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
hezia@cs.technion.ac.il
[2] Dept. of Computer Science,
Technion - Israel Institute of Technology,
Haifa 32000, Israel
erez@cs.technion.ac.il

Abstract. We study an incorporation of generations into a modern reference counting collector. We start with the two on-the-fly collectors suggested by Levanoni and Petrank: a reference counting collector and a tracing (mark and sweep) collector. We then propose three designs for combining them so that the reference counting collector collects the young generation or the old generation or both. Our designs maintain the good properties of the Levanoni-Petrank collector. In particular, it is adequate for multithreaded environment and a multiprocessor platform, and it has an efficient write barrier with no synchronization operations. To the best of our knowledge, the use of generations with reference counting has not been tried before.
We have implemented these algorithms with the Jikes JVM and compared them against the concurrent reference counting collector supplied with the Jikes package. As expected, the best combination is the one that lets the tracing collector work on the young generation (where most objects die) and the reference counting work on the old generation (where many objects survive). Matching the expected survival rate with the nature of the collector yields a large improvement in throughput while maintaining the pause times around a couple of milliseconds.

Keywords: Runtime systems, Memory management, Garbage collection, Generational Garbage Collection.

1 Introduction

Automatic memory management is well acknowledged as an important tool for fast development of large reliable software. It turns out that the garbage collection process has an important impact on the overall runtime performance. Thus, clever design of efficient memory management and garbage collection is an important goal in today's technology.

1.1 Reference counting

Reference counting is the most intuitive method for automatic storage management known since the sixties (c.f. [11].) The main idea is that we 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.

Reference counting seems very promising to future garbage collected systems, especially with the spread of the 64 bit architectures and the increase in usage of very large heaps. Tracing collectors must traverse all live objects, and thus, the bigger the usage of the heap (i.e., the amount of live objects in the heap), the more work the collector must perform. Reference counting is different. The amount of work is proportional to the amount of work done by the user program between collections plus the amount of space that is actually reclaimed. But it does not depend on the space consumed by live objects in the heap.

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 [27, 5, 23] building on and improving on previous work. Bacon et al. [5] and Levanoni and Petrank [23] following DeTreville [12] 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).

1.2 Automatic memory management on a multiprocessor

In this work, we concentrate on garbage collection for multiprocessor machines. Multiprocessor platforms have become quite standard for server machines and are also beginning to gain popularity as high performance desktop machines. 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 (the so-called *stop-the-world* concept). This causes bad processor utilization, and hinders scalability.

In order to make better use of a multiprocessor, concurrent collectors have been presented and studied (see for example, [7, 30, 15, 4, 12, 9, 17, 25, 16, 28]). A concurrent collector is a collector that does most of its collection work concurrently with the program without stopping the program threads. Most of the 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 in a halt is short. On-the-fly collectors [15, 17, 16, 18, 19, 5, 23] never stop 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.

1.3 Generational collection

Generational garbage collection was introduced by Lieberman and Hewitt [24], and the first published implementation was by Ungar [33]. Generational garbage collectors rely on the assumption that many 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 several 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).

Since in this paper we discuss an on-the-fly collector, we do not expect to see reduction of the pause time: they are extremely low already. Our goal is to keep the low pauses of the original algorithm. However, increased efficiency and better locality may give us a better overall collection time and a better throughput. This is indeed what we achieve.

1.4 This work

In this work, we study how generational collection interacts with reference counting. Furthermore, we employ a modern reference counting algorithm adequate for running on a modern environment (i.e., multithreaded) and modern platform (i.e., multiprocessor). We study three alternative uses of reference counting with generations. In the first, both the young and the old generations are collected using reference counting. In the second, the young generation is collected via reference counting and the collector of the old generation is a mark-and-sweep collector. The last alternative we explore is a use of reference counting to collect the old generation and mark-and-sweep to collect the young generation. As building blocks, we use the Levanoni-Petrank sliding view collectors [23]: the reference counting collector and the mark-and-sweep collector. Our new generational collectors are on-the-fly and employ a write barrier that uses no synchronization operation (like the original collectors).

Note that one combination is expected to win the race. Normally, the percent of objects that survive is small in the young generation and high in the old generation. Looking at the complexity of the involved algorithms - reference counting has complexity related to the number of dead objects. Thus, it matches the death rate of the old generation. Tracing collectors do better when most objects die - thus, they match the death rate of the young generation. Indeed the combination employing tracing for the young generation and reference counting for the old yields the best results.

A second combination that did well is the one that uses reference counting to collect the young generation and tracing for the full heap collection. The advantage of this combination is that it is not important to keep the reference count accurate. Using a simple promotion policy in which any surviving young object is promoted, it is enough to know during the young collection if a young object has reference count zero or higher. Inaccuracy in the exact value of the reference counting will not affect future collections, since the object is later promoted and handled by the tracing collector. Thus, we can eliminate some of the reference counting work on the young objects on one hand, and on the other hand, spare the tracing collector of many short-lived objects that are collected by the reference counting collector.

In addition to the new study of generations with reference counting, our work is also interesting as yet another attempt to run generations with an on-the-fly collector. 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 [19][3].

1.5 Generational collection without moving objects

Usually, on-the-fly garbage collectors do not move objects; the cost of moving objects while running concurrently with program threads is too high. Demers, et al. [2] 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.

1.6 Implementation and results

We have implemented our algorithm on Jikes - a Research Java Virtual Machine 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[31].

In Section 6 we report the measurements we ran with our collectors. We tested our new collectors against the Jikes concurrent collector distributed with the Jikes Research Java Virtual Machine package. This collector is a reference counting concurrent collector developed at IBM and reported by Bacon et al. [5]. Our most efficient collector (the one that uses reference counting for the old generation) achieves excellent performance measures. The throughput is improved by up to 40% for the SPECjbb2000 benchmark. The pauses are also smaller. These results hold for the default heap size of the benchmarks. Running the collectors on tight heaps show that our generational collector is not suitable for very small heaps. In such conditions, the original Jikes algorithm performs better.

[3] A partial incorporation of generations with an on-the-fly collector, used only for immutable objects was used by Doligez, Leroy and Gonthier [17, 16]. 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.

1.7 Cycle collection

A major disadvantage of reference counting is that it does not collect cycles. In case the old generation is collected with a mark-and-sweep collector, there is no issue, since the cycles will be collected then. When reference counting is used for the old generation we also use the mark-and-sweep collector seldom to collect the full heap and reclaim garbage cycles.

1.8 Organization

In Section 2 we review reference counting developments through recent years and mention related work. In section 3 we present the Levanoni-Petrank collectors we build on. In section 4 we present the generational algorithms. In section 5 we discuss our implementation and in section 6 we present our measurements. We conclude in section 7.

2 An overview on reference counting algorithms

The traditional method of reference counting, was first developed for Lisp by Collins [11]. 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 into 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 predecessors are then decremented as well and the reclamation may continue recursively.

This simple method was later used in SmallTalk-80 [21], the AWK [3] and Perl [34] programs. Improvements to the naive algorithm were suggested in several subsequent papers. Weizman [35] studied ameliorating the delay introduced by recursive deletion. Several works [29, 37] use a single bit for each reference counter with a mechanism to handle overflows. The idea being that most objects are singly-referenced, except for the duration of short transitions.

Deutsch and Bobrow [14] noted that most of the overhead on counter updates originates from the frequent updates of local references (in stack and registers). They suggested to use 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. To collect objects, a collection is invoked. During the collection one can reclaim all objects with zero heap reference count that are not accessible from local references. Their 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+ [12]. Further study on reducing work for local variables can be found in [8] and [26].

Reference counting seemed to have an intrinsic problem with multithreading. First, the updates on the counters are done concurrently by all the threads and so the updates must be atomic. Using a synchronization operation in the write barrier seems to be too costly. 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 [12]. 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 [27] proposed a collection method based on reference counting for architectures that support explicit multi-threading on the processor level. Bacon et al. [5] 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. They use a compare and swap to do the actual update, and locally log the transaction in a local transaction log. The main idea is that if the decrements of the counters are not immediately executed, but are delayed to the next collection cycle, then correctness is insured (at a cost of some floating garbage).

Concurrently with Bacon et. al., Levanoni and Petrank [23] 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 with no need for synchronization. Using their write barrier, a program race would result in multiple copies of transactions in the write barrier rather then 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. Measurements have shown that this improvement reduces the number of logs and counter updates by a factor of 100-1000 for standard Java benchmarks.

3 The Levanoni-Petrank collectors

In this section we provide a short overview of the Levanoni-Petrank collectors. We adopt a convention of adding an asterisk to any line in the code that differs from the original collector. The full algorithm is described in the original paper [23].

3.1 The sliding-view reference counting algorithm

The Levanoni-Petrank collectors [23] are based on computing differences between heap snapshots. The algorithms operate in cycles. A cycle begins with a collection and ends with another. Let us describe the collector actions during cycle k. Using a write barrier, the mutators record all heap objects whose pointer slots are modified during cycle k. 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. A dirty flag is used to let only one record be kept for any modified object. The analysis shows that (seldom) races may cause more than one record be created for an object, but all such records contain essentially 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 figure 1 (ignore the lines with asterisk in their numbers).

All created objects are marked dirty during creation (figure 2). There is no need to record their slots 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 4-6). A sliding-view is essentially a non-atomic snapshot of the heap. It is obtained incrementally, i.e. the mutators are not stopped simultaneously. A snooping mechanism is used to ensure that the sliding view of the heap does not confuse the collector into reclaiming live objects: while the view is being read from the heap, the write-barrier mark any object that is assigned a new reference 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.

Getting further into the details, the Levanoni-Petrank collector employs four handshakes (see figures: 4-7) during the collection cycle. 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 (figure 4). During the first handshake, mutator local buffers are retrieved and then are cleared. The objects which are listed in the buffers are exactly those objects that have been changed since the last cycle. Next, the dirty flags of the objects listed in the buffers are cleared while the mutators are running (figure 5). This step may clear dirty marks that have been concurrently set by the running mutators. The logging in the threads' local buffers is being used in order to keep these dirty bits set in the second handshake (figure 6). The third handshake is carried out to assure the reinforcement is visible to all mutators. During the fourth handshake threads local states are scanned and objects directly reachable from the roots are marked as *Roots* (figure 7).

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 10). Each object which is logged to one of the mutator's local buffers was modified since the previous collection cycle, thus we need to decrement the *rc* of its slots values 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' replica at the retrieved local buffers. Each object replica contains the object slots' value at the time the previous sliding-view was taken.

