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.

* Most of this work was done while the author was at the Computer Science Dept., Technion – Israel Institue of Technology.
** This research was supported by the E. AND J. BISHOP RESEARCH FUND and by the FUND FOR PROMOTION OF RESEARCH at the Technion.

G. Hedin (Ed.): CC 2003, LNCS 2622, pp. 185–199, 2003.
© Springer-Verlag Berlin Heidelberg 2003

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. [8].) 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 building on and improving on previous work. Levanoni and Petrank [17] following DeTreville [9] have presented an 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 Generational Collection

Generational garbage collection was introduced by Lieberman and Hewitt [18], and the first published implementation was by Ungar [24]. 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.

1.3 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 [17]: 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 percentage of objects that survive is small in the young generation and high in the old generation. If we look 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.

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 in which generations are used with a mark and sweep collector [15][1].

1.4 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

[1] A partial incorporation of generations with an mark and sweep collector, used only for immutable objects was used by Doligez, Leroy and Gonthier [13,12]. 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.

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.

1.5 Implementation and Results

We have implemented our algorithms 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[23]. In Section 5 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 in [3]. 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. A possible explanation to this phenomena is that reference counting is more efficient than the tracing collection (of the young generation) when the collections are too frequent. In this case, the tracing collector must trace the live (young) objects repeatedly, whereas the reference counting only spends time proportional to the work done in between the collections.

1.6 Cycle Collection

A major disadvantage of reference counting is that it does not collect cycles. If 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 occasionally to collect the full heap and reclaim garbage cycles[2].

1.7 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 in section 5 we discuss our implementation and present our measurements. We conclude in section 6.

[2] Another option was to use the cyclic structures collector of Bacon and Rajan [4] but from their measurements it seems that tracing collectors should be more efficient. Thus, we chose to use the readily available tracing collector.

2 An Overview on Reference Counting Algorithms

The traditional method of reference counting, was first developed for Lisp by Collins [8]. 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 (its children values at the previous sliding-view) are then decremented as well and the reclamation may continue recursively. Improvements to the naive algorithm were suggested in several subsequent papers. Weizman [25] studied ameliorating the delay introduced by recursive deletion. Several works [22,26] 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 [10] 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+ [9]. Further study on reducing work for local variables can be found in [6] and [19].

Reference counting seemed to have an intrinsic problem with multithreading implying that a semaphore must be used for each pointer update. The problems were dealt with a series of paper [9,20,3,17]. The sliding views algorithm of Levanoni and Petrank [17] presented a reference counting collector that completely eliminated the need for a synchronization operation in the write barrier. In this work, we use the sliding views algorithms as the basic clock for the generational algorithms. A detailed description of the Levanoni Petrank collectors follow.

3 The Levanoni-Petrank Collectors

In this section we provide a short overview of the Levanoni-Petrank collectors. Due to space limitations we omit the pseudo code. More details appear in our technical report [1]. The full algorithm is described in the original paper [17].

3.1 The Sliding-View Reference Counting Algorithm

The Levanoni-Petrank collectors [17] 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 records 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 slot. The analysis shows that (infrequent) 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.

All created objects are marked dirty during creation. 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. 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 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. 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. 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. 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*. After the fourth handshake the collector proceeds to adjust rc fields due to differences between the sliding views of the previous and current cycle. 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*.

3.2 The Sliding-View Tracing Algorithm

"Snapshot at the beginning" [16] mark&sweep collectors exploit the fact that a garbage object remains garbage until the collector recycles it. i.e., being garbage is a stable property. 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. After the Mark-Roots stage, the collector starts tracing according to the sliding view associated with the cycle. When in needs 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 already committed (by some mutator) replica which is pointed to 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.

4 The Generational Collectors

In this section we describe the collectors we have designed. Due to lack of space, we concentrate on the winning collector. The description of the other two collectors appears in our technical report [1].

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. Recall that 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. By the experience of Domani et al [15] we believe that spending more collection efforts on an aging mechanism does not pay. See [15] for more the details of this experience.

