-and-sweep algorithm used infrequently during the full collection.

5.3. Mutator Code

The mutator code in this algorithm is similar to that of Levanoni-Petrank sliding-view reference-counting algorithm in [31]. The **New** procedure (in figure 4) logs the address of the newly allocated object into a local to the mutator buffer called $Young\text{-}Objects_i$. This buffer includes the addresses of the objects which will reside in the young generation to be processed by the next collection cycle. As in the original collector, there is no need to record their children slots values as they are all null at creation time. Objects that will be referenced by these slots during the next collection must be noted and their reference counts must be incremented.

The **Update** procedure (in figure 5) is responsible for the logging of objects values at the sliding views to local buffers. Each object's *LogPointer* is optimistically probed twice (lines 1 and 5 in the pseudo-code) so that if the object is dirty (which is often the case), then the write barrier is extremely fast. If the object was not logged (i.e., the LogPointer of an object is NULL) then after the first probe, the object's values are recorded into the local $Updates_i$ (lines 2-4). The second probe at line 5, ensures that the object has not yet been logged (by another thread). If *LogPointer* is still NULL (in the second probe), then the recorded values are committed to and the buffer pointer is modified (lines 6-8). Then, the object's *LogPointer* field is set to point to these values (line 8). After logging has occurred, the actual pointer modification happens. Finally, while marking the roots of the mutators, the snoop flag is on. At that time, the new target of the pointer assignment is recorded in the local snooped buffer. This happens in lines 10-11. The variables $Updates_i$, *CurrPos*, *snoop* and $Snoop_i$ are local to the mutator. Besides the logging to the *Updates* buffer, the *Update* procedure includes an additional test that checks whether the flag $LogIGP_i$ is raised and the referenced object is new, in that case it adds the slot into IGP_i local buffer for the next collection use. As explained

```
Procedure Update(obj: Object, offset: int , new: Object)
begin
1.    if obj.LogPointer = NULL then // object is not dirty
2.        TempPos := CurrPos
          // take a temporary replica of the object
3.        foreach field ptr of obj which is not NULL
4.            Updates_i[TempPos++] := ptr
5.        if obj.LogPointer = NULL then // commit the replica
6.            Updates_i[TempPos++] := address-of obj
7.            CurrPos := TempPos
              // set the dirty flag
8.            obj.LogPointer := address-of Updates_i[CurrPos]
9.    write( obj, offset ,new)
10.   if (Snoop_i and new != NULL) then
11.       Snooped_i := Snooped_i ∪ {new}
*12.      if (LogIGP_i and new.old = false) then
*13.          IGP-Buffer_i := IGP-Buffer_i ∪ {obj}
end
```

Figure 5. Mutator code: **Update Operation**

in Section 5.1 above, this buffer will contain additional potential inter-generational-pointers that the *Updates* buffer stolen from the mutators in the next collection cycle does not contain for the use of the next collection cycle.

5.4. Collector Code

Procedure Collection-Cycle (figure 6) contains the high level main algorithm. The minor collection is a tracing collection whereas the full collection is a reference counting collection. The global flag *majorCollection* determines the current collection type: young or full.

Procedure Initiate-Collection-Cycle (figure 7) starts the collection. The $Snoop_i$ flag tells the thread to start snooping and the $LogIGP_i$ flag tells the mutators to start logging to the IGP_i Buffer slots that are candidates to being future inter-generational-pointers. Note that the setting of the snoop flag for each thread is done without stopping the threads. Later, the threads are stopped via a soft handshake (the first handshake) to gather the local buffers of the threads.

Procedure Clear-Dirty-Marks (figure 8) clears all dirty marks concurrently with mutators run.

Procedure Reinforce-Clearing-Conflict-Set (figure 9) implements the reinforcement step for the dirty bits and assures that it is visible to all mutators. Two handshakes appear here, and

```
Procedure **Collection-Cycle**
begin
1.     Initiate-Collection-Cycle
2.     Clear-Dirty-Marks
3.     Reinforce-Clearing-Conflict-Set
4.     Mark-Roots
*5.    if ($majorCollection$) then
*6.        Update-Reference-Counters
*7.        Reclaim-Garbage
*8.    else
*9.        Add-Potential-IGPs-To-Roots
*10.       Mark
*11.       Sweep
end
```

Figure 6. **Collection Cycle**

```
Procedure **Initiate-Collection-Cycle**
begin
1.     for each thread $T_i$ do
2.         $Snoop_i$ := true
*3.        $LogIGP_i$ := true
4.     for each thread $T_i$ do
5.         suspend thread $T_i$
           // copy (without duplicates).
6.         $Updates := Updates \cup Updates_i$
7.         $Updates_i := \oslash$ // clear buffer.
8.         $Young\text{-}Objects := Young\text{-}Objects \cup Young\text{-}Objects_i$
9.         $Young\text{-}Objects_i := \oslash$ // clear buffer.
10.        resume thread $T_i$
end
```

Figure 7. **Initiate-Collection-Cycle**

note that the second handshake (the third overall) has no operation. The goal is only to make sure that the dirty bits are visible to all threads.

Procedure Mark-Roots (figure 10) uses a fourth handshake to get the root set of this collection cycle. A standard color-toggle scheme [17, 24] is used to mark all black objects white and start the trace. New objects are going to be colored black and will not be reclaimed this cycle. Note that it is correct to mark new objects black during the collection, since they are alive and have no children at the time of creation. The inter-generational pointers are gathered in

```
Procedure **Clear-Dirty-Marks**
**begin**
**1.**    for each object $o \in Updates$ do
**2.**        $o$.LogPointer := NULL
**3.**    for each object $o \in Young\text{-}Objects$ do
**4.**        $o$.LogPointer := NULL
**end**
```

Figure 8. **Clear-Dirty-Marks**

```
Procedure **Reinforce-Clearing-Conflict-Set**
**begin**
**1.**    $ClearingConflictSet := \emptyset$
**2.**    for each thread $T_i$ do
**3.**        suspend thread $T_i$
***4.**       $LogIGP_i$ := **false**
**5.**        $ClearingConflictSet :=$
                  $ClearingConflictSet \cup Updates_i[1 \ldots CurrPos_i - 1]$
**6.**        resume thread $T_i$
**7.**    for each object $o \in ClearingConflictSet$ do
**8.**        if $o$.LogPointer = NULL then // reinforce
**9.**            $o$.LogPointer := address of $o$'s replica in $Updates$
**10.**   for each mutator $T_i$ do
**11.**       suspend mutator $T_i$
**12.**       **nop**
**13.**       resume $T_i$
**end**
```

Figure 9. **Reinforce-Clearing-Conflict-Set**

the global *IGP-Buffer*. As discussed in Section 5.1 some of these pointers are also recorded in the *prevIGP-Buffer* for the next collection, and those recorded previously are used in this collection.

Procedure Mark (figure 11), and *Procedure Trace* (figure 12) implement the tracing stage of the young generation collector. Before they run, *Procedure Add-Potential-IGPs-To-Roots* (figure 13) is called to add the inter-generational pointers to the root set.

Procedure Sweep (figure 14) reclaims the dead young objects and promotes the surviving ones. Promotion is done by setting the **old** flag. The Sweep procedure also prepares the *Major-New-Objects* and *Major-Updates* buffers which will serve the full collection as its "*Young-Objects*"

```
Procedure **Mark-Roots**
**begin**
**1.**    $black := 1\text{-}black$
**2.**    $white := 1\text{-}white$
***3.**   $PrevIGP\text{-}Buffer := PrevIGP\text{-}Buffer \cup IGP\text{-}Buffer$
***4.**   $IGP\text{-}Buffer := \oslash$
**5.**    for each thread $T_i$ do
**6.**        suspend thread $T_i$
**7.**        $AllocColor_i := black$
**8.**        $Snoop_i :=$ **false**
**9.**        $Roots := Roots \cup State_i$ // copy thread local state.
***10.**      $IGP\text{-}Buffer := IGP\text{-}Buffer \cup IGP\text{-}Buffer_i$
**11.**       resume thread $T_i$
***12.**      $IGP\text{-}Buffer_i := \oslash$
**13.**   for each thread $T_i$ do
              // copy and clear snooped objects set
**14.**       $Roots := Roots \cup Snooped_i$
**15.**       $Snooped_i := \oslash$
**end**
```

Figure 10. **Mark-Roots**

```
Procedure **Mark**
**begin**
          // Mark objects from Roots
**1.**    for each object $o \in Roots$ do
**2.**        **Trace**($o$)
**end**
```

Figure 11. **Mark**

and "*Updates*" buffers respectively. After the live objects are determined, the *Major-Updates* buffer is updated, but only pointers to objects that were not reclaimed are copied.