The *rc* increment operation of each modified object is more complicated as the mutators can change the current sliding-view values of the object's slots while the collector tries to increment their *rc* field. This race is solved by taking a replica of the object to be adjusted and committing it. First, we check if the object's dirty flag, *o.LogPointer*, is set. If it is set it points already to a committed replica (taken by some mutator) of the object's slots at the time the current sliding-view was taken. Otherwise, we take a temporary replica of the object and commit it by checking afterwards that the object's dirty flag is still not set. If it is committed the replica contains the object's slots value at the time the current sliding-view was taken and can be used to increment the *rc* of the object's slots value. Otherwise, if the dirty flag is set, we use the replica pointed by the set dirty flag in order to adjust the *rc* of the object's slots.

A collection cycle ends with reclamation which recursively free any object with zero *rc* field which is not marked as *local* (figure 11).

3.2 The sliding-view tracing algorithm

"Snapshot at the beginning" [20] mark&sweep collectors exploit the fact that a garbage object remains a garbage until the collector recycles it. i.e., being garbage is a stable property. The Levanoni-Petrank sliding-view tracing collector takes the idea of "Snapshot at the beginning" one logical step further and show how it is possible to trace and sweep given a "sliding view at the beginning". The collector computes a sliding-view exactly as in the previous reference counting algorithm (see figures: 4-7). After the Mark-Roots stage, the collector starts tracing according to the sliding view associated with the cycle (figures 7, 8). 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 committed 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 9).

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. 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. 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.

4 The Generational Collectors

In this section we describe the collectors we have designed. There was one winning collector whose performance outweigh the other two. It is described in section 4.1 below. We now go on with discussing issues that are common to all three algorithms.

Our generational mechanism 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 algorithms we use. Generations are not segregated in the heap since we do not move objects in the heap. In order to quickly determine 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.

Recall that we are using the Levanoni-Petrank sliding view collectors as the basis for this work. The sliding view 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 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. So the write barrier remains (almost) the same as in the original algorithm. The write barrier appears in Figure 1. The slight modification is discussed in Subsection 4.1 below.

One issue to notice is that while we are collecting, it is possible that the address of an object to be reclaimed appears in the *Updates* buffer. This may happen since the collector is run concurrently with the mutators and the mutators are never stopped all at once for the collection. When this rare case happens, we have decided to invalidate the logged object such as the next collection will not deal with its adjustments. After invalidating the object log entry we continue to reclaim it.

4.1 Reference counting for the full collection

Here, we describe the algorithm that worked best: using reference counting for the full collections and tracing (mark-and-sweep) for the minor collections. Later, we provide an overview of the two other algorithms designed in this work: the one that uses reference counting only for the minor collections and the one that uses reference counting for both the minor and the full collections. Although this algorithm was doing best, the other two algorithms generally outweigh the original algorithm of Levanoni-Petrank ([23]) and the reference collector of Bacon et. al ([5]) in almost all of the used benchmarks, too. We use the same simple promotion policy as was discussed in the beginning of this section. Let us fully explain the minor and the full collections.

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 and were logged by each mutator i to its local $Young\text{-}Objects_i$ buffer. These local buffers hold addresses of all newly created objects (created since the previous 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 $Young\text{-}Objects$. This $Young\text{-}Objects$ buffer is the young generation to be processed in this minor collection cycle by the collector.

Recall that we are using the Levanoni-Petrank sliding view collectors as the basis for this work. The sliding view algorithm uses a dirty flag for each object to tell if it was modified since the previous collection. All modified objects are kept in a *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 previous 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. At first glance it may appear that this is enough. However, the collection cycle is not atomic in the view of the program. It runs concurrently with the run of the program. Thus, referring to the time of the last collection cycle is not accurate. During the following discussion, we assume that the reader is familiar with the Levanoni-Petrank [23] original collectors. 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 seen the first handshake executes an update operation 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 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.

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 has 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* 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.

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

Finally, we note that the sweep phase processes only young objects. It scans each object's color in the $Young\text{-}Buffer$, which is actually the young generation that is processed by this minor collection. Objects which are marked with *white* color are reclaimed, otherwise, they are promoted by setting their *old* flag as true.

The full (reference counting) collection. The $Major\text{-}Young\text{-}Objects$ and $Major-$ *Updates* buffers are full collection buffers that correspond to the minor collection's $Young\text{-}Objects$ and *Updates* buffers. These buffers are prepared by the minor collections to serve the full collection. Only those objects which were promoted by the minor collections should be logged to the major buffers as these objects will live till the next full collection. The minor collection avoids repetition in these buffers using an additional bitmap called $LoggedToMajorBuffers$. Other than the special care required with the buffers, the major collection cycle is similar to the original reference counting collector besides. The *rc* field adjustments are executed for each modified object. Each modified objects is essentially logged to $Major\text{-}Updates$ buffer or to the *Updates* buffer. The *rc* of the object's previous sliding-view slots values is decremented and the object's current sliding-view slots values *rc* is incremented. As for young objects, the same procedure needs only to increment the *rc* fields of the current sliding-view slots values for each young object, thus, logged to $Major\text{-}Young\text{-}Objects$ or $Young\text{-}Objects$. No decrement operation should be taken on the *rc* field of *Young-Objects* objects slots because their object did not exist in the previous collection cycle and was created only afterwards and their value then was null.

Using deferred reference counting ([23] following [14]), 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* as well as in the *Major-New-Objects* buffer are appended to the ZCT that is reclaimed by the collector.

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

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

Fig. 1. Mutator code: **Update Operation [RC for full]**

4.2 Reference counting for the young generation

We continue with an overview of the second generational algorithm. Using reference counting for the minor collections and mark-and-sweep for the full collections. That algorithm was doing well too. It has different advantages with respect to the previous algorithm.

The use of reference counting in the young generation and mark&sweep(tracing) in the full generation gives an interesting advantage. All multithreaded reference counting collectors, and in particular the Levanoni-Petrank collector, let the collector do all

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

Fig. 2. Mutator code: **Allocation Operation**

Procedure **Collection-Cycle**
begin
Initiate-Collection-Cycle
Clear-Dirty-Marks
Reinforce-Clearing-Conflict-Set
Mark-Roots
if ($majorCollection$) then
    Update-Reference-Counters
    Reclaim-Garbage
else
    Mark
    Sweep
end

Fig. 3. Collection Cycle [RC for full]

Procedure **Initiate-Collection-Cycle**
begin
for each thread T_i do
    $Snoop_i$:= **true**
* $LogIGP_i$:= **true**
for each thread T_i do
    suspend thread T_i
    // copy (without duplicates).
    $Updates := Updates \cup Updates_i$
    $Updates_i := \oslash$ // clear buffer.
    $Young\text{-}Objects := Young\text{-}Objects \cup Young\text{-}Objects_i$
    $Young\text{-}Objects_i := \oslash$ // clear buffer.
    resume thread T_i
end

Fig. 4. Initiate-Collection-Cycle [RC for full]

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

Fig. 5. Clear-Dirty-Marks

Procedure **Reinforce-Clearing-Conflict-Set**
begin
$ClearingConflictSet := \emptyset$
for each thread T_i do
  suspend thread T_i
  $LogIGP_i :=$ **false**
  $ClearingConflictSet := ClearingConflictSet \cup Updates_i[1 \ldots CurrPos_i - 1]$
  resume thread T_i
for each object $o \in ClearingConflictSet$ do
  o.LogPointer := address of o's replica in $Updates$
end

Fig. 6. Reinforce-Clearing-Conflict-Set [RC for full]

```
Procedure Mark-Roots
begin
1.      black := 1-black
2.      white := 1-white
*3.     PrevIGP-Buffer := PrevIGP-Buffer ∪ IGP-Buffer
*4.     IGP-Buffer := ∅
5.      for each thread T_i do
6.          suspend thread T_i
7.          AllocColor_i := black
8.          Snoop_i := false
9.          Roots := Roots ∪ State_i // copy thread local state.
*10.        IGP-Buffer := IGP-Buffer ∪ IGP-Buffer_i
11.         resume thread T_i
*12.    Roots := Roots ∪ PrevIGP-Buffer
*13.    PrevIGP-Buffer := ∅
14.     for each thread T_i do
            // copy and clear snooped objects set
15.         Roots := Roots ∪ Snooped_i
16.         Snooped_i := ∅
        // Mark objects from Roots
17.     for each object o ∈ Roots do
18.         Trace(o)
end
```

Fig. 7. Mark-Roots [RC for full]

```
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 := 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 logged in this cycle
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
```

Fig. 8. Trace [RC for full]

```
Procedure Sweep
begin
1.     foreach object swept in Young-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.         Major-New-Objects := Major-New-Objects ∪ {swept}
       // Log o and its promoted(old) children
       // addresses to the Major-Updates buffer
*11.   foreach object's o replica in Updates do
*12.      o.loggedToMajorBuffers := true
*13.      Major-Updates := Major-Updates ∪ address-of{o}
*14.      foreach slot s in o's replica in Updates do
*15.         Major-Updates := Major-Updates ∪ {s}
end
```

Fig. 9. Sweep [RC for full]

updates of the reference counts. The program threads provide the collector a list of all modifications. In [23] it is noted that it is enough to provide the collector with a small fraction of this list: Only the first modification of each object o since the previous collection is required. The collector has to execute only two operations for each such object o that was modified since the last collection. For each slot s of o: decrement the reference count of the object that was referenced by s during the previous collection and increment the reference counts of the object that was pointed by s during the current collection. Now, with a young (minor) generational collection, the relevant objects for the collections are objects that did not exist in the previous collection, since all surviving objects have been promoted. Thus, there is no need to decrement the counts of objects that were referenced in the previous collection: they were either promoted or reclaimed. So, we only need to increment the counters and half the work on counter updates is eliminated, see figure 20.