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.
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 young unmarked objects. The young generation contains all the objects that were created since the *previous* collection cycle and were logged by the i-th mutator to its local $Young\text{-}Objects_i$ buffer. These local buffers hold addresses of all newly created objects since the recent collection and can be also viewed as holding pointers to all objects in the young generation to be processed by the next collection. These buffers are retrieved by the collector in the first handshake of the collection and their union *Young-Objects* buffer of the collector is the young generation to be processed(swept) in this minor collection cycle.

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 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 [17] 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 a young object to the collector only in the next collection. But the update is recorded in the current collection (the update was executed before the first handshake in the view of Mutator M_i) and will not be seen in the next collection. Thus, an inter-generational pointer will be missing from the view of the next collection.

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 a young objects in the uncertainty period from before the first handshake has begun until 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-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-Buffer*. 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-Young-Objects* and *Major-Updates* buffers are full collection buffers that correspond to the minor collection's *Young-Objects* and *Updates* buffers. These buffers are prepared by the minor collections to serve the full collection. Only those objects which 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, thus, logged to *Major-Updates* buffer or *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-Young-Objects* or *Young-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 ([17] following [10]), 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.

5 Implementation and Results

We have implemented our collectors into Jikes. We have decided to use the non-copying allocator of Jikes, which is based on the allocator of Boehm Demers and Shenker [7]. This allocator is suitable 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). 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.

We have taken measurements on a 4-way IBM Netfinity 8500R server with 550MHz Intel Pentium III Xeon processors and 2GB of physical memory. We also measured the run of our collector on a client machine: a single 550MHZ Intel Pentium III 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[23].

The Jikes concurrent collector. Our collectors measurements are compared with the concurrent reference counting collector supplied with the Jikes package and reported in [3]. The Jikes concurrent collector is an advanced on-the-fly pure reference-counting collector and it has similar characteristics as our collectors, namely, the mutators are loosely synchronized with the collector, allowing very low pause times.

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 multi-threaded benchmark, we have also modified the _227_mtrt benchmark from the SPECjvm98 suite to run on a varying number of threads. We measured its run

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

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. In our technical report [1] we provide measurements for all collectors. Also, in our technical report, we provide a systematic report on how we selected our parameters, such as the triggering policy, the allocator parameters, the size of the young generation, etc. We omit these reports from this short paper for lack of space.

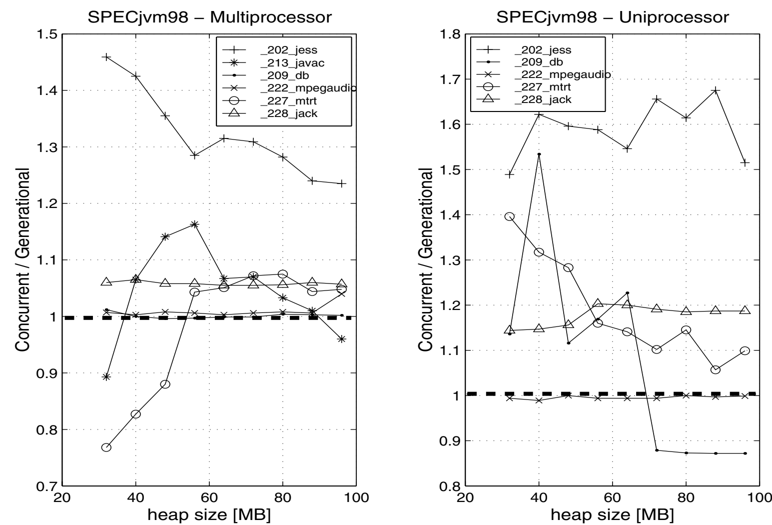

Fig. 2. Running time ratios (Jikes-Concurrent/Generational) for the SPECjvm98 suite with varying heap sizes. The graph on the left shows results on a multiprocessor and the graph on the right reports results for a uniprocessor.