Procedure Update-Reference-Counters (figure 15) updates reference counters of modified objects. Buffers are accumulated both from the recent short (young) cycle and from the buffers prepared by all recent young generations.

Procedure Reclaim-Garbage (figure 16) scans the *ZCT* buffer which is appended with *Young-Objects* and *Major-New-Objects* buffers. Each object that has a positive *rc* field or is marked as *Roots* is promoted. Otherwise it is collected by the Collect procedure.

```
Procedure Trace(o: Object)
begin
1.     if o.color = white then
2.         o.color := black
3.         if o.LogPointer = NULL then // if not dirty
4.             temp := copy(o) // get a replica
5.             // is still not dirty?
6.             if o.LogPointer = NULL then
7.                 for each slot s of temp do
8.                     v := read(s)
*9.                    if ¬v.old then
10.                        Trace(v)
11.                return
12.        // object is dirty, thus has a logged and committed replica
13.        for each slot s in the replica of o at o.LogPointer do
14.            v := read(s)
*15.           if ¬v.old then
16.                Trace(v)
end
```

Figure 12. **Trace**

```
Procedure Add-Potential-IGPs-To-Roots
begin
       // add the potential inter-generational-pointers to the Roots set.
*1.    $Roots := Roots \cup Updates$
*2.    $Roots := Roots \cup PrevIGP\text{-}Buffer$
*3.    $PrevIGP\text{-}Buffer := \emptyset$
end
```

Figure 13. **Add-Potential-IGPs-To-Roots**

Procedure Collect (figure 17) deletes a garbage object. As this procedure also handles objects recently created during the recent young collection cycle, it must promote the surviving objects. The old bit is cleared for each object which is reclaimed and is raised for each object which survives the collection.

6. An Implementation for Java

We have implemented all three generational collectors into the Jikes RVM - a Research Java Virtual Machine version 2.0.3 (upon Linux Red-Hat 7.2). The entire system, including

```
Procedure **Sweep**
**begin**
**1.**    foreach object *swept* in $Young\text{-}Objects$ do
**2.**       if *swept*.color = $white$ then
**3.**          *swept*.color := *blue*
**4.**          if *swept*.LogPointer != $NULL$ then
**5.**             invalidate swept's log entry for next cycle
**6.**          *swept*.old := **false**
**7.**          return *swept* to the allocator
**8.**       else
***9.**         *swept*.loggedToMajorBuffers := **true**
***10.**        *swept*.old := **true**
***11.**        $Major\text{-}New\text{-}Objects := Major\text{-}New\text{-}Objects \cup \{swept\}$
          // Log o and its promoted(old) slots to the $Major\text{-}Updates$ buffer
***12.**  foreach object's *o* replica in *Updates* do
***13.**     *o*.loggedToMajorBuffers := **true**
***14.**     $Major\text{-}Updates := Major\text{-}Updates \cup address\text{-}of\{o\}$
***15.**     foreach slot *s* in *o*'s replica in *Updates* do
***16.**        $Major\text{-}Updates := Major\text{-}Updates \cup \{s\}$
**end**
```

Figure 14. **Sweep**

the collector itself is written in Java (extended with unsafe primitives available only to the Java Virtual Machine implementation to access raw memory). The Jikes RVM uses a polling mechanism: 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. The check is executed at safe points, when garbage collection may occur. This design simplifies garbage collection. In addition, rather than implementing Java threads as operating system threads, the Jikes RVM multiplexes Java threads on *virtual-processors*, implemented as operating-system threads. The Jikes RVM establishes one virtual processor for each physical processor. Jikes RVM's locking mechanisms are implemented without operating system support. This decision (not to map Java threads to operating system threads directly) was motivated by the need to be able to support rapid thread switching and garbage collection. The only operating-system service that is used by the Jikes RVM is a periodic timer interrupt provided by some system call. Jikes's locking mechanisms make no system calls, i.e. they are implemented without using any operating system services.

The Jikes RVM boot image contains various objects and precompiled classes necessary for booting the Jikes RVM, including the compiler, class loader and other essential elements of the virtual machine. None of the Jikes RVM collectors collect boot image objects. Bacon et al [6] have this same limitation in their reference counting algorithm. Our algorithms follow the Jikes RVM collectors and do not collect boot image objects. Note, there is no overhead in our algorithms for not reclaiming dead boot image objects. The tracing collection will never reach

```
Procedure Update-Reference-Counters
begin
*1.   $Updates := Updates \cup Major\text{-}Updates$
*2.   $Young\text{-}Objects := Young\text{-}Objects \cup Major\text{-}New\text{-}Objects$
3.    for each object $o$ whose replica $r$ in $Updates$ do
          // decrement previous values of the object $o$
4.        for each slot $s$ in the replica of $r$ do
5.            $previous\text{-}value$ := read(s)
6.            $previous\text{-}value$.rc−−
      // increment current values of the object $o$
7.    for each object o in $Updates \cup Young\text{-}Objects$ do
8.    object-is-logged:
9.        if $o$.LogPointer != NULL then
10.           $new\text{-}replica := o$.LogPointer
11.           for each slot $s$ in $new\text{-}replica$ of $o$ do
12.               $curr$ := read($s$)
13.               $curr$.rc++
14.       else
15.           $temp\text{-}replica$ := copy($o$)
16.           if $o$.LogPointer = NULL then
              // the taken replica temp-replica is valid
17.               for each slot $s$ in temp-replica of $o$ do
18.                   $curr$ := read($s$)
19.                   $curr$.rc++
20.           else
21.               goto object-is-logged
end
```

Figure 15. **Update-Reference-Counters**

```
Procedure Reclaim-Garbage
begin
1.    $ZCT := ZCT \cup Young\text{-}Objects$
2.    $Young\text{-}Objects := \oslash$
3.    for each object $o \in ZCT$ do
4.        if $o$.rc $> 0 \vee o \in Roots$ then
5.            $ZCT := ZCT - \{o\}$
*6.           $o$.old := true
7.    for each object $o \in ZCT$ do
*8.       $o$.old := false
*9.       Collect($o$)
*10.  Clear-All loggedToMajorBuffers flags
end
```

Figure 16. **Reclaim-Garbage**

```
Procedure Collect(o : Object)
begin
1.      // object-is-logged:
2.      if o.LogPointer != NULL then
*3.         Invalidate-Log-Entry(o.LogPointer)
4.          replica := o.LogPointer
5.          for each slot s in replica of o do
6.              curr := read(s)
7.              curr.rc−−
8.              if curr.rc = 0 then
9.                  if $curr \notin Roots$ then
10.                     Collect(curr)
*11.                else
*12.                    curr.old := true
13.     else
14.         temp-replica := copy(o)
15.         if o.LogPointer = NULL then
            // the taken replica temp-replica is valid
16.             for each slot s in temp-replica of o do
17.                 curr := read(s)
18.                 curr.rc−−
19.                 if curr.rc = 0 then
20.                     if $curr \notin Roots$ then
21.                         Collect(curr)
*22.                    else
*23.                        curr.old := true
24.         else
25.             goto Line 5.4 object-is-logged
*26.    o.old := false
27.     return o to the general purpose allocator.
end
```

Figure 17. **Collect**

such objects and the reference counting collection buffers will not contain logging of these objects, thus it will not deal with updating the reference counts of the dead objects' children.