An important issue is the promotion procedure. Promotion at major collections is simple. During the sweep (figure 19) it is clear that unmarked objects are reclaimed and marked objects are promoted. However, reclamation and promotion in reference counting is not as simple (figure 21). We start by going over the objects in the young generation (i.e, the *Young-Objects* buffer) and checking if they have zero counts. Those that have zero counts are reclaimed. But an object whose count is positive cannot yet be promoted. Its count is not final and may still decrease to zero as his ancestors may be later reclaimed. In order to refrain from traversing the young generation again in the end of the counter updates just for promotion, we use the

```
Procedure Update-Reference-Counters
begin
*   Updates := Updates ∪ Major-Updates
*   Young-Objects := Young-Objects ∪ Major-New-Objects
    for each object o whose replica r in Updates do
        // decrement previous values of the object o
        for each slot s in the replica of r do
            previous-value := read(s)
            previous-value.rc−−
    // increment current values of the object o
    for each object o in Updates ∪ Young-Objects do
    object-is-logged:
        if o.LogPointer != NULL then
            new-replica := o.LogPointer
            for each slot s in new-replica of o do
                curr := read(s)
                curr.rc++
        else
            temp-replica := copy(o)
            if o.LogPointer = NULL then
            // the taken replica temp-replica is valid
                for each slot s in temp-replica of o do
                    curr := read(s)
                    curr.rc++
            else
                goto object-is-logged
end
```

Fig. 10. Update-Reference-Counters [RC for full]

```
Procedure Reclaim-Garbage
begin
    ZCT := ZCT ∪ Young-Objects
    Young-Objects := ∅
    for each object o ∈ ZCT do
        if o.rc > 0 ∨ o ∈ Roots then
            ZCT := ZCT − {o}
*           o.old := true
    for each object o ∈ ZCT do
*       o.old := false
*       Collect(o)
end
```

Fig. 11. Reclaim-Garbage [RC for full]

```
Procedure Collect(o: Object)
begin
object-is-logged:
if o.LogPointer != NULL then
    Invalidate-Log-Entry(o.LogPointer)
    replica := o.LogPointer
    for each slot s in replica of o do
        curr := read(s)
        curr.rc−−
        if curr.rc = 0 then
            if curr ∉ Roots then
                Collect(curr)
            else
                curr.old := true
else
    temp-replica := copy(o)
    if o.LogPointer = NULL then
    // the taken replica temp-replica is valid
        for each slot s in temp-replica of o do
            curr := read(s)
            curr.rc−−
            if curr.rc = 0 then
                if curr ∉ Roots then
                    Collect(curr)
                else
                    curr.old := true
    else
        goto object-is-logged
o.old := false
return o to the general purpose allocator.
end
```

Fig. 12. Collect [RC for full]

following promotion mechanism. When an object is found to have a positive rc, it is marked as a candidate for promotion via two flags: the *old* flag, which is the regular flag signifying that the object is old, but also a designated *pendingPromotion* flag. The second flag tells the recursive reclamation procedure that the object is still young for the current collection and can be reclaimed if its count is decremented to zero. The marking of young objects as candidates for promotion takes place at the Reclaim-Garbage procedure. The Reclaim-Garbage procedure sieves the ZCT table so that only objects with zero rc field which are not marked as *Roots* remain in it. Then, it recursively collects young or promotion candidate objects only. Notice that the deleted candidate objects are not young objects as they were marked as old. At the end of a collection cycle all the *pendingPromotion* flags are cleared, thus, promoting all the young surviving objects, see also figure 22.

Note that in this case, we do not worry about collecting cyclic structures. Such structures are collected by the tracing collector in a full collection of the algorithm, (figures 18, 19). In these full collections the collector starts tracing according to the sliding view associated with the cycle. After the marking stage is over the collector proceeds to reclaim garbage objects by sweeping the heap. See section 3.2 for further explanation about the tracing sliding-view algorithm.

```
Procedure Update(obj: Object, offset: int , new: Object)
begin
    if obj.LogPointer = NULL then // object is not dirty
        TempPos := CurrPos
        // take a temporary replica of the object
        foreach field ptr of obj which is not NULL
            Updates_i[TempPos++] := ptr
        if obj.LogPointer = NULL then // commit the replica
            Updates_i[TempPos++] := address-of obj
            CurrPos := TempPos
            // set the dirty flag
            obj.LogPointer := address-of Updates_i[CurrPos]
    write( obj, offset ,new)
    if (Snoop_i and new != NULL) then
        Snooped_i := Snooped_i ∪ {new}
end
```

Fig. 13. Mutator code: **Update Operation [RC for minor, RC for both]**

4.3 Reference counting for both generations

We now go on with a sketch of the last generational algorithm. As mentioned earlier, we only provide a sketch of this design. The last generational algorithm uses the reference counting method in the minor collection as well as in the full collection. The main problem with the young generation here is that it is not clear how to save in the

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

Fig. 14. Initiate-Collection-Cycle [RC for minor, RC for both]

Procedure **Reinforce-Clearing-Conflict-Set**
begin
1. $ClearingConflictSet := \emptyset$
2. for each thread T_i do
3. suspend thread T_i
4. $ClearingConflictSet := ClearingConflictSet \cup Updates_i[1 \dots CurrPos_i - 1]$
5. resume thread T_i
6. for each object $o \in ClearingConflictSet$ do
7. o.LogPointer := address of o's replica in $Updates$
end

Fig. 15. Reinforce-Clearing-Conflict-Set [RC for minor, RC for both]

```
Procedure Mark-Roots
begin
    black := 1-black
    white := 1-white
    for each thread T_i do
        suspend thread T_i
        AllocColor_i := black
        Snoop_i := false
        Roots := Roots ∪ State_i // copy thread local state.
        resume thread T_i
    for each thread T_i do
        // copy and clear snooped objects set
        Roots := Roots ∪ Snooped_i
        Snooped_i := ∅
    // Mark objects from Roots
    for each object o ∈ Roots do
        Trace(o)
end
```

Fig. 16. Mark-Roots [RC for minor, RC for both]

```
Procedure Collection-Cycle
begin
    Initiate-Collection-Cycle
    Clear-Dirty-Marks
    Reinforce-Clearing-Conflict-Set
    Mark-Roots
    if (majorCollection) then
        Mark
        Sweep
    else
        Update-Reference-Counters
        Reclaim-Garbage
        Clear-All pendingPromotion flags
end
```

Fig. 17. Collection Cycle [RC for minor]

```
Procedure Trace(o: Object)
begin
    if o.color = white then
        o.color := black
        if o.LogPointer = NULL then // if not dirty
            temp := o // get a replica
            // is still not dirty?
            if o.LogPointer = NULL then
                for each slot s of temp do
                    v := read(s)
                    Trace(v)
                return
        // object is dirty, thus logged in this cycle
        for each slot s in the replica of o at o.LogPointer do
            v := read(s)
            Trace(v)
end
```

Fig. 18. Trace [RC for minor]

```
Procedure Sweep
begin
    foreach object swept in the heap do
        if swept.color = white then
            swept.color := blue
            if swept.LogPointer != NULL then
                invalidate swept's log entry for next cycle
            swept.old := false
            return swept to the allocator
        else
            // promote object
            swept.old := true
end
```

Fig. 19. Sweep [RC for minor]

```
Procedure Update-Reference-Counters
begin
        // no need to decrement previous values of o
        // increment current values of the object o
        for each object o in Updates ∪ Young-Objects do
        object-is-logged:
           if o.LogPointer != NULL then
              new-replica := o.LogPointer
              for each slot s in new-replica of o do
                 curr := read(s)
                 curr.rc++
           else
              temp-replica := copy(o)
              if o.LogPointer = NULL then
              // the taken replica temp-replica is valid
                 for each slot s in temp-replica of o do
                    curr := read(s)
                    curr.rc++
              else
                 goto object-is-logged
end
```

Fig. 20. Update-Reference-Counters [RC for minor]

```
Procedure Reclaim-Garbage
begin
        ZCT := ZCT ∪ Young-Objects
        Young-Objects := ∅
        for each object o ∈ ZCT do
           if o.rc > 0 ∨ o ∈ Roots then
              ZCT := ZCT − {o}
              o.pendingPromotion := true
        for each object o ∈ ZCT do
           o.old := false
           Collect(o)
end
```

Fig. 21. Reclaim-Garbage [RC for minor]

```
Procedure Collect(o: Object)
begin
    object-is-logged:
    if o.LogPointer != NULL then
        Invalidate-Log-Entry(o.LogPointer)
        replica := o.LogPointer
        for each slot s in replica of o do
            curr := read(s)
            if curr.old = false ∨ curr.pendingPromotion then
                curr.rc−−
                if curr.rc = 0 then
                    if curr ∉ Roots then
                        Collect(curr)
                    else
                        curr.old := true
    else
        temp-replica := copy(o)
        if o.LogPointer = NULL then
        // the taken replica temp-replica is valid
            for each slot s in temp-replica of o do
                curr := read(s)
                if curr.old = false ∨ curr.pendingPromotion then
                    curr.rc−−
                    if curr.rc = 0 then
                        if curr ∉ Roots then
                            Collect(curr)
                        else
                            curr.old := true
        else
            goto object-is-logged
    o.old := false
    return o to the general purpose allocator.
end
```

Fig. 22. : **Collect [RC for minor, RC for both]**

counter updates during the minor collection. If we go over the *Updates* buffer then we might as well do all the counter updates instead of copying the irrelevant entries and executing them later. Another alternative is to use a much heavier write barrier that routes each update according to the ages of the two objects that lose and gain a reference. We have decided that this latter option would be too costly and chose the first. Namely, in the minor collection, the algorithm keeps updating the reference counts of all objects but reclaims only the young dead objects, (figures 24, 25). Old dead objects are reclaimed in a major collection (figure 25). This algorithm seems to have the lowest potential for savings and indeed this concern materializes in our measurements.

In this algorithm we employ two zero count tables in order to store potential dead young and dead old objects, respectively. The promotion policy is the same naive policy as before, and the problem with determining which object is promoted is solved in the same manner as in the first algorithm.

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

```
Procedure Collection-Cycle
begin
      Initiate-Collection-Cycle
      Clear-Dirty-Marks
      Reinforce-Clearing-Conflict-Set
      Mark-Roots
      Update-Reference-Counters
      Reclaim-Garbage(Minor-ZCT)
      if (majorCollection) then
          Reclaim-Garbage(Major-ZCT)
      else
          Clear-All pendingPromotion flags
end
```

Fig. 23. Collection Cycle - RC for both

5 An Implementation for Java

We have implemented all three generational collectors into Jikes - a Research Java Virtual Machine 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 available only to the Java Virtual Machine implementation to access raw memory). Jikes uses *safe−points*: rather than interrupting threads with asynchronous signals, each thread periodically checks a bit in a condition register that indicates that the runtime system wishes to gain control. This design significantly simplifies garbage collection. In

```
Procedure Update-Reference-Counters
begin
        // decrement previous values of the object o
        for each slot s in the replica of r do
            previous-value := read(s)
            previous-value.rc−−
            if previous-value.rc = 0 then
                if previous-value.old = false then
                    Minor-ZCT := Minor-ZCT ∪ {o}
                else
                    Major-ZCT := Major-ZCT ∪ {o}
    // increment current values of the object o
    for each object o in Updates ∪ Young-Objects do
    object-is-logged:
        if o.LogPointer != NULL then
            new-replica := o.LogPointer
            for each slot s in new-replica of o do
                curr := read(s)
                curr.rc++
        else
            temp-replica := copy(o)
            if o.LogPointer = NULL then
            // the taken replica temp-replica is valid
                for each slot s in temp-replica of o do
                    curr := read(s)
                    curr.rc++
            else
                goto object-is-logged
end
```

Fig. 24. Update-Reference-Counters [RC for both]

```
Procedure Reclaim-Garbage( zct : Zero-Count-Table )
begin
    zct := zct ∪ Young-Objects
    Young-Objects := ∅
    for each object o ∈ zct do
        if o.rc > 0 ∨ o ∈ Roots then
            zct := zct − {o}
*           o.pendingPromotion := true
    for each object o ∈ zct do
*       o.old := false
*       Collect(o)
end
```

Fig. 25. Reclaim-Garbage-For-Minor [RC for both]

addition, rather than implementing Java threads as operating system threads, Jikes multiplexes Java threads on $virtual - processors$, implemented as operating-system threads. Jikes establishes one virtual processor for each physical processor. Following we'll discuss attributes in Jalapeño that influenced the design of our collector under this platform.