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 2 the running time ratio of our collector

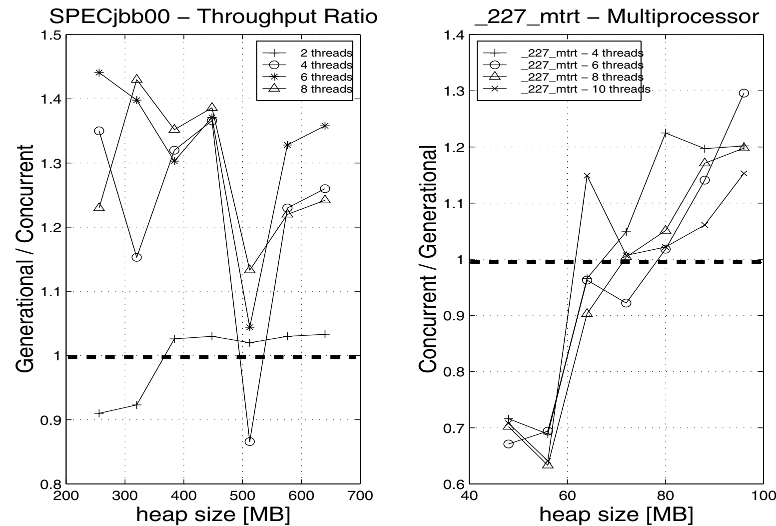

Fig. 3. The graph on the left shows SPEC_jbb2000 throughput ratios (Generational/Jikes-Concurrent) on a multiprocessor and the graph on the right reports running time ratio (Jikes-Concurrent/Generational) for the _227_mtrt benchmarks on a multiprocessor.

and the Jikes concurrent collector. The higher the number, the better our collector performs. In particular, a value above 1 means our collector outperforms the Jikes concurrent collector.

We ran each of the SPECjvm98 benchmarks on a multiprocessor, allowing a designated processor to run the collector thread. We report these results in figure 2 (graph on left). These results demonstrate performance when the system is not busy and the collector may run concurrently on an idle processor. In practically all measurements, our collector did better than the Jikes concurrent collector, up to an improvement of 48% for _202_jess on small heaps. The behavior of the collector on a busy system may be tested when the number of application threads exceeds the number of (physical) processors. A special case is when the JVM is run on 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 running time measures are reported in the right graph of figure 3. 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 Jikes concurrent collector does better.

The results of SPECjbb2000 are measured a bit differently. The run of SPECjbb2000 requires a multi-phased run with an 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

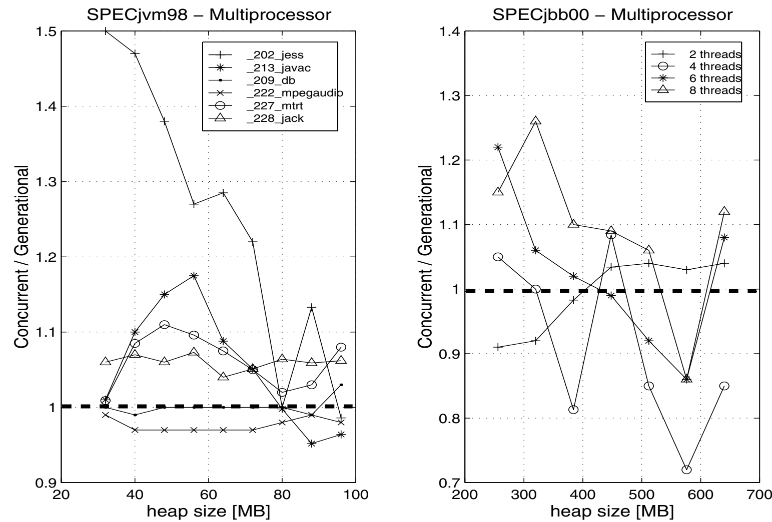

Fig. 4. The results of the second generational algorithm which uses reference counting for the minor generation. The graph on the left shows SPEC_jvm98 running time ratios (Jikes-Concurrent/Generational) on a multiprocessor and the graph on the right reports throughput ratio (Generational/Jikes-Concurrent) for SPEC_jbb2000 on a multiprocessor.