6.1. Allocator

The Jikes RVM comes with several collector implementations that use four types of allocators: copying, non-copying, generational-copying, generational non-copying. In our implementation we decided to use the non-copying allocator. This allocator is similar to the one used in [6] and builds on the allocator of Boehm, Demers and Shenker [11]. This allocator fits collectors that do not move objects. It keeps the fragmentation low and allows both efficient sporadic reclamation of objects (as required by the reference counting) and efficient linear reclamation of objects, as required by the sweep procedure.

6.2. Triggering

Each virtual processor has a collector thread associated with it. Garbage collection is triggered explicitly when a mutator explicitly requests it or when a mutator makes a request for space that the allocator cannot satisfy. A full heap collection will be triggered when the amount of available memory drops below a predefined threshold. A minor heap collection will be triggered after every 200 new allocator-block allocations. This kind of triggering strategy emulates allocations from a young generation whose size is limited. During mutation, all collector threads are in a waiting state. When a collection is requested, the collector threads are notified and scheduled on their virtual processors, one at a time. When a collector thread starts executing a handshake, it disables threads switching on its virtual processor till the end of the handshake.

6.3. Fewer handshakes

The handshakes mechanism of Levanoni-Petrank can be simplified when using the Jikes RVM platform. In their algorithm presentation Levanoni-Petrank show that the second handshake (Reinforce-Clearing-Conflict-Set) must be separated from the Mark-Roots (which is their fourth handshake) in the view of all the mutators to keep the correctness of the algorithm. To that purpose, they add a third additional handshake between. The handshake is empty, i.e. no action is taken during it. The reason for this handshake addition is that a thread can fall behind a sibling thread by at most one handshake. Thus, threads that have responded to the fourth handshake will not be interfered with by operations carried out by threads during the clearing or reinforcement stages (which is the second handshake), i.e. threads that still have not responded to the third handshake. The Jikes RVM design introduces the notion of an Uninterruptible interface. The intent of Uninterruptible is to instruct the compiler to omit yieldpoints anywhere in the methods of the class (in the prologues and on the backedges). A yieldpoint is the only location where a thread switch can happen, thus only at yieldpoints can a mutator get control over the collector. By declaring the handshakes method's class uninterruptible we get the third handshake for free. The needed separation is ensured by the uninterruptible principle: no mutator can get the control while an uninterruptible method is being executed by the collector, thus there can not be a write barrier update which will last from before the Mark-Roots and till after it, because there is separation of the Mark-Roots uninterruptible method. Thus, in our algorithm presentation and implementation we have used only three handshakes and yet preserved the correctness of our algorithms, thus simplifying the original Levanoni-Petrank handshake mechanism.

6.4. Changes from the conference version

The results presented in the following section differ from the results in our conference version [5]. The changes are more quantitative than qualitative and the reason for the change is many implementation optimizations applied to the basic sliding views reference counting collector used for the full collections and to the sliding views tracing collector used with the young generation. The same optimizations were applied to the stand-alone Levanoni-Petrank reference counting collector with which we compare.

Changes include, for example, saving scanning an object's descendants. In the original implementation, a copy of the object was taken if it was not dirty, so that if it gets dirty during the scan, we still work on a stable copy of the object. Instead, we now work at the pointer level. After reading each pointer to a local variable, we check if the object is still not dirty. In the vast majority of the checks, the object's dirty bit is not modified during the scan of the object and thus making the copy of the object may be spared. In the seldom case that the object does get dirty while scanning, we move to scan the next pointer from the logged copy of that object. Other changes include improvements to the use of bitmaps (mark bits, reference counts, etc.), a change to the order of operations to improve locality, and other implementation improvements.

7. Results

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 we used were the SPECjvm98 benchmark suite and the SPECjbb2000 benchmark. These benchmarks are described in detail in SPEC's Web site [42]. We tested our new collectors against the Jikes RVM concurrent collector distributed with the Jikes RVM package reported by Bacon et al [6]. The Jikes RVM concurrent collector is an advanced on-the-fly pure reference-counting collector and it is also an on-the-fly collector in which the mutators are only very loosely synchronized with the collector, allowing very low pause times. We also compared the generational collector against the Levanoni-Petrank reference counting collector, from which the generational collector was constructed. Finally, we discuss the generational behavior of the different benchmarks (7.5) and the tuning of parameters (7.6), namely we discuss the chosen size of the young generation. We stress that the measurements are somewhat different from the conference version as explained in Section 6.4 above.

7.1. 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 final measurement is the average of all these five runs. To get an additional multithreaded benchmark, we have also modified the _227_mtrt benchmark from the SPECjvm98 suite to run with a varying number of threads. We measured its run with 2, 4, 6, 8 and 10 threads.

In order to understand better the behavior of these collectors under tight and relaxed conditions, we tested them 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, as reported in the graphs.

Experience led us to choose 200 allocator-blocks as the amount of block allocations needed in order to trigger minor collection, where block size is 16KBytes, i.e. the size of the young generation is about 3MBytes (200 multiply 16KBytes). This was best for most benchmarks. Also, a good triggering for initiating a major (full) collection was when the heap's live objects

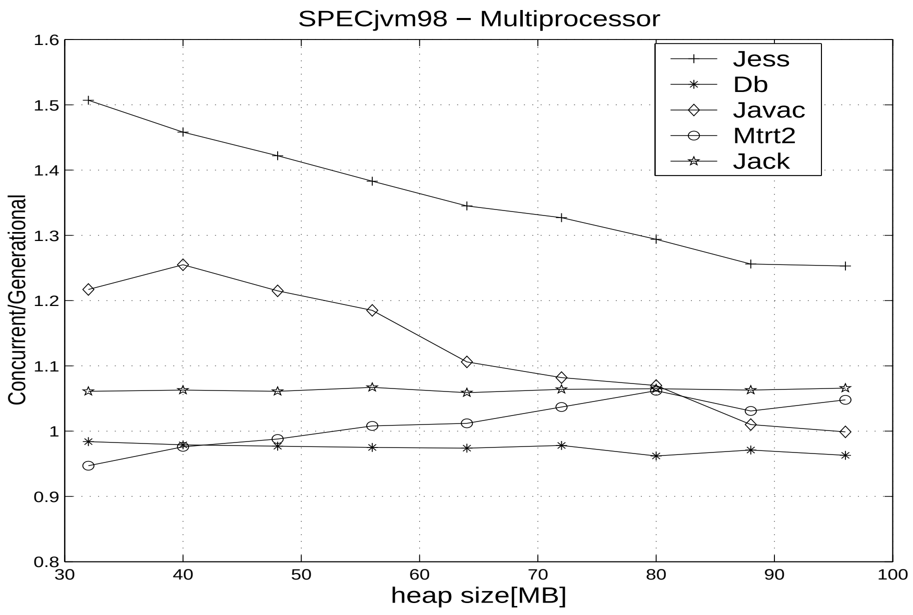

Figure 18. Running time ratios (Jikes-Concurrent/Generational) for the SPECjvm98 suite with varying heap sizes. Results on a multiprocessor

filled 75% of it. Otherwise, the collector may not be able to free space before the program threads consume the whole heap and then the throughput and latency deteriorate.

7.2. Server measurements

7.2.1. The SPECjvm98 benchmarks

The SPECjvm98 benchmarks (and so also the _227_mtrt modified benchmark) provide a measure of the elapsed running time which we report. We report in figure 18 the running time ratio of our collector and the Jikes RVM concurrent collector. The higher the number, the better our collector performs. In particular, a value above 1 means our collector outperforms the Jikes RVM concurrent collector.

We ran each of the SPECjvm98 benchmarks on a multiprocessor, allowing a designated processor to run the collector thread. We report these results in figure 18. These results demonstrate performance when the system is not busy and the collector may run concurrently on an idle processor. In practically all measurements, our collector did better than the Jikes RVM concurrent collector, with an improvement of up to 50% for _202_jess on small heaps.