Jalapeño attributes

Jalapeño Processors and Threads. Rather than implement Java threads as operating system threads, Jalapeño multiplexes Java threads on $virtual - processors$, implemented as operating-system threads. Jalapeño '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 Jalapeño establish one virtual processor for each physical processor. The only operating-system service that is used by Jalapeño is a periodic timer interrupt provided by some system call. Jalapeño 's locking mechanisms make no system calls, i.e. they are implemented without using any operating system services.

Jalapeño compilers. Jalapeño does not interpret bytecodes. Instead these are compiled to machine code before execution. Jalapeño supports two interoperable compilers that address different trade-offs between development time, compile time and run time. These compilers are integral to Jalapeño 's design: they enable thread scheduling, synchronization, type-accurate garbage collection, exception handling, and dynamic class loading. Our generational garbage collection algorithms' implementation uses the baseline compiler instead of its optimizing compiler. Throughout the paper we have assumed that the underlying system conforms to sequential consistency constraints over the memory model. Several optimizations of the optimizing compiler violate this assumption over the underlying system, thus, we have decided to use the baseline compiler.

Object model and memory layout. Values in the Java language are either *primitive* (e.g., int, double, etc.) or they are $references$ (that is, pointers) to objects. Objects are either $arrays$ having components or $scalars$ having fields. Assuming the reference to an object is in a register, the object's fields can be accessed at a fixed displacement in a single instruction. To facilitate array access, the reference to an array points to the first(zeroth) component of an array and the remaining components are laid out in ascending order. The number of components in an array, its length, is kept just before its first component. Jalapeño 's arrays grow up from the object reference (with the array length at a fixed negative offset), while scalar objects grow down from the object reference with all fields at a negative offset. A field access is accomplished with a single instruction using base-displacement addressing.

Object-Headers. A two-word object header is associated with each object. This header supports virtual methods dispatch, dynamic type checking, memory management, synchronization, and hashing. It is located below the value of a reference to the object. One word of the header is a *status* word. The other word of an object header is a reference to the $TypeInformationBlock(TIB)$ for the object's class. A TIB is an array of Java object references. Its first component describes the object's class (including its superclass, the interfaces it implements, offsets of any object reference fields, etc.). The remaining components are compiled method bodies (executable code) for the virtual methods of the class. Thus, the TIB serves as Jalapeño 's virtual method table.

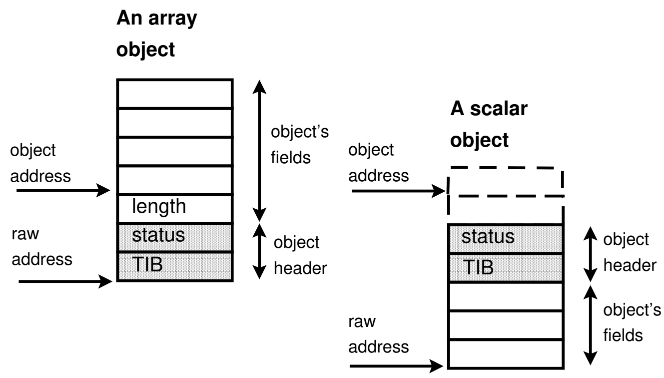

Fig. 26. Jalapeño 's object model

Run-time subsystem. The whole virtual machine cannot be implemented without some low-level constructs that the does not exist in the Java language. The Jalapeño impelements theses constructs and low-level services in a class called: MAGIC. That class implements all types of services needed by the virtual machine implementation in general and to the collector's implementation in particular. Services like - exception handling, dynamic type checking, raw addresses manipulation, I/O, reflection and more are provided through this class.

Our implementation

Bit-Maps. The implementation of the garbage collection algorithms calls out for a mechanism to fast relate objects to sets, i.e. given an object and some set of objects we would like to efficiently (first priority in time, second in space) support four functions:

- Add it to the set.
- Remove it from the set.
- Find out if it is contained in the set.
- Clear all the objects from the set.

We've chosen to implement that mechanism as bitmap tables (1 bit for each 16 bytes) separated from the data (not in the objects' headers). This has enhanced locality of reference as consecutive accesses to objects' sets will be done from a smaller and near set of addresses and will mess with the data cache less than with a solution that uses the object header. An important attribute and an advantage of the bitmap table solution is that the fourth function (Clear all the bitmap table) can be implemented very efficiently in terms of time. This solution is not space efficient, using a bitmap table can be quite wasteful in terms of space: we need to allocate a flag in the bitmap per the granule of object alignment. Since objects are usually aligned on 16 bytes or smaller granules and since a typical object is some 50 bytes long, inlining the flag (as an indicator for a set relation) inside the object results in a substantial saving of space (not to mention the cases in which some unused bit in the object header is waiting to be exploited). In our algorithms we've used several bitmap tables, given an object we would like to know:

- *rc* - the reference count of the object.
- *mark* - the object was marked by the Mark&Sweep part of the algorithm.
- *local* - the object is in the root set of this collection cycle.
- *zct* - the object is in a *Zero-Count-Table.*
- *old* - the object is old, (i.e. the object's age is old).
- $loggedToOldBuffers$ - the object is logged to the buffers of the major generation's.
- $promotionPending$ - this object was temporarily promoted in this collection cycle.

Object-Headers. As was written in the previous section, two-word object header is associated with each object. One word of the header is a *status* word. The other word of an object header is a reference to the $TypeInformationBlock(TIB)$ for the object's class. In our implementation each object has one more word in its header. This word functions as the "Dirty" indication of each object, i.e. whether the object is logged to our garbage collector's auxiliary buffers or not. Locating this indication interspersed with the data has several benefits: conservation of space, since we can allocate space per flags on a per type basis, rather then conservatively for every word of memory, as is done in a bitmapped solution and increased locality of reference, as the flags are accessed by the mutators in conjunction with their perspective slots, (at least in the reference-counting part of our algorithms) and there is no need for the collector to implement the "clear all" operation. This method has disadvantages too, mainly the inability to quickly clear the dirty flags, i.e. they must be cleared one at a time and decreased locality of reference in the Mark&Sweep part of our algorithms as the sweep operation needs to look at that pointer for each object it want to evacuate to see if it is logged to some buffer. However we'll discuss later some technique to overcome that problem.

Another issue is the need for a raw to reference address conversion. In Jalapeño , the object model (i.e. the way the objects with their headers are located in memory) is designed such that the offsets of either scalars and arrays from their headers will be the same. As a side effect from this object model there is no way to know some object's reference address from its raw address given it is not known in advance if it is scalar or array. In the Mark&Sweep part of our algorithm such thing is a need. The Sweep procedure scans all the alive objects in the heap and check if they were marked, the sweep procedure reaches the raw address of each object and in order to find out

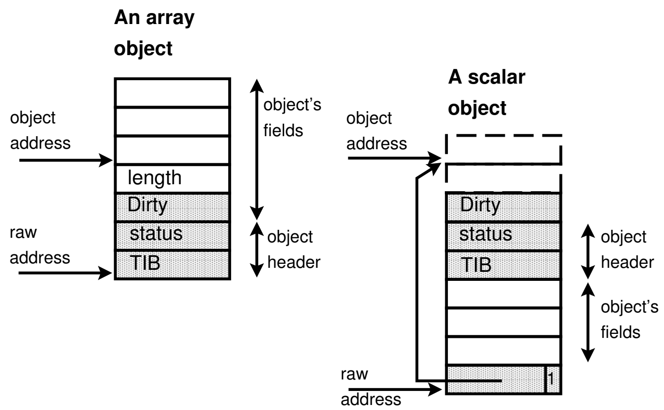

Fig. 27. Our object model

if it is marked or not there is a need to find the reference address that corresponds to the raw address (as the object was marked by its reference address and not by its raw address).

Allocator. Jalapeño comes with several collectors implementations that uses 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 in [5] and builds on the allocator of Boehm Demers and Shenker [9]. This allocator fits for 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. The chosen allocator partitions heap memory into two different heap spaces: $large-object-space$ and $small-object-space$.

Large-Heap. Each manager uses a non-copying large-object space, managed as a sequence of pages. Requests for large objects are satisfied on a first-fit basis. After a garbage collection event, adjacent free pages are coalesced. In order to support concurrent and on-the-fly allocations of small objects by copying managers, each virtual processor maintains a large *chunk* of local space from which objects can be allocated without requiring synchronization. (These local chunks are not logically separate from the global heap: an object allocated by one virtual processor is accessible from any virtual processor that gets a reference to it). Allocation is performed by incrementing a space pointer by the required size and comparing the result to the limit of the local chunk. If the comparison fails the allocator atomically obtains a new local chunk from the shared global space. This technique works without locking unless a new chunk is required. The cost of maintaining local chunks is that memory fragmentation is increased slightly, since each chunk may not filled completely.

Small-Heap. The non-copying allocator divide the small-object heap into fixed-size blocks (of 32KBytes). Each block is dynamically subdivided into fixed-size slots. The number of these sizes and their values, are build-time constants that can be tuned to fit an observed distribution of small-object sizes. When an allocator receives a request for an object, it determines the size of the smallest slot that will satisfy the request, and obtains the current block for that size. To avoid locking overhead, each virtual processor maintains a local current block of each size. If the current block is full (not the normal case), it makes the next block for that size available. If all such blocks are full, it obtains a block from the shared pool and makes the newly obtained block current. Since the block sizes and the number of slot sizes are relatively small, the space impact of replicating the current blocks for each virtual processor is insignificant.

From mutation to collection. 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 it.

Note that when a collector thread is running(performing a handshake), all the mutators of its virtual processor must be at yield points. When the number of mutator threads is large, this could be an important performance consideration. Since all yield points in Jikes are safe points, the collector thread may now proceed with the handshake. After each handshake has completed, the collector threads re-enable thread switching on its virtual processor and then wait for the next collection. Mutator threads of some virtual processor start up automatically as the collector thread release their virtual processor.

6 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[31]. We tested our new collectors against the Jikes concurrent collector distributed with the Jikes Research Java Virtual Machine package. This collector is a reference counting concurrent collector developed at IBM and reported by Bacon et al. [5].

