Age-Oriented Concurrent Garbage Collection

Harel Paz[⋆1], Erez Petrank[⋆⋆1], and Stephen M. Blackburn[⋆⋆⋆2]

[1] Dept. of Computer Science, Technion, Haifa 32000, Israel.
[2] Dept. of Computer Science, ANU, Canberra ACT 0200, Australia.

Abstract. Generational collectors are well known as a tool for shortening pause times incurred by garbage collection and for improving garbage collection efficiency. In this paper, we investigate how to best use generations with on-the-fly collectors. On-the-fly collectors run concurrently with the program threads and induce very short program pauses. Thus, the motivation for incorporating generations is focused at improving the throughput; pauses do not matter, since they are already very short. We propose a new collection approach, denoted *age-oriented* collection, for exploiting the generational hypothesis to obtain better efficiency. This approach is particularly useful when reference counting is used to collect the old generation, yielding a highly efficient and non-obtrusive on-the-fly collector. Finally, an implementation is provided demonstrating how the age-oriented collector outperforms both the non-generational and the generational collectors' efficiency.

1 Introduction

Dynamic memory management and garbage collection is arguably a key factor in supporting fast and reliable large software products. However, naive garbage collection algorithms may have undesirable effects on program behavior, most notably long pauses and reduced throughput[3]. Generational garbage collection [20, 27] ameliorates both problems by reducing the average pause times and increasing efficiency. The basic assumption underlying generational collectors design is the weak generational hypothesis: "most objects have short lifetimes". Given this hypothesis, it makes sense to concentrate the effort on young objects which are most likely to be unreachable. Generational collectors segregate objects according to their age into two or more groups called generations, and run frequent collections of the young generation. Keeping the young generation small yields frequent short collections that make room for further allocations. The older generation (or the entire heap) is collected infrequently when space is exhausted. Full heap collections require long pauses, but are infrequent.

⋆ Email: pharel@cs.technion.ac.il.
⋆⋆ Email: erez@cs.technion.ac.il. Research supported by the Bar-Nir Bergreen Software Technology Center of Excellence and by the IBM Faculty Partnership Award.
⋆⋆⋆ Email: Steve.Blackburn@anu.edu.au
[3] *Throughput* is the amount of work completed in a fixed time period.

If the generational hypothesis is indeed correct, we get several advantages. First, reducing pauses is achieved for most collections. Second, collections are more efficient since they concentrate on the young part of the heap where a high percentage of garbage is found. 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).

On-the-Fly Garbage Collection for Multiprocessors.
Many garbage collectors work while program threads are stopped. On multiprocessor platforms, it is not desirable to stop the program and perform the collection in a single thread on one processor, as this leads both to long pause times and poor processor utilization. A concurrent collector runs concurrently with the program threads. The program threads may be stopped for a short time to initiate and/or finish the collection. An *on-the-fly* collector is a concurrent collector that does not need to stop the program threads simultaneously, not even for the initialization or the completion of the collection cycle. Such collectors are targeted at multiprocessors, usually employed as server machines.

1.1 This work

In this work, we propose a new way, denoted *age-oriented* collection, to better exploit the generational hypothesis with concurrent and on-the-fly garbage collectors. Concurrent collectors already achieve short pause times and therefore the main interest in using the generational hypothesis is to try and improve the application throughput. An age-oriented collector is defined as follows.

Definition 1: *An* age-oriented *collector is a collector that*

- *always collects the entire heap (unlike generational collectors),*
- *during a collection it treats each generation differently (like generational collectors).*

Age-oriented collectors differ from generational collectors because the entire heap is always collected (infrequently). Like the generational framework, an age-oriented collector may be instantiated in various ways, depending on the choice of collector for the young generation and the choice of collector for the old generation. Reasonable instantiations should handle the young generation with a collector that is efficient with a high death rate, and handle the old generation with a collector that is efficient with lower death rates. In particular, our flagship instantiation of the generic age-oriented collector employs reference counting for the old generation and mark and sweep for the young generation. The complexity of reference counting is proportional to the number of pointer updates and the amount of unreachable space. Therefore, it can handle huge live spaces efficiently. Mark and sweep benefits from a high death rate since its complexity bottleneck is the scanning of the live objects[4].

[4] Avoiding the sweep by using copying collectors may be even better for the young generation, but concurrent versions of copying collectors are not easy to obtain.

One other instantiation that we have tried (and is now delivered) with the Jikes RVM is a parallel age-oriented collector denoted *copyMS*, employing mark and sweep for the old generation, and copying for the young generation. In this paper, we focus on the use of *concurrent* age-oriented collectors, which was most successful in practice.