The behavior of the collector on a busy system may be tested when the number of application threads exceeds the number of (physical) processors. A special case is when the JVM is run on

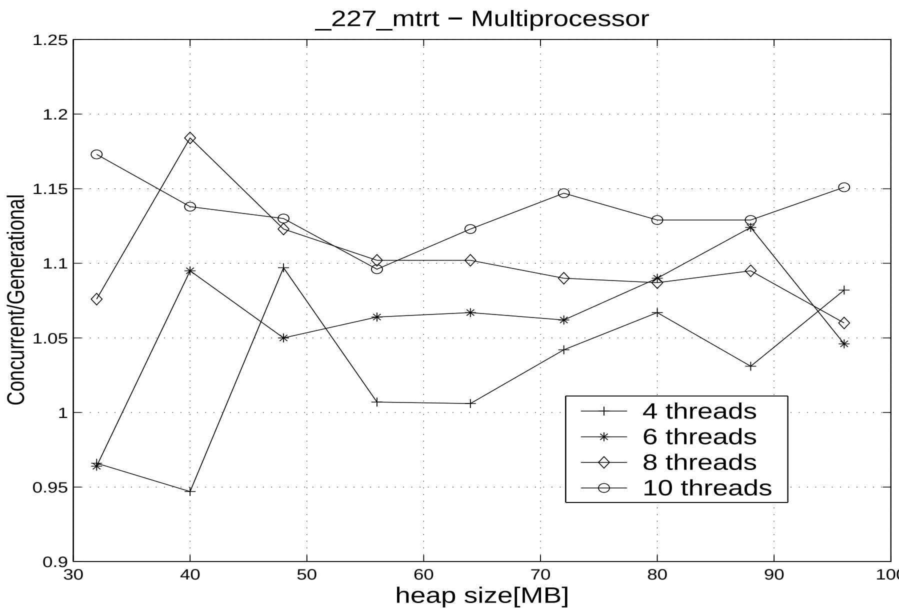

Figure 19. Running time ratio (Jikes-Concurrent/Generational) for the _227_mtrt benchmarks on a multiprocessor.

a uniprocessor. In these cases, the efficiency of the collector is important: the throughput may be harmed when the collector spends a lot of CPU time. A similar effect is obtained with the modified _227_mtrt benchmark when run with 4, 6, 8, and 10 threads on a 4-way multiprocessor. The throughput measurements for the modified _227_mtrt benchmark are reported in figure 19. The measurements show an improved performance for almost all parameters with typical to large heaps, with the highest improvement being 18% for _227_mtrt with 8 threads and heap size 40MBytes.

7.2.2. The SPECjbb2000 benchmark

The results of SPECjbb2000 are measured a bit differently. The run of SPECjbb2000 requires multi-phased run with increasing number of threads. Each phase lasts for two minutes with a ramp-up period of half a minute before each phase. Again, we report the ratio between the results of the collectors. Here the result is throughput and not running-time. For clarity of representation, we report the inverse ratio, i.e., the throughput of the generational collector divided by the throughput of the original collector, so that higher ratios show better performance of our collector, and ratios larger than 1 imply our collector outperforming the Jikes RVM concurrent collector. The measurements are reported for a varying number of threads (and varying heap sizes) in figures 20-21. When the system has no idle processor for

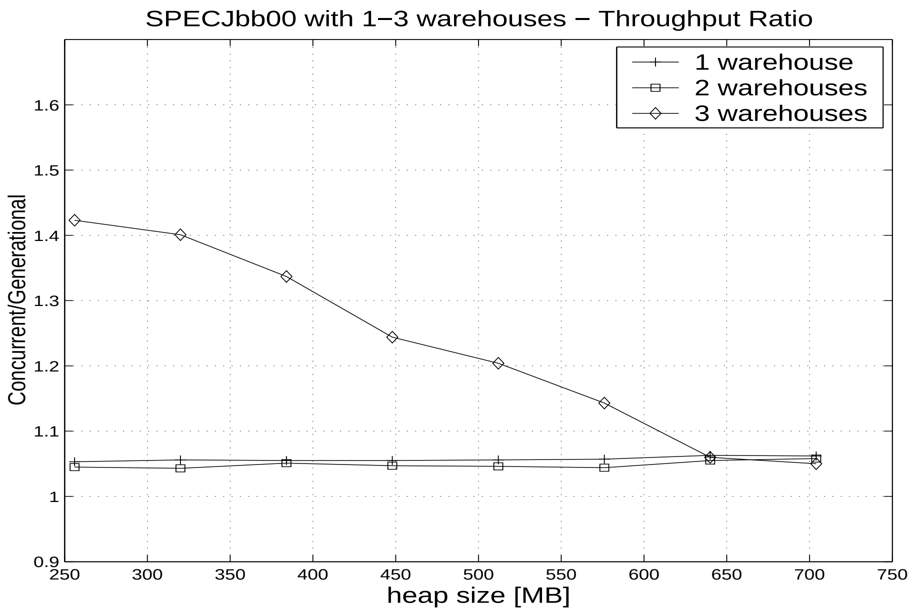

Figure 20. SPEC_jbb2000 throughput ratios (Generational/Jikes-Concurrent) with 1-3 warehouses on a multiprocessor

the collector (4-8 warehouses), our collector clearly outperforms the Jikes RVM concurrent collector. The typical improvement is 50% and the highest improvement is 67%. In case of 1-3 warehouses, where the collector is free to run on an idle processor, our collector also performs better, especially with 3 warehouses. The general better performance may be attributed to the efficient write barrier. With three warehouses and tight heaps, the Jikes RVM concurrent collector is not fast enough to reclaim garbage and so the mutators sometimes are stalled trying to allocate an exhausted heap and must wait until the collector finishes and frees some space. This seldom happens with our fast generational collector.

7.2.3. Pause times

The maximum pause times for the SPECjvm98 benchmarks and the SPECjbb2000 benchmark are reported in table I. The SPECjvm98 benchmarks were run with heap size 64MBytes and those of SPECjbb2000 (with 1,2,3 threads) with heap size 256MBytes. Note that if the number of threads exceed the number of processors, then long pause times appear because threads lose the CPU to other mutators or the collector. Hence the reported settings. It can be seen that the maximum pause times (see table I) are as low as those of the Jikes RVM concurrent collector and they are all below 5ms. The pause times measured in our algorithm as well as in

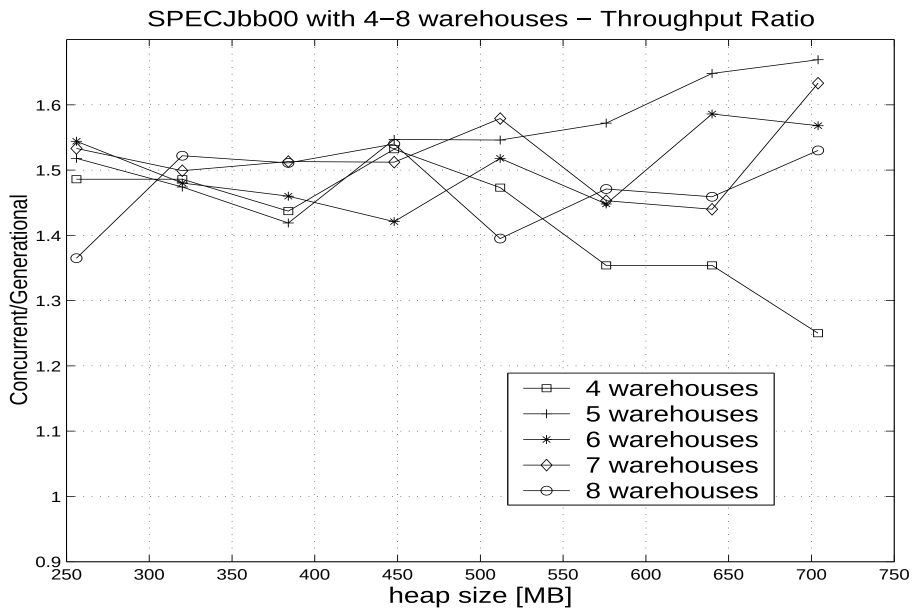

Figure 21. SPEC_jbb2000 throughput ratios (Generational/Jikes-Concurrent) with 4-8 warehouses on a multiprocessor

Table I. Max pause time (msec) measurements for SPECjvm98 and SPECjbb2000 benchmarks on a multiprocessor. SPECjbb2000 was measured with 1, 2, and 3 warehouses.