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). To get an additional multithreaded benchmark, we have also modified the _227_mtrt benchmark from the SPECjvm98 suite to run on a varying

Max Pauses[ms] with SPECjvm98 and SPECjbb2000(1-3 threads)								
Collector	jess	javac	db	mtrt	jack	jbb-1	jbb-2	jbb-3
Generational: RC for full	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

Fig. 28. Max pause time measurements for SPECjvm98 and SPECjbb2000 benchmarks on a multiprocessor. SPECjbb2000 was measured with 1, 2, and 3 warehouses.

number of threads. We measured its run with 2, 4, 6, 8 and 10 threads. Finally, 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. In the results, we concentrate on the best collector, i.e., the collector that uses reference-counting for the full collection. 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. this is the size of the young generation. This was best for most benchmarks. Also, it was best to initiate a full collection when the heap was 75% full. 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. In figures 36 - 44 we report all our three collectors results over varying heap size and minor generation size. From these figures it can be concluded that there is no optimal minor generation size for all benchmarks, some of the benchmarks call for small minor generation size (i.e., trigger minor collection for every 150 allocator blocks allocation), see figures 36, 39, 42. Some of the benchmarks call for large minor generation size (i.e., trigger minor collection for every 350 allocator blocks allocation), see figures 37, 40, 43.

The reference collector. We have chosen the concurrent reference counting collector supplied with the Jikes package of Bacon et al [5] as our reference collector. All the results are compared against this collector. The reference collector is an advanced on-the-fly pure reference-counting collector and it has similar characteristics as our collector, namely, the mutators are only very loosely synchronized with the collector, allowing very low pause times.

Server measurements. 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 29 the running time ratio of our collector and the reference collector. The higher the number, the better our collector performs. In particular, a value above 1 means our collector outperforms the reference 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 29. 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 reference collector, up to an improvement of 48% for _202_jess on small heaps.

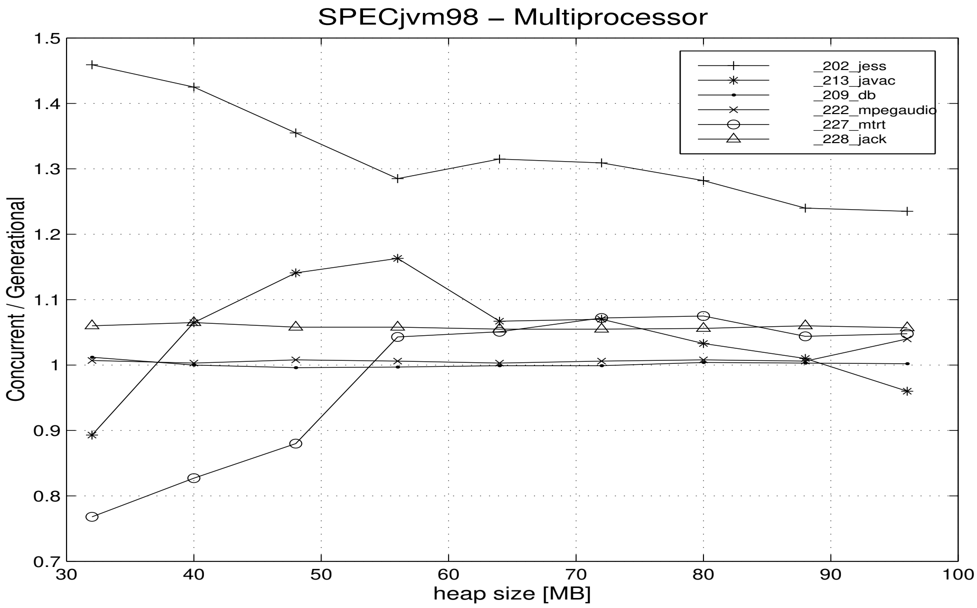

Fig. 29. Running time ratios (Jikes-Concurrent/Generational) for the SPECjvm98 suite with varying heap sizes. Results on a multiprocessor

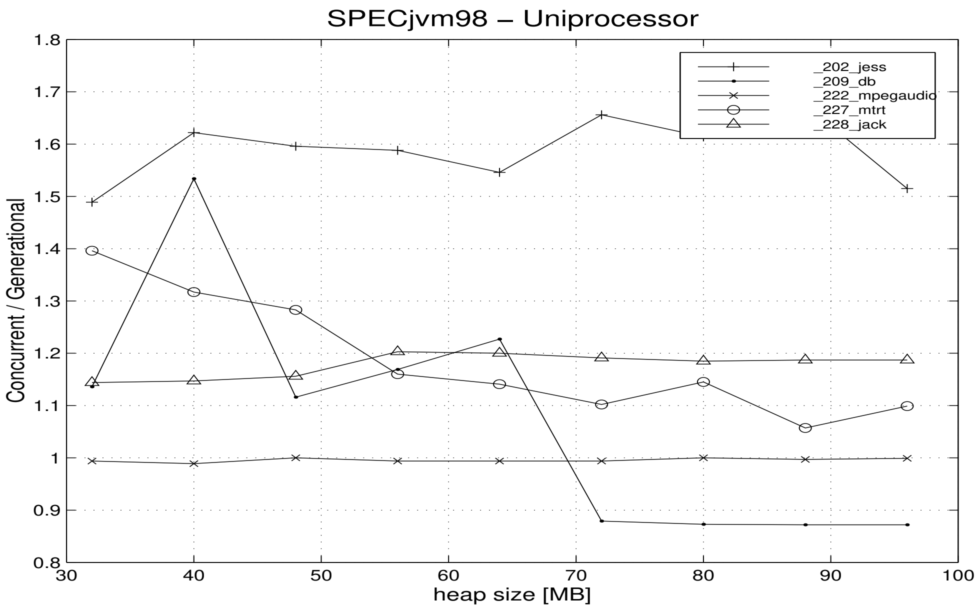

Fig. 30. Running time ratios (Jikes-Concurrent/Generational) for the SPECjvm98 suite with varying heap sizes. Results for a uniprocessor.

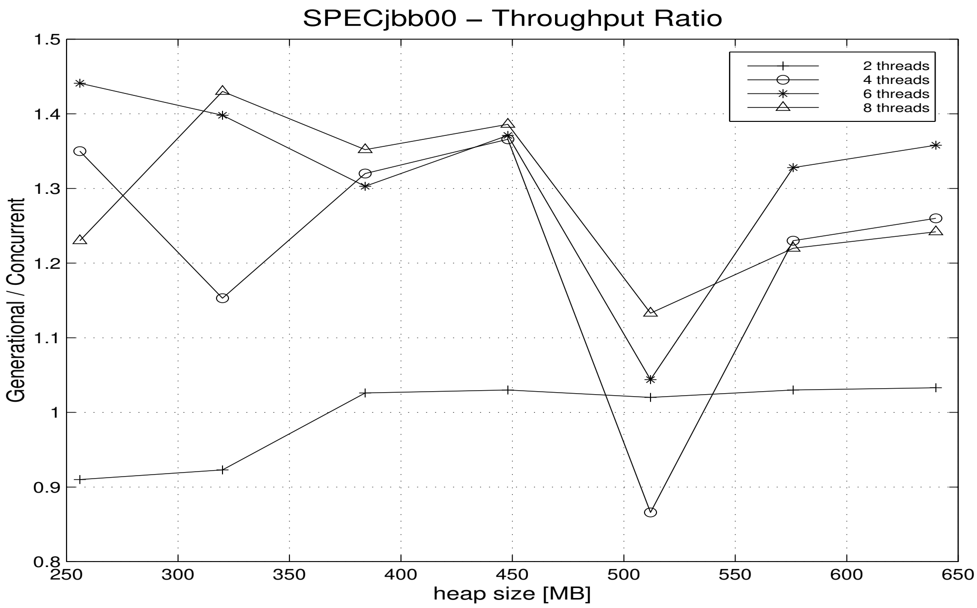

Fig. 31. SPEC_jbb2000 throughput ratios (Generational/Jikes-Concurrent) on a multiprocessor

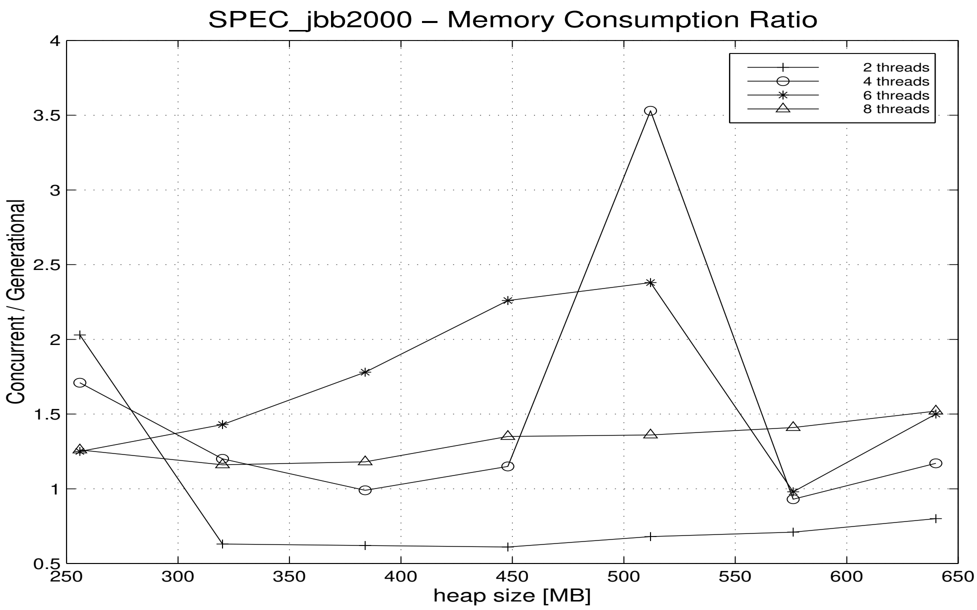

Fig. 32. SPEC_jbb2000 memory consumption (Generational vs. Jikes-Concurrent) on a multiprocessor

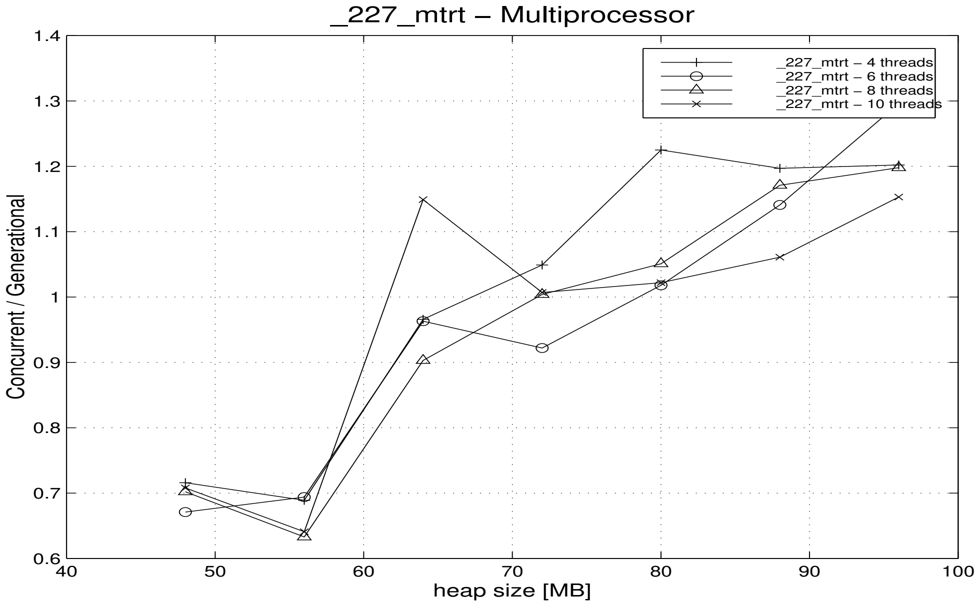

Fig. 33. Running time ratio (Jikes-Concurrent/Generational) for the _227_mtrt benchmarks on a multiprocessor.