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 Jikes concurrent collector. The measurements are reported for a varying number of threads (and varying heap sizes) in the left graph of Figure 3. When the system has no idle processor for the collector (4,6, and 8 warehouses), our collector clearly outperforms the Jikes concurrent collector. The typical improvement is 25% and the highest improvement is 45%. In the case in which 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 Jikes concurrent collector wins.

The maximum pause times for the SPECjvm98 benchmarks and the SPECjbb2000 benchmark are reported in figure 1. 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 1) are as low as those of the Jikes concurrent 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 4 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 Jikes concurrent 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. We report the uniprocessor tests in figure 2 (graph on right). It turns out that the generational algorithm is better than the Jikes concurrent collector in almost all tests. Note the large improvement of around 60% for the _202_jess benchmark.

6 Conclusions

We have presented a design 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. A tracing collector is infrequently 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. D. Bacon, C. Attanasio, H. Lee, V. Rajan, and S. Smith. Java without the coffee breaks: A nonintrusive multiprocessor garbage collector. *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Snowbird, Utah, June 20–22 2001.
4. D. Bacon and V. Rajan. Concurrent Cycle Collection in Reference Counted Systems. *Fifteenth European Conference on Object-Oriented Programming* (ECOOP), University Eötvös Lorand, Budapest, Hungary, June 18–22 2001.
5. Henry G. Baker. List processing in real-time on a serial computer. *Communications of the ACM*, 21(4):280–94, 1978.

6. Henry G. Baker. Minimising reference count updating with deferred and anchored pointers for functional data structures. *ACM SIGPLAN Notices*, 29(9), September 1994.
7. Hans-Juergen Böhm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. collector. *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI, 1991)*, *ACM SIGPLAN Notices*, 26(6):157–164, 1991.
8. George E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 3(12):655–657, December 1960.
9. John DeTreville. Experience with concurrent garbage collectors for Modula-2+. Technical Report 64, DEC Systems Research Center, Palo Alto, CA, August 1990.
10. L. Peter Deutsch and Daniel G. Bobrow. An efficient incremental automatic garbage collector. *Communications of the ACM*, 19(9):522–526, September 1976.
11. 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.
12. Damien Doligez and Georges Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *POPL* 1994.
13. Damien Doligez and Xavier Leroy. A concurrent generational garbage collector for a multi-threaded implementation of ML. In *POPL* 1993.
14. 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. *ISMM*, 2000.
15. 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.
16. Shinichi Furusou, Satoshi Matsuoka, and Akinori Yonezawa. *Parallel conservative garbage collection with fast allocation.* In Paul R. Wilson and Barry Hayes, editors, GC workshop at OOPSLA, October 1991.
17. 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.
18. 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.
19. Young G. Park and Benjamin Goldberg. Static analysis for optimising reference counting. *IPL*, 55(4):229–234, August 1995.
20. Manoj Plakal and Charles N. Fischer. Concurrent Garbage Collection Using Program Slices on Multithreaded Processors.
21. Tony Printezis and David Detlefs. A generational mostly-concurrent garbage collector. *ISMM* 2000.
22. David J. Roth and David S. Wise. One-bit counts between unique and sticky. *ACM SIGPLAN Notices*, pages 49–56, October 1998. ACM Press.
23. Standard Performance Evaluation Corporation, `http://www.spec.org/`
24. D. Ungar. Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm. ACM SIGPLAN Notices Vol. 19, No. 5, May 1984, pp. 157–167.
25. J. Weizenbaum. Symmetric list processor. *Communications of the ACM*, 6(9):524–544, September 1963.
26. David S. Wise. Stop and one-bit reference counting. Technical Report 360, Indiana University, Computer Science Department, March 1993.