Collector	jess	javac	db	mtrt	jack	jbb-1	jbb-2	jbb-3
Generational	2.6	3.2	1.3	1.8	2.2	2.3	3.5	4.2
Jikes-Concurrent	2.7	2.8	1.8	1.8	1.6	2.3	3.1	5.5

Bacon et al [6] were not sensitive to changes in heap sizes as long as the collectors satisfy the application's allocation requests.

7.3. Client measurements

As a sanity check, we have also measured our generational collector on a uniprocessor to check how it handles a client environment with the SPECjvm98 benchmark suite (the specifications of the uniprocessor configuration appear in Section 6 above). Both collectors were initially designed for a multiprocessor and are thus not best on this platform. Thus, these results should only be taken as a sanity check. The results are reported in figure 22. It turns out that

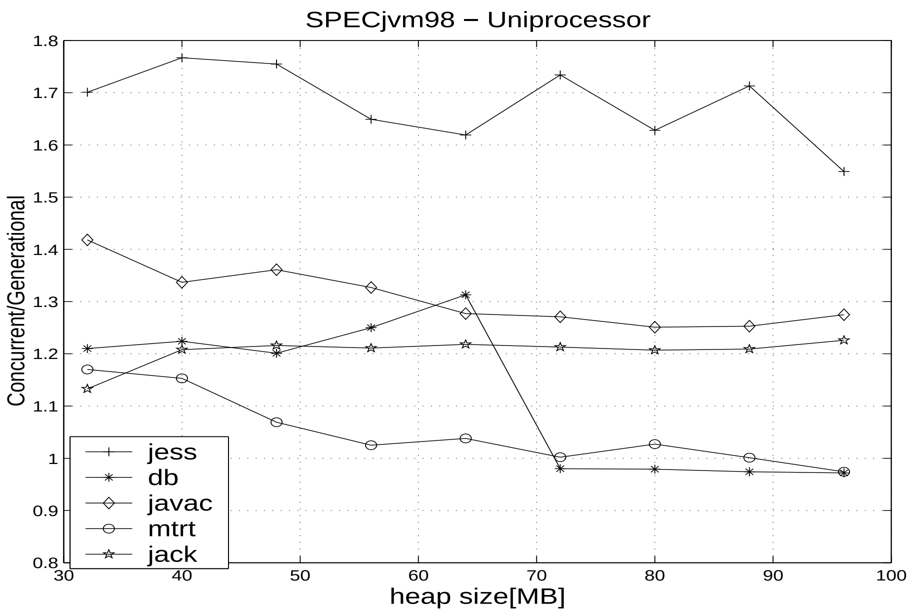

Figure 22. Running time ratios (Jikes-Concurrent/Generational) for the SPECjvm98 suite with varying heap sizes. Results for a uniprocessor.

the generational algorithm is better than the Jikes RVM concurrent collector in almost all tests. Note the large improvement of around 70% for the _202_jess benchmark.

7.4. Results against the original algorithm of Levanoni-Petrank

In this section we present the improvement of the *generational* algorithm throughput against the original *on-the-fly reference-counting* algorithm of Levanoni-Petrank. Namely, we compare the generational collector to the building block from which it was constructed. Our main multithreaded benchmark is the SPECjbb2000 benchmark. It can be seen in Figures 23 and 24 that the generational collector improves the throughput of the benchmark substantially, demonstrating its effectiveness. When used with the small-scale single threaded SPECjvm98 benchmarks, as can be seen from figures 25 and 26, there is no substantial difference between the collectors (less than 3%). With the multithreaded (yet, small scale) mtrt benchmark, generations win with large heaps and lose with tight ones. This shows the benefit of using generations with reference counting collectors.

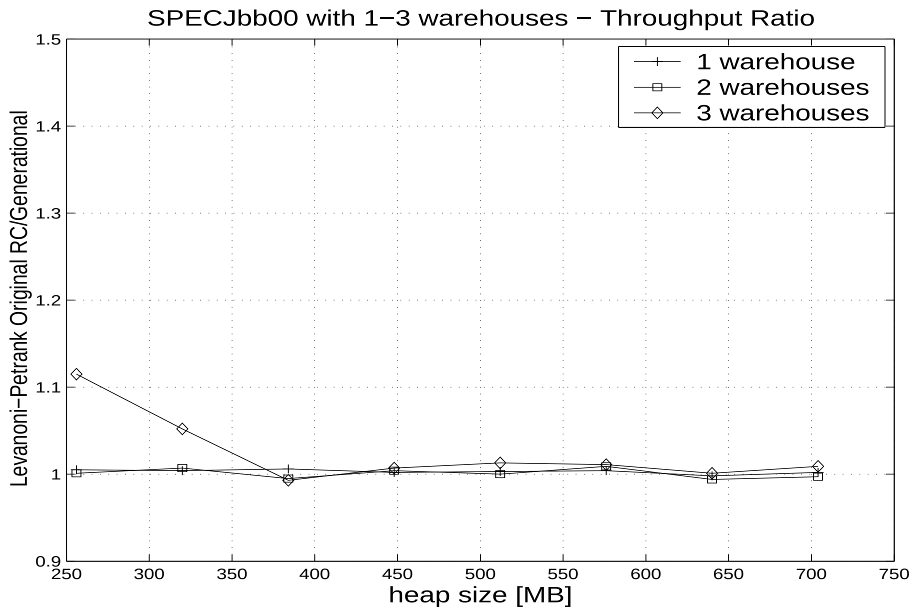

Figure 23. Throughput ratio (Generational/Levanoni-Petrank-Original-RC) for the SPECjbb2000 benchmark with 1-3 warehouses on a multiprocessor.

Table II. Number of collections and their types at the original reference counting algorithm of Levanoni and Petrank with a typical heap size of 64 megabytes

	Levanoni and Petrank Original RC Collector	
Benchmark	RC	Tracing
_201_compress	5	5
_202_jess	10	4
_209_db	5	10
_213_javac	1	1
_222_mpegaudio	3	1
_227_mtrt	6	3
_228_jack	30	2

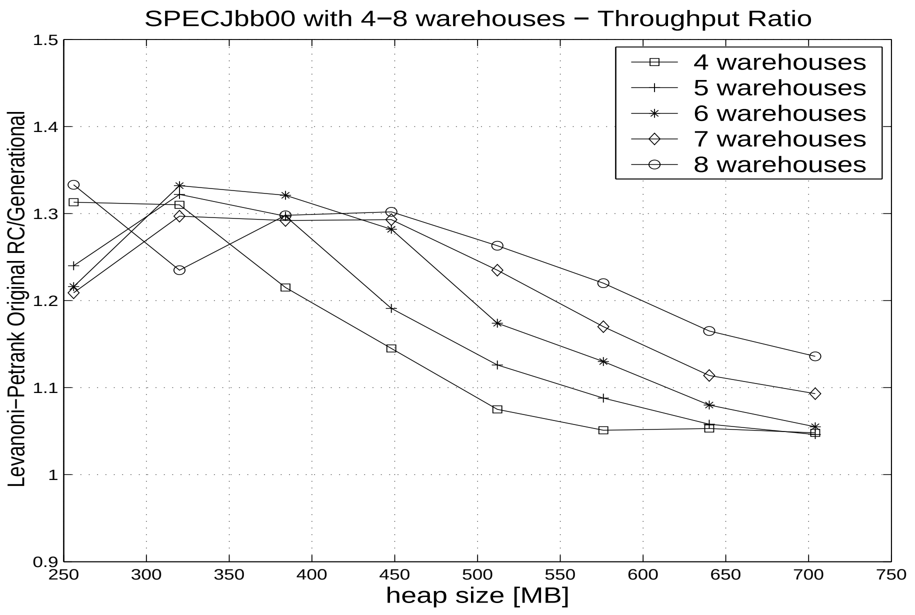

Figure 24. Throughput ratio (Generational/Levanoni-Petrank-Original-RC) for the SPECjbb2000 benchmark with 4-8 warehouses on a multiprocessor.

7.5. The generational behavior of the benchmarks

We now measure benchmark properties that reflect their generational behavior. In particular, we measure how many objects are scanned during the collection, how many of them are scanned due to inter-generational pointers and what percentage of the objects are promoted. The complement of this number is the averaged percentage of the objects which are freed in every collection cycle (minor or major cycle).