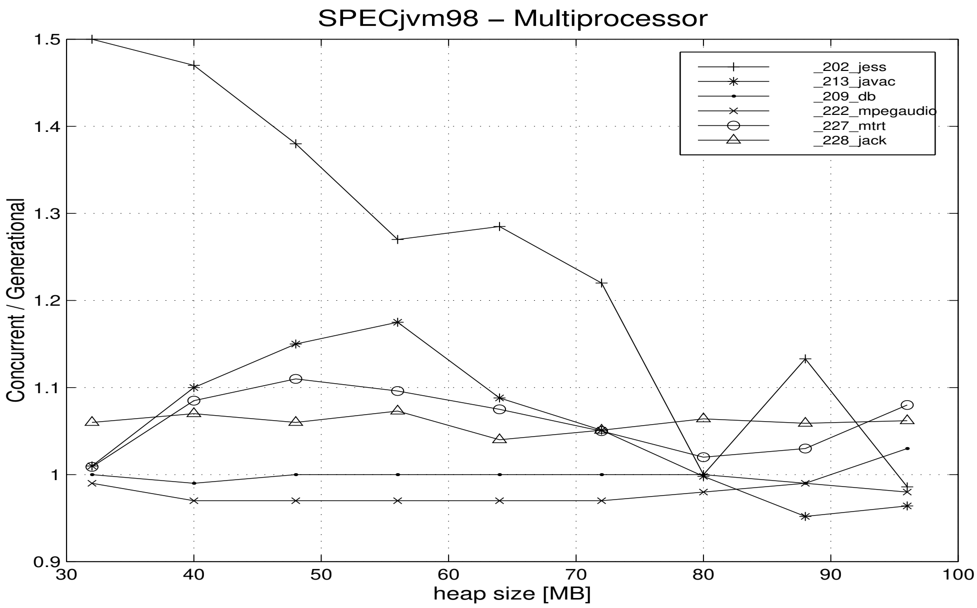

Fig. 34. The results of the second generational algorithm which uses reference counting for the minor generation. SPEC_jvm98 running time ratios (Jikes-Concurrent/Generational) on a multiprocessor

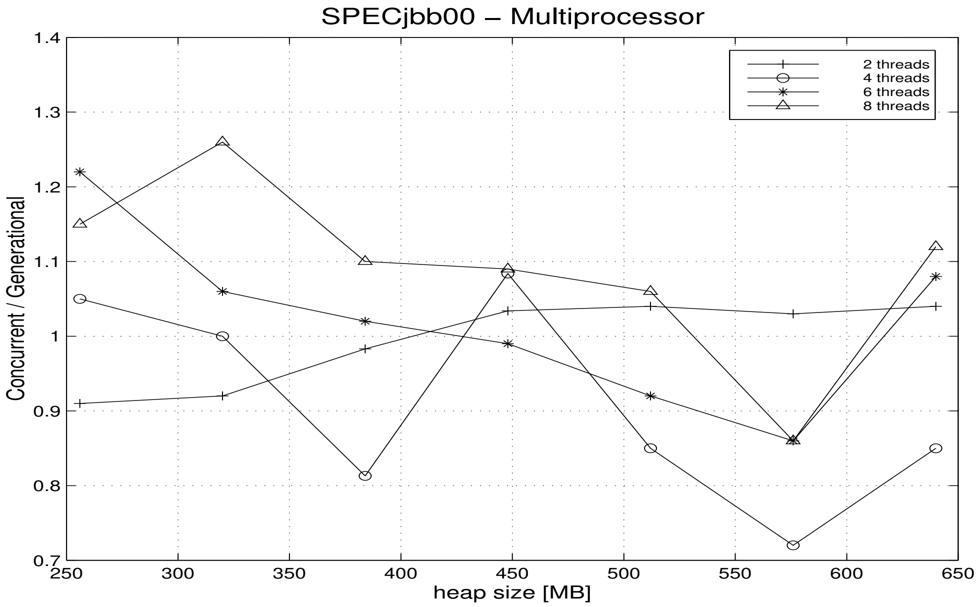

Fig. 35. The results of the second generational algorithm which uses reference counting for the minor generation. Throughput ratio (Generational/Jikes-Concurrent) for SPEC_jbb2000 on a multiprocessor.

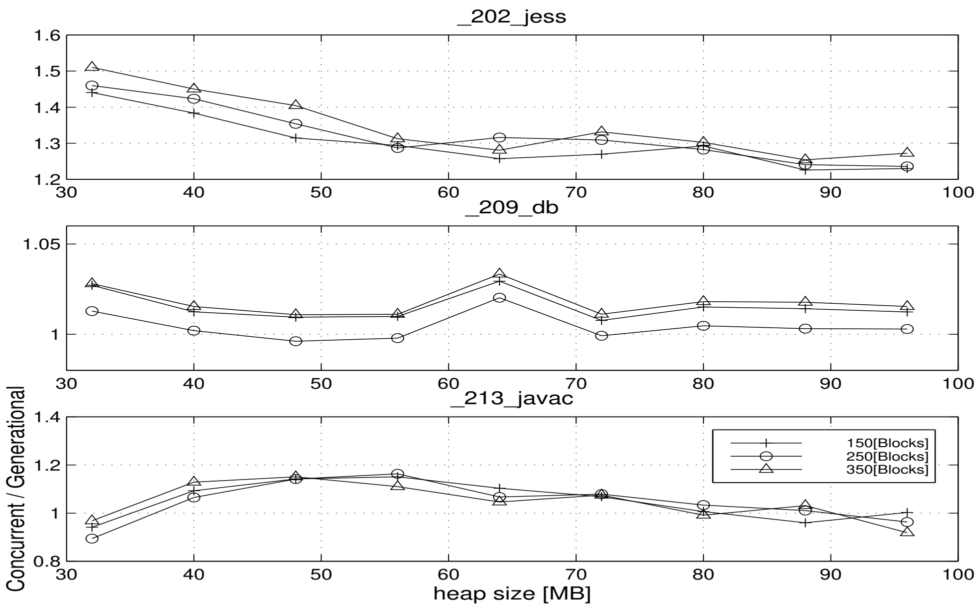

Fig. 36. RC for full algorithm - running time ratios (Jikes-Concurrent/Generational) of SPEC_jvm98 on a multiprocessor with varying minor generation size. Here, large minor generation (350 blocks) is better.

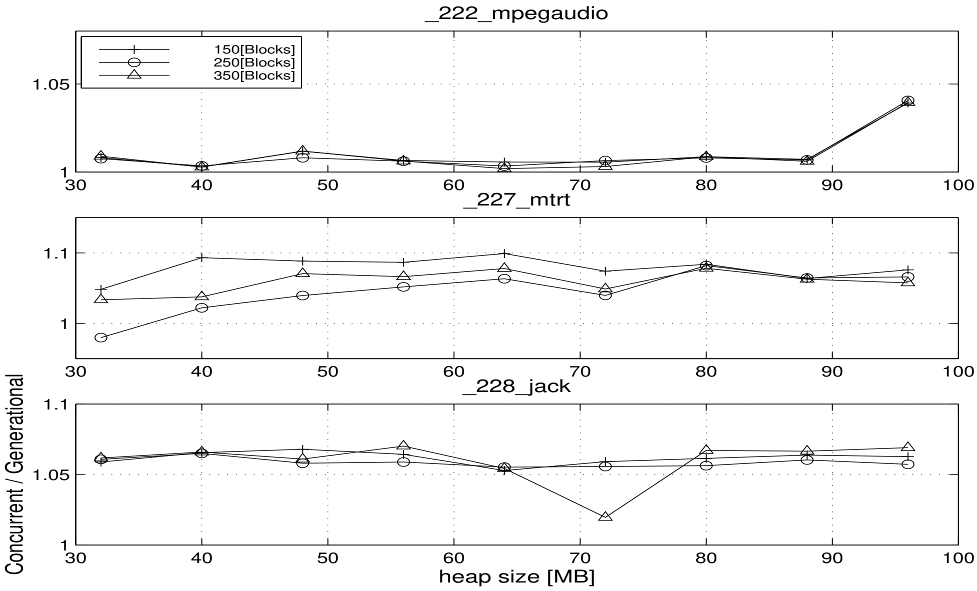

Fig. 37. RC for full algorithm - running time ratios (Jikes-Concurrent/Generational) of SPEC_jvm98 on a multiprocessor with varying minor generation size. Here, small minor generation is better.

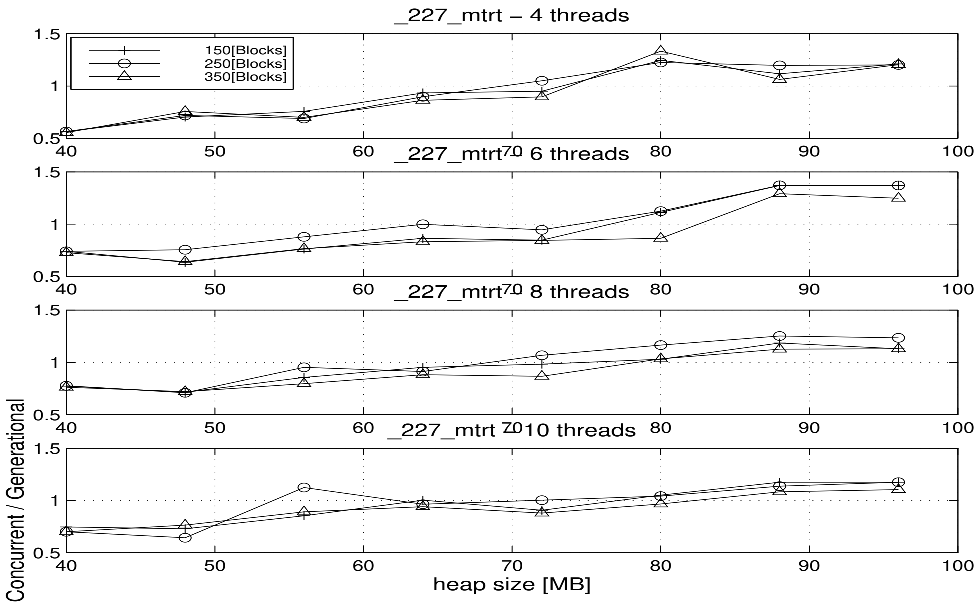

Fig. 38. RC for full algorithm - running time ratios (Jikes-Concurrent/Generational) of _227_mtrt on a multiprocessor with varying minor generation size. Here, small minor generation is better.

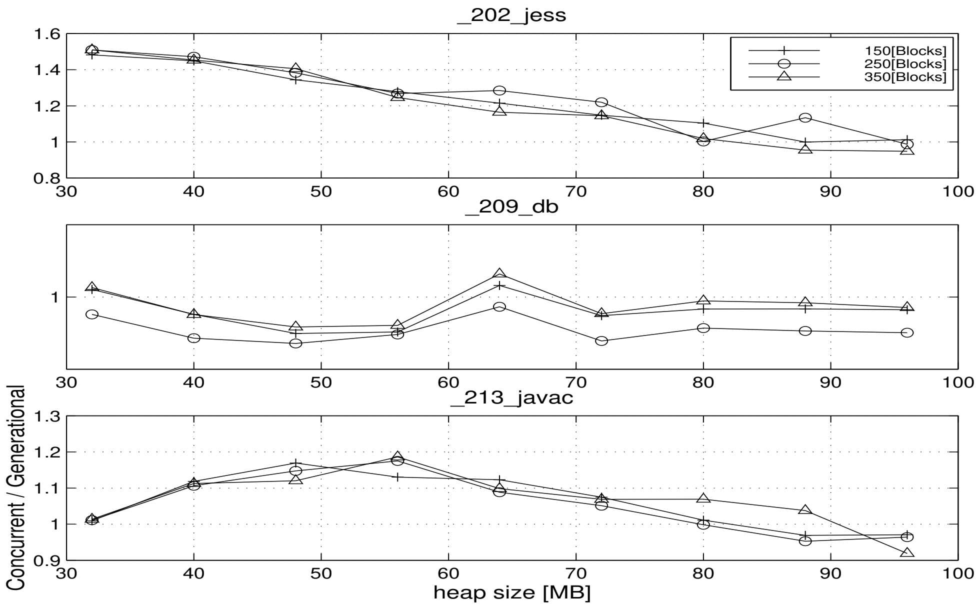

Fig. 39. RC for minor algorithm - running time ratios (Jikes-Concurrent/Generational) of SPEC_jvm98 on a multiprocessor with varying minor generation size. Here, large minor generation is better

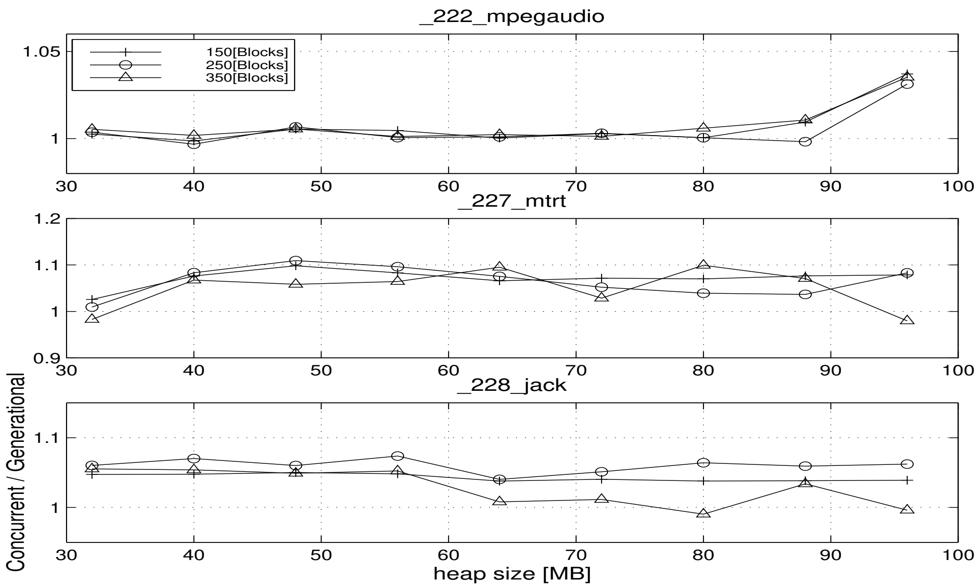

Fig. 40. RC for minor algorithm - running time ratios (Jikes-Concurrent/Generational) of SPEC_jvm98 on a multiprocessor with varying minor generation size. Here, small minor generation is better

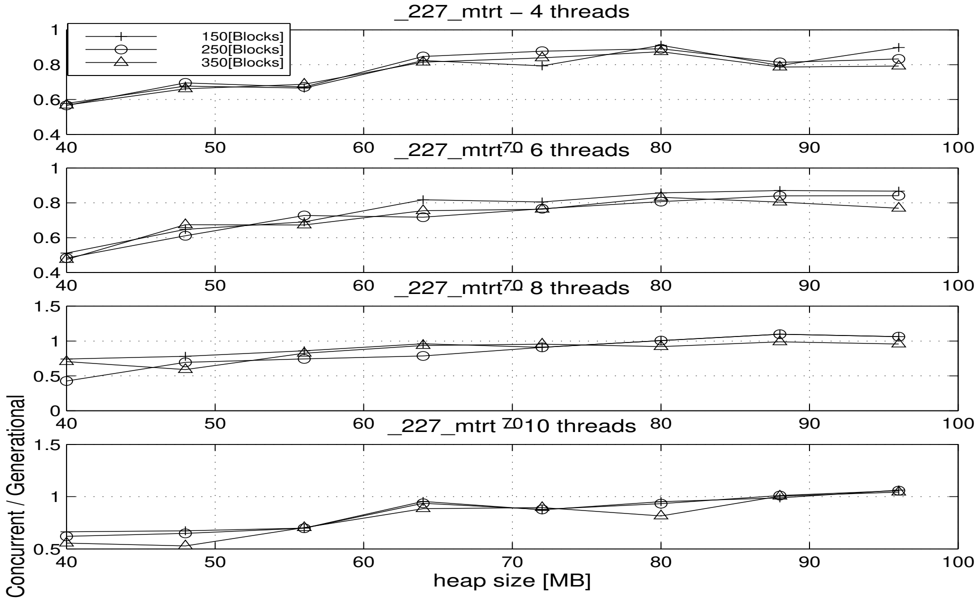

Fig. 41. RC for minor algorithm - running time ratios (Jikes-Concurrent/Generational) of _227_mtrt on a multiprocessor with varying minor generation size. Here, small minor generation is better.

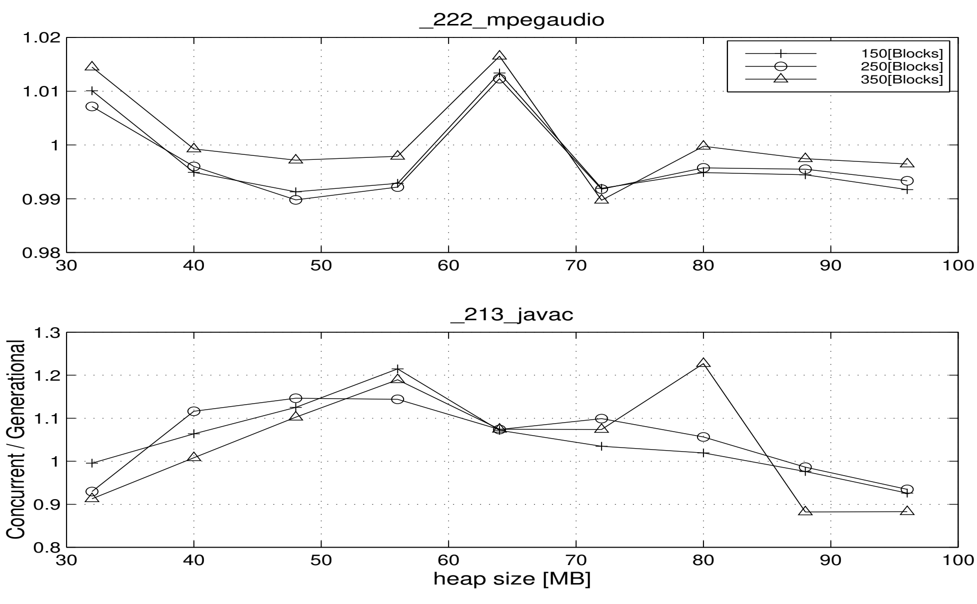

Fig. 42. RC for both algorithm - running time ratios (Jikes-Concurrent/Generational) of SPEC_jvm98 on a multiprocessor with varying minor generation size. Here, large minor generation is better

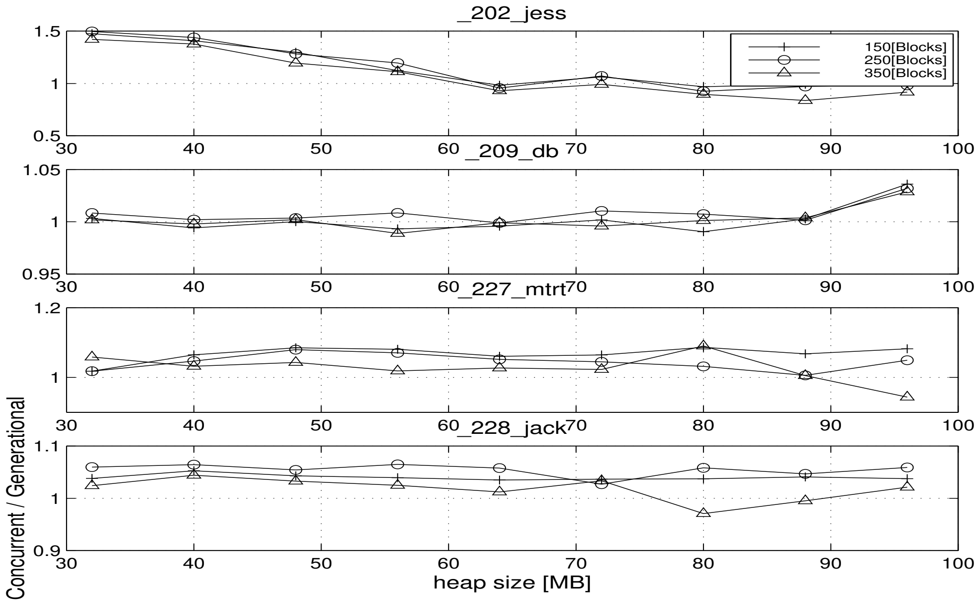

Fig. 43. RC for both running time ratios (Jikes-Concurrent/Generational) of SPEC_jvm98 on a multiprocessor with varying minor generation size. Here, small minor generation is better