Tables III and IV present the average of the real inter-generational pointers percentage (from all inter-generational candidate pointers) in each collection (minor as well as major) in a full run of each benchmark. Our candidate list consist mostly of modified objects in the old generation. Of-course, not all of them actually include inter-generational pointers. As can be seen in table III, for _201_compress, _202_jess, _213_javac, _222_mpegaudio, and _228_jack more than 20% of the total potential pointers scanned during the search for inter-generational pointers were actual inter-generational (for most heap sizes). To conclude from all benchmarks results a percentage result of around 20% is relatively good and is achieved by most benchmarks for almost all heap sizes. As the heap size increases the inter-generational pointer hit rate decreases probably due to the larger time interval between two collection cycles that lets more entries enter the potential inter-generational buffers. That can be a reason for

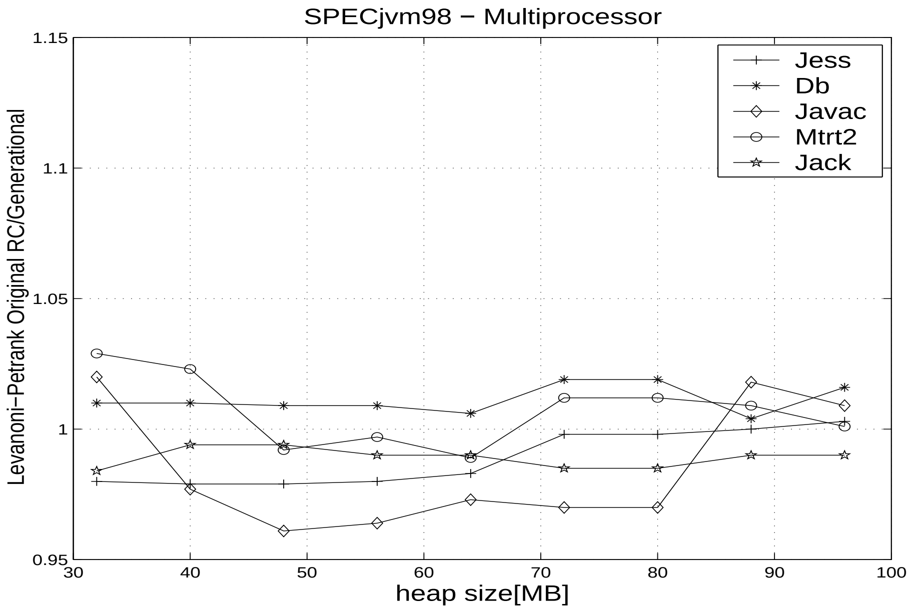

Figure 25. Running time ratios (Levanoni-Petrank-Original-RC/Generational) for the SPECjvm98 suite with varying heap sizes. Results on a multiprocessor

Table III. The Percentage of actual inter-generational pointers among all inter-generational candidates scanned, as a function of varying heap size

Heap Size	_201 _compress	_202 _jess	_213 _javac	_222 _mpegaudio	_209 _db	_227 _mtrt	_228 _jack
32M	36.31%	37.12%	30.09%	47.40%	10.31%	11.56%	28.47%
40M	36.30%	25.78%	25.13%	47.44%	12.15%	12.90%	25.29%
48M	36.31%	21.95%	20.92%	47.44%	12.18%	16.37%	22.61%
56M	36.30%	19.08%	42.88%	47.41%	12.18%	9.47%	19.86%
64M	36.31%	19.91%	20.45%	47.41%	12.19%	9.22%	19.25%
72M	36.30%	18.31%	19.62%	47.44%	12.20%	7.95%	18.34%
80M	36.29%	17.45%	15.78%	47.44%	12.23%	7.07%	18.51%
88M	36.31%	17.29%	17.00%	47.40%	12.31%	6.89%	18.28%
96M	36.31%	17.57%	17.00%	47.40%	12.18%	8.11%	19.07%

the relatively deterioration of several benchmarks throughput as the heap size increases (see for example figure 18).

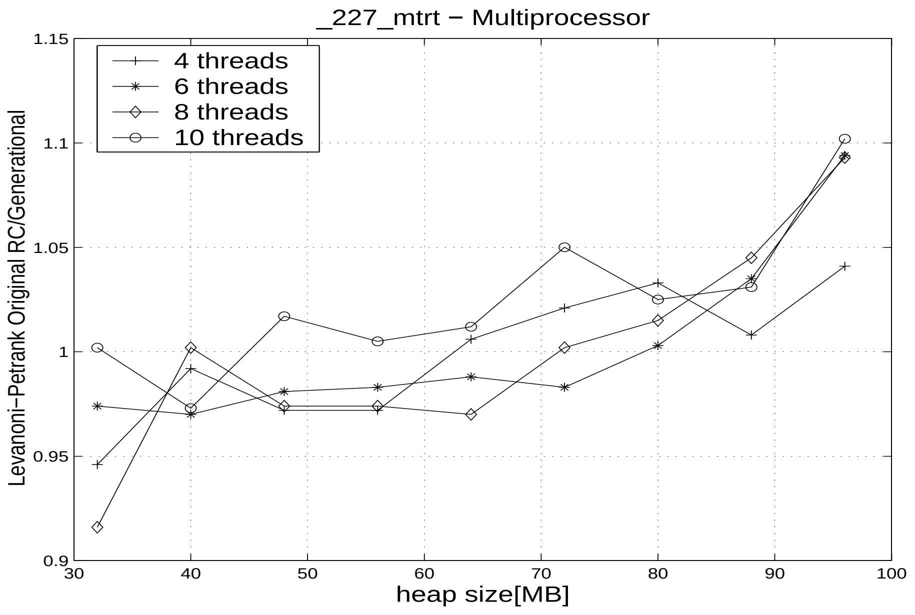

Figure 26. Running time ratio (Levanoni-Petrank-Original-RC/Generational) for the _227_mtrt benchmarks on a multiprocessor.

Table IV. The Percentage of actual inter-generational pointers among all inter-generational candidates scanned for _227_mtrt - multiple threads, as a function of varying heap size

Heap Size	4 threads	6 threads	8 threads	10 threads
32M	12%	12.2%	12%	12%
40M	16.50%	17%	12%	12.2%
48M	11%	12%	16%	12.7%
56M	11%	10%	6.3%	15.2%
64M	14%	9%	7%	6.7%
72M	11.5%	8.5%	6.5%	8.6%
80M	10%	6%	7.6%	7.7%
88M	9%	6.6%	6.3%	6.5%
96M	10%	6.7%	5.2%	7.5%

Table V. The percentage of the promoted objects with respect to all newly created objects at each collection cycle (minor or major) with varying heap size

Heap Size	_201 _compress	_202 _jess	_213 _javac	_222 _mpegaudio	_209 _db	_227 _mtrt	_228 _jack
32M	73.33%	25.47%	63.77%	29.16%	30.63%	92.58%	20.99%
40M	73.32%	22.89%	51.91%	29.12%	9.59%	78.94%	14.54%
48M	73.33%	18.48%	43.56%	29.12%	9.59%	59.29%	10.841%
56M	73.32%	9.55%	36.02%	29.12%	9.59%	45.96%	8.958%
64M	73.33%	6.93%	32.21%	29.16%	9.59%	40.10%	12.29%
72M	73.32%	1.15%	30.53%	29.12%	9.59%	36.60%	8.37%
80M	73.30%	1.04%	29.25%	29.17%	9.59%	28.75%	6.19%
88M	73.32%	1.03%	33.74%	29.16%	9.59%	27.28%	1.97%
96M	73.33%	1.03%	27.61%	29.16%	9.59%	24.06%	1.92%

Table VI. The percentage of the promoted objects with respect to all newly created objects at each collection cycle (minor or major) with _227_mtrt - multiple threads, with varying heap size

Heap Size	4 threads	6 threads	8 threads	10 threads
32M	73%	83%	84%	86%
40M	56%	70%	84%	56%
48M	44%	59%	63%	56%
56M	39%	56%	36%	54%
64M	28%	37%	34%	44%
72M	15%	28%	27%	36%
80M	9%	18%	20%	30%
88M	9%	22%	24%	20%
96M	9%	13%	20%	23%

Tables V and VI present the averaged percentage of objects which are promoted in each collection (minor or major) in a full benchmark run. As can be seen in table V, objects do not tend to die young in _201_compress and in _227_mtrt (70% and 40% promotion percentage, respectively). The benchmark _201_compress is not very interesting from a memory management point of view. The benchmark _227_mtrt has a unique allocation behavior. It starts by allocating a large volume of objects that are kept until the end of the program. After that the program has a generational behavior, with most allocated objects tending to die young. More on the behavior of SPECjvm98 benchmarks may be found in [19]. More on various parameters of _227_mtrt including a non-standard variation of the number of threads appear in table VI. This measurement can give some intuition on the best choice of the young generation size (see section 7.6). With most of the other applications almost all objects die young with most of the heap sizes tried.