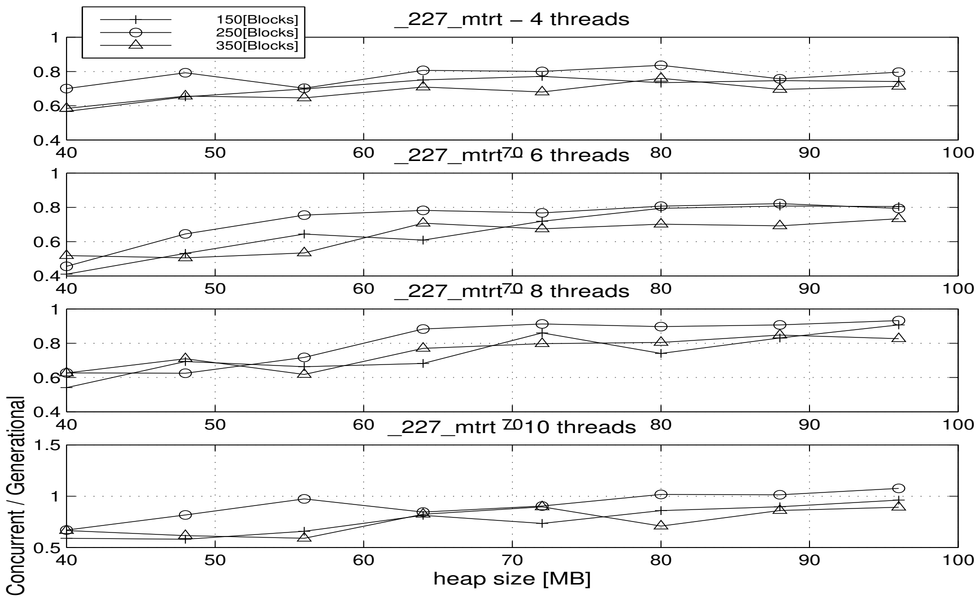

Fig. 44. RC for both algorithm - running time ratios (Jikes-Concurrent/Generational) of _227_mtrt on a multiprocessor with varying minor generation size. Here, small minor generation is better.

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 a uniprocessor. In these cases, the efficiency of the collector is important: the throughput may be harmed when the collector spends too much CPU time. We have modified the _227_mtrt benchmark to work with varying number of threads (4, 6, 8, 10 threads) and the resulting throughput measures are reported in figure 33. The measurements show an improved performance for almost all parameters with typical to large heaps, with the highest improvement being 30% for _227_mtrt with 6 threads and heap size 96MBytes. However, on small heaps the reference collector does better.

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 throughput ratio improvement. Here the result is throughput and not running-time. For clarity of representation, we report the inverse ratio, so that higher ratios still show better performance of our collector, and ratios larger than 1 imply our collector outperforming the reference collector. The measurements are reported for a varying number of threads (and varying heap sizes) in figure 31. When the system has no idle processor for the collector (4,6, and 8 warehouses), our collector clearly outperforms the reference collector. The typical improvement is 25% and the highest improvement is 45%. In the case 2 warehouses are run and the collector is free to run on an idle processor, our collector performs better when the heap is not tight, whereas on tighter heaps, the reference collector wins.

The maximum pause times for the SPECjvm98 benchmarks and the SPECjbb2000 benchmark are reported in figure 28. 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 figure 28) are as low as those of the reference collector and they are all below 5ms.

We go on with a couple of graphs presenting measurements of the second best collector: the one that runs reference counting for the young generation and mark and sweep for the full collection. In figure 34, 35 we report the running time and throughput ratio of this collector. As seen from these graphs this collector does not perform significantly worse. In most measurements, it did better than the reference collector, up to an improvement of 50% for _202_jess on small heaps and 25% for the SPECjbb2000 benchmark with 8 number of threads. More measurements appear in our technical report.

Client measurements. Finally, 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 appears in Section 5 above). We report the uniprocessor tests in figure 30. It turns out that the generational algorithm is better than the reference collector in almost all tests. Note the large improvement of around 60% for the _202_jess benchmark. Finally, SPECjbb2000 reports the heap consumption. We report this measure in Figure 32. As can be seen, the heap consumption has typically decreased. This can be explained by the frequent reuse of the young generation space.

7 Conclusions

We have presented three designs for integrating generations with an on-the-fly reference counting collector: using reference counting for the full collection and mark and sweep for collecting the young generation, using reference counting for collecting the young generation and mark and sweep for collecting the full collection, and using reference counting for the both generations collection. A tracing collector is seldom used to collect cyclic garbage structures. We used the Levanoni-Petrank sliding view collectors as the building blocks for this design. The collector was implemented on Jikes and was run on a 4-way IBM Netfinity server.

Our measurements against the Jikes concurrent collector show a large improvement in throughput and the same low pause times. The collector presented here is the best among the three possible incorporation of generations into reference counting collectors.

References

1. Hezi Azatchi and Erez Petrank. Integrating Generations with Advanced Reference Counting Garbage Collectors. Technical Report, Faculty of Computer Science, Technion, Israel Institute of Technology, October 2002. Available at http://www.cs.technion.ac.il/∼erez/publications.html.
2. Alan Demers, Mark Weiser, Barry Hayes, Hans Boehm, 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, January 1990. ACM Press, pages 261-269.
3. Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger. *The AWK Programming Language*. Addison-Wesley, 1988.
4. Andrew W. Appel, John R. Ellis, and Kai Li. Real-time concurrent collection on stock multiprocessors. *ACM SIGPLAN Notices*, 23(7):11–20, 1988.
5. D. Bacon, C. Attanasio, H. Lee, V. Rajan, and S. Smith. Java without the coffee breaks: A nonintrusive multiprocessor garbage collector. To appear in the *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Snowbird, Utah, June 20-22 2001.
6. D. Bacon and V. Rajan. Concurrent Cycle Collection in Reference Counted Systems. To appear in the *Fifteenth European Conference on Object-Oriented Programming* (ECOOP), University Eötvös Lorand, Budapest, Hungary, June 18-22 2001.
7. Henry G. Baker. List processing in real-time on a serial computer. *Communications of the ACM*, 21(4):280–94, 1978.
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. Hans-Juergen Böhm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. *ACM SIGPLAN Notices*, 26(6):157–164, 1991.

10. T. Chikayama and Y. Kimura. Multiple reference management in Flat GHC. *ICLP*, pages 276–293, 1987.
11. George E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 3(12):655–657, December 1960.
12. John DeTreville. Experience with concurrent garbage collectors for Modula-2+. Technical Report 64, DEC Systems Research Center, Palo Alto, CA, August 1990.
13. John DeTreville. Experience with garbage collection for modula-2+ in the topaz environment. In *OOPSLA/ECOOP '90 Workshop on Garbage Collection in Object-Oriented Systems*, October 1990.
14. L. Peter Deutsch and Daniel G. Bobrow. An efficient incremental automatic garbage collector. *Communications of the ACM*, 19(9):522–526, September 1976.
15. 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.
16. Damien Doligez and Georges Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *POPL* 1994.
17. Damien Doligez and Xavier Leroy. A concurrent generational garbage collector for a multi-threaded implementation of ML. In *POPL* 1993.
18. Tamar Domani, Elliot K. Kolodner, Ethan Lewis, Elliot E. Salant, Katherine Barabash, Itai Lahan, Yossi Levanoni, Erez Petrank, and Igor Yanover. Implementing an On-the-fly Garbage Collector for Java. *The 2000 International Symposium on Memory Management*, October, 2000.
19. Tamar Domani, Elliot K. Kolodner, and Erez Petrank. Generational On-the-fly Garbage Collector for Java. *ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation* (PLDI) 2000.
20. Shinichi Furusou, Satoshi Matsuoka, and Akinori Yonezawa. *Parallel conservative garbage collection with fast allocation.* In Paul R. Wilson and Barry Hayes, editors, OOPSLA, October 1991.
21. Adele Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.
22. Richard E. Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.* Wiley, July 1996.
23. Yossi Levanoni and Erez Petrank. An On-the-fly Reference Counting Garbage Collector for Java, *proccedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'01).* See also the Technical Report CS-0967, Dept. of Computer Science, Technion, Nov. 1999.
24. H. Lieberman and C. E. Hewitt. A Real Time Garbage Collector Based on the Lifetimes of Objects. *Communicaitons of the ACM*, 26(6), pages 419-429, 1983.
25. James W. O'Toole and Scott M. Nettles. Concurrent replicating garbage collection. Also LFP94 and OOPSLA93 Workshop on Memory Management and Garbage Collection.
26. Young G. Park and Benjamin Goldberg. Static analysis for optimising reference counting. *IPL*, 55(4):229–234, August 1995.
27. Manoj Plakal and Charles N. Fischer. Concurrent Garbage Collection Using Program Slices on Multithreaded Processors.
28. Tony Printezis and David Detlefs. A generational mostly-concurrent garbage collector. *ISMM* 2000.
29. David J. Roth and David S. Wise. One-bit counts between unique and sticky. *ACM SIGPLAN Notices*, pages 49–56, October 1998. ACM Press.
30. Guy L. Steele. Multiprocessing compactifying garbage collection. *Communications of the ACM*, 18(9):495-508, September 1975.
31. Standard Performance Evaluation Corporation, `http://www.spec.org/`

32. Will R. Stoye, T. J. W. Clarke, and Arthur C. Norman. Some practical methods for rapid combinator reduction. In *LFP*, pages 159–166, August 1984.
33. D. Ungar. Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm. ACM SIGPLAN Notices Vol. 19, No. 5, May 1984, pp. 157-167.
34. Larry Wall and Randal L. Schwartz. *Programming Perl.* O'Reilly and Associates, Inc., 1991.
35. J. Weizenbaum. Symmetric list processor. *Communications of the ACM*, 6(9):524–544, September 1963.
36. David S. Wise. Stop and one-bit reference counting. *IPL*, 46(5):243–249, July 1993.
37. David S. Wise. Stop and one-bit reference counting. Technical Report 360, Indiana University, Computer Science Department, March 1993.