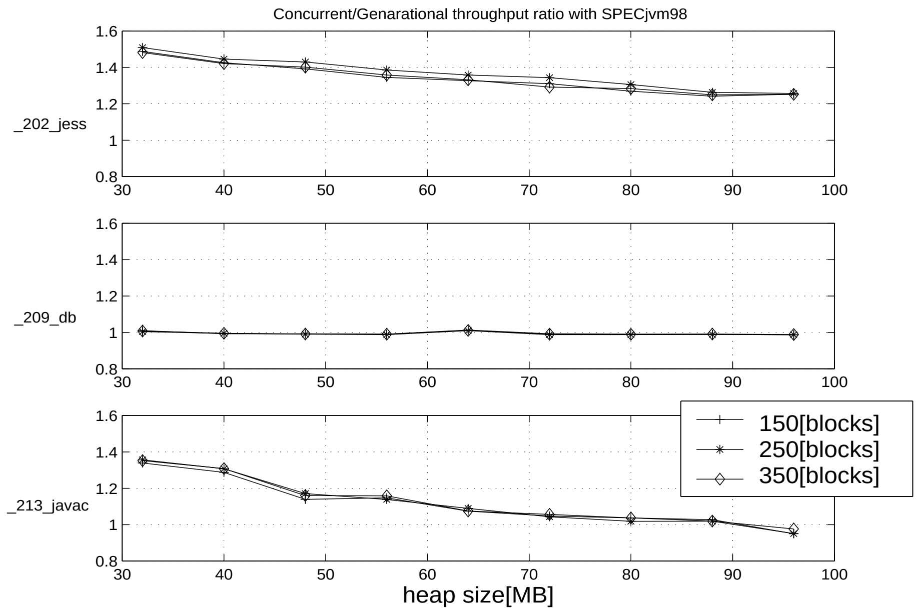

Figure 27. Running time ratios (Jikes-Concurrent/Generational) of SPEC_jvm98 on a multiprocessor with varying minor generation size.

7.6. Tuning the size of the young generation

Finally, we compare various sizes for the young generation in figures 27-29. We compare the sizes of 150, 250, and 350 blocks as possible alternatives for the size of the young generation. Our allocator block size is 16Kbytes, thus the possible alternatives for the size of the young generation are 2.5, 4, and 6 megabytes. The results do not point a single best size for all benchmarks, and do not show a major difference between the running times. Some benchmarks are slightly better with larger young generation, i.e. trigger of minor collection for every 350 allocator blocks allocation while other benchmarks are better with smaller young generation, i.e. trigger of minor collection for every 150 allocator blocks allocation . We have concluded that the best averaged performance (over all benchmarks) is obtained by taking a compromise value for the young generation size of about 3.5 megabytes or 200 allocator blocks. We used this size of young generation for all runs.

8. Conclusions

We have presented a design and implementation of a generational reference counting collector. Our building blocks were the sliding views garbage collectors and the resulting collector is on-

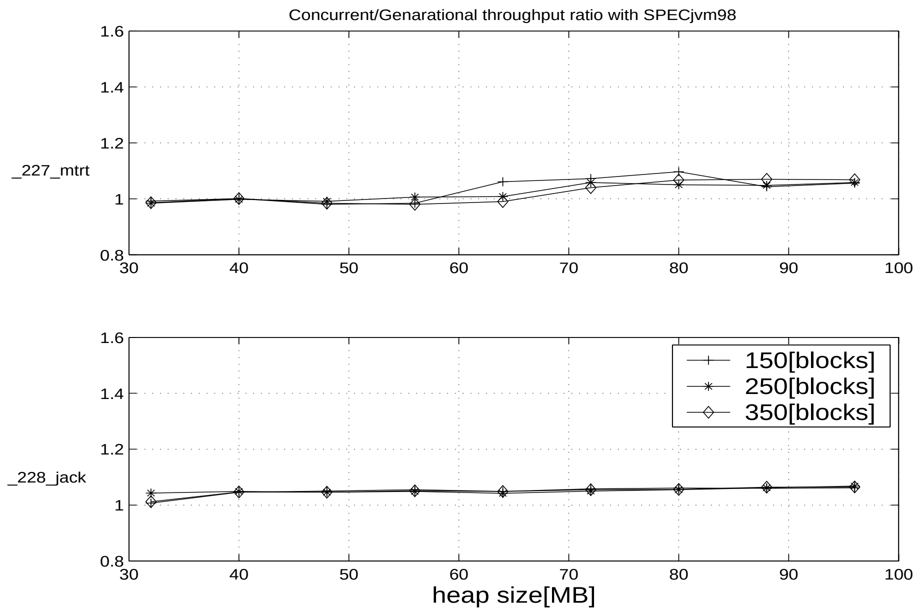

Figure 28. Running time ratios (Jikes-Concurrent/Generational) of SPEC_jvm98 on a multiprocessor with varying minor generation size.

the-fly with pause times of several milliseconds. We have used reference counting for the full collection and tracing for the young generation. This choice of collectors matches the typical high percentage of dead objects in the young generation and low percentage of dead objects in the old generation.

The collector was implemented on the Jikes RVM and was run on a 4-way IBM Netfinity server. Measurements against the Jikes RVM concurrent collector and against the original algorithm of Levanoni and Petrank show a large improvement in throughput and the same low pause times.

9. Acknowledgments

We thank Harel Paz for useful discussions and for his help in the measurements. We thank the anonymous referees for their most useful remarks.

REFERENCES

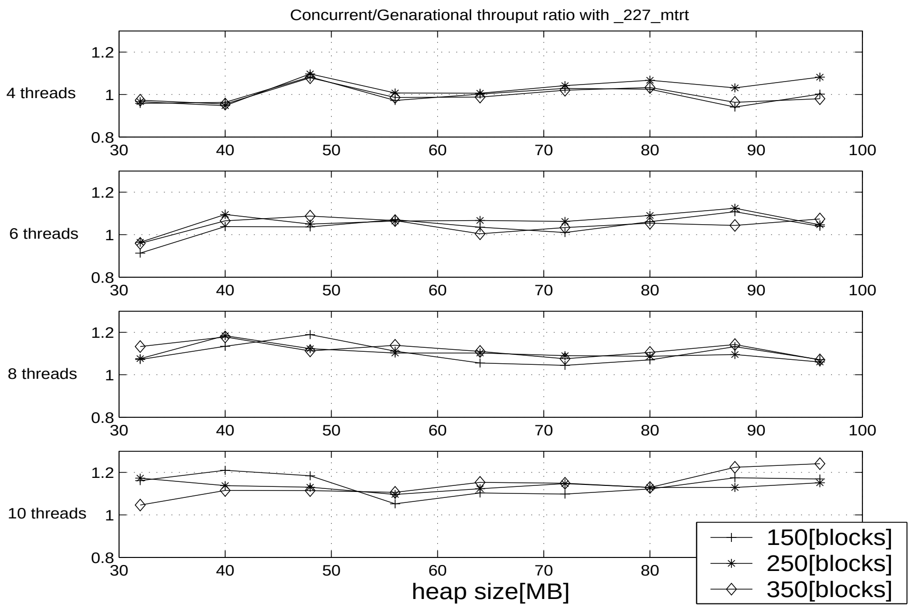

Figure 29. Running time ratios (Jikes-Concurrent/Generational) of _227_mtrt on a multiprocessor with varying minor generation size.

1. Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger. *The AWK Programming Language.* Addison-Wesley, 1988.
2. 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.
3. Andrew W. Appel, John R. Ellis, and Kai Li. Real-time concurrent collection on stock multiprocessors. *ACM SIGPLAN Notices*, 23(7):11–20, 1988.
4. Hezi Azatchi, Yossi Levanoni, Harel Paz, and Erez Petrank. An on-the-fly mark and sweep garbage collector based on sliding view. In OOPSLA [35].
5. Hezi Azatchi and Erez Petrank. Integrating generations with advanced reference counting garbage collectors. In *Proceedings of the Compiler Construction: 12th International Conference on Compiler Construction, CC 2003*, volume 2622 of *Lecture Notes in Computer Science*, pages 185 – 199, Warsaw, Poland, May 2003. Springer-Verlag Heidelberg.
6. 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.
7. 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.
8. Henry G. Baker. Minimising reference count updating with deferred and anchored pointers for functional data structures. *ACM SIGPLAN Notices*, 29(9), September 1994.
9. Katherine Barabash, Yoav Ossia, and Erez Petrank. Mostly concurrent garbage collection revisited. In OOPSLA [35].

10. Stephen M. Blackburn and Kathryn S. McKinley. Ulterior reference counting: Fast garbage collection without a long wait. In OOPSLA [35].
11. Hans-Juergen Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. *ACM SIGPLAN Notices*, 26(6):157–164, 1991.
12. Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, 1988.
13. C. J. Cheney. A non-recursive list compacting algorithm. *Communications of the ACM*, 13(11):677–8, November 1970.
14. George E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 3(12):655–657, December 1960.
15. Alan Demers, Mark Weiser, Barry Hayes, Daniel G. Bobrow, and Scott Shenker. Combining generational and conservative garbage collection: Framework and implementations. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, pages 261–269, San Francisco, CA, January 1990. ACM Press.
16. John DeTreville. Experience with concurrent garbage collectors for Modula-2+. Technical Report 64, DEC Systems Research Center, Palo Alto, CA, August 1990.
17. John DeTreville. Experience with garbage collection for modula-2+ in the topaz environment. In Eric Jul and Niels-Christian Juul, editors, *OOPSLA/ECOOP '90 Workshop on Garbage Collection in Object-Oriented Systems*, Ottawa, October 1990.
18. L. Peter Deutsch and Daniel G. Bobrow. An efficient incremental automatic garbage collector. *Communications of the ACM*, 19(9):522–526, September 1976.
19. Sylvia Dieckmann and Urs Hölzle. A study of the allocation behaviour of the SPECjvm98 Java benchmarks. In Eric Jul, editor, *Proceedings of 12th European Conference on Object-Oriented Programming, ECOOP98*, volume 1445 of *Lecture Notes in Computer Science*, pages 92–115, Brussels, July 1998. Springer-Verlag.
20. 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.
21. Robert Dimpsey, Rajiv Arora, and Kean Kuiper. Java server performance: A case study of building efficient, scalable JVMs. *IBM Systems Journal*, 39(1):151–174, 2000.
22. Damien Doligez and Georges Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *Conference Record of the Twenty-first Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, Portland, OR, January 1994. ACM Press.
23. 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.
24. 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.
25. Tamar Domani, Elliot Kolodner, and Erez Petrank. A generational on-the-fly garbage collector for Java. Technical Report 88.385, IBM Haifa Research Laboratory, 2000. Fuller version of [24].
26. Shinichi Furusou, Satoshi Matsuoka, and Akinori Yonezawa. Parallel conservative garbage collection with fast allocation. In Paul R. Wilson and Barry Hayes, editors, *OOPSLA/ECOOP '91 Workshop on Garbage Collection in Object-Oriented Systems, Addendum to OOPSLA'91 Proceedings*, October 1991.
27. Adele Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.
28. Richard L. Hudson and J. Eliot B. Moss. Sapphire: copying garbage collection without stopping the world. *Concurrency and Computation: Practice and Experience*, 15(3-5):223–261, 2003.
29. Richard E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.* Wiley, Chichester, July 1996. With a chapter on Distributed Garbage Collection by R. Lins.
30. Yossi Levanoni and Erez Petrank. A scalable reference counting garbage collector. Technical Report CS–0967, Technion — Israel Institute of Technology, Haifa, Israel, November 1999.
31. 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.
32. Henry Lieberman and Carl E. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, 1983. Also report TM–184, Laboratory for Computer Science, MIT, Cambridge, MA, July 1980 and AI Lab Memo 569, 1981.

33. John McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3:184–195, 1960.
34. Scott M. Nettles and James W. O'Toole. Real-time replication-based garbage collection. In *Proceedings of SIGPLAN'93 Conference on Programming Languages Design and Implementation*, volume 28(6) of *ACM SIGPLAN Notices*, Carnegie Mellon University, USA, June 1993. ACM Press.
35. *OOPSLA'03 ACM Conference on Object-Oriented Systems, Languages and Applications*, ACM SIGPLAN Notices, Anaheim, CA, November 2003. ACM Press.
36. Yoav Ossia, Ori Ben-Yitzhak, Irit Goft, Elliot K. Kolodner, Victor Leikehman, and Avi Owshanko. A parallel, incremental and concurrent GC for servers. In *Proceedings of SIGPLAN 2002 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, pages 129–140, Berlin, June 2002. ACM Press.
37. James W. O'Toole and Scott M. Nettles. Concurrent replicating garbage collection. Technical Report MIT–LCS–TR–570 and CMU–CS–93–138, MIT and CMU, 1993. Also LFP94 and OOPSLA93 Workshop on Memory Management and Garbage Collection.
38. Young G. Park and Benjamin Goldberg. Static analysis for optimising reference counting. *Information Processing Letters*, 55(4):229–234, August 1995.
39. Manoj Plakal and Charles N. Fischer. Concurrent garbage collection using program slices on multithreaded processors. In 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.
40. Tony Printezis. *Management of Long-Running High-Performance Persistent Object Stores.* PhD thesis, University of Glasgow, May 2000.
41. David J. Roth and David S. Wise. One-bit counts between unique and sticky. In Richard Jones, editor, *ISMM'98 Proceedings of the First International Symposium on Memory Management*, volume 34(3) of *ACM SIGPLAN Notices*, pages 49–56, Vancouver, October 1998. ACM Press.
42. SPEC Benchmarks. Standard Performance Evaluation Corporation. http://www.spec.org/, 1998,2000.
43. Guy L. Steele. Multiprocessing compactifying garbage collection. *Communications of the ACM*, 18(9):495–508, September 1975.
44. Guy L. Steele. Corrigendum: Multiprocessing compactifying garbage collection. *Communications of the ACM*, 19(6):354, June 1976.
45. David M. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices*, 19(5):157–167, April 1984. Also published as ACM Software Engineering Notes 9, 3 (May 1984) — Proceedings of the ACM/SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, 157–167, April 1984.
46. Larry Wall and Randal L. Schwartz. *Programming Perl.* O'Reilly and Associates, Inc., 1991.
47. J. Weizenbaum. Symmetric list processor. *Communications of the ACM*, 6(9):524–544, September 1963.
48. Paul R. Wilson. Uniprocessor garbage collection techniques. In Yves Bekkers and Jacques Cohen, editors, *Proceedings of International Workshop on Memory Management*, volume 637 of *Lecture Notes in Computer Science*, University of Texas, USA, 16–18 September 1992. Springer-Verlag.
49. Paul R. Wilson. Uniprocessor garbage collection techniques. Technical report, University of Texas, January 1994. Expanded version of the IWMM92 paper.
50. David S. Wise. Stop-and-copy and one-bit reference counting. Technical Report 360, Indiana University, Computer Science Department, March 1993.
51. Taichi Yuasa. Real-time garbage collection on general-purpose machines. *Journal of Software and Systems*, 11(3):181–198, 1990.