We build on three previous on-the-fly collectors.

1. The on-the-fly reference counting collector of Levanoni and Petrank [19].
2. The on-the-fly mark and sweep collector of Azatchi et al. [3].
3. The *generational* on-the-fly collector of Azatchi and Petrank [4] that uses collector (2) for the young generation and collector (1) for full heap collections.

The third (generational) collector that builds on the first two collectors, outperformed the original collectors. In this paper, we also use the first and second collector, but we combine them in an age-oriented manner. We show that the obtained age-oriented collector outperforms even the more efficient third (generational) collector of [4].

Organization. In Section 2, we introduce the age-oriented framework and our proposed instantiation. In Section 3, an overview of the original reference counting collector [19] is presented. An overview of the age-oriented collector algorithm is introduced in Section 4. Performance results are described in Section 5. Related work is discussed in Section 6. We conclude in Section 7.

2 Age-Oriented Collection: Motivation and Overview

Although generational collectors reduce pauses and improve efficiency, they also impose some overhead. One major overhead is the manipulation of inter-generational pointers. These are pointers that point from the old generation to the young generation. If the young generation is collected while the old generation is not, these pointers must be accounted for: they may be the only evidence that a young object is reachable. Keeping record of all inter-generational pointers and using them as roots for the young generation collection poses an overhead. Many papers investigate reducing this overhead via efficient recording methods (e.g., card marking). A second overhead of generational collection is the frequent initiation of young generation collections, which repeatedly involves synchronization with the program threads, marking of all roots, etc. Using a large young generation implies less frequent collections and better throughput, but also longer pauses (for young generation collections).

Previous on-the-fly generational collectors [14, 4] have used a fixed sized young generation. Using a small fixed sized young generation is useful for the stop-the-world framework as they shorten most pause times. However, the size of the young generation does not determine the pause times with on-the-fly collectors[5]. Hence, we can use a larger young generation in order to achieve better

[5] We normally measure pauses induced by on-the-fly collectors when the number of program threads is smaller than the number of CPU's. If the number of threads

throughput. It has been noted in [2, 7] that the larger the young generation is, the more efficient the generational collector gets.

Age-oriented collectors, i.e., collectors which follow definition 1, have the following advantages over generational collectors. Such collectors use the largest possible young generation as they collect the old generation each time to make more young generation space. Age-oriented collectors may usually avoid recording inter-generational pointers because the entire heap is collected and inter-generational pointers may be determined during the collection. Finally, age-oriented collectors perform fewer collections than a generational collector. All these properties potentially make an age-oriented collector more efficient than a generational collector, when instantiated appropriately. Let us now motivate reference counting for use with age-oriented collection by making a couple of observations.

First, there is a difference between using tracing and using reference counting to collect the old generation. A tracing collection work is proportional to the number of reachable objects, hence there is a (relatively) fixed cost for each full collection. Delaying a tracing collection of an old generation as far as possible is desirable as it decreases the accumulated garbage collection work. On the other hand, the work of reference-counting is proportional to the mutators' work and to the number of dead objects. This work is accumulative. Thus, delaying a reference-counting collection does not decrease the overall garbage collection work (it only delays and accumulates it).

A second point to note is that on-the-fly collections are triggered way before the heap gets exhausted in order to let the collection terminate concurrently *before* the free space in the heap is exhausted. If the heap does get exhausted, concurrency is lost as the program threads must wait for the collector to finish before they can next allocate. Mutators' halting yields poor processor utilization: only one processor actually works (while the rest are idle).

Putting the above two observations together we get a good match for using reference counting with the old generation in an age-oriented collector. First, when running a collection on-the-fly we may need to trigger it more frequently to let it terminate on time. Furthermore, whereas a generational collector collects the young generation repeatedly in order to defer as much as possible the collection of the old generation, an age-oriented collector does not make such a deferring attempt. When reference counting is used, running a bit more frequent old collections because of the concurrent setting or due to the age-oriented framework, does not hurt the throughput.

To summarize this motivational discussion with an overview, we instantiate the age-oriented generic collector by choosing reference counting for the old generation and mark and sweep for the young generation. We build on a previous generational collector of [4]. The underlying techniques come from [19], which is reviewed in the next section.

---

exceeds the number of processors, than large pause are induced by threads losing the CPU to one another. The lengths of such pauses depend on the operating system scheduler and is not attributed to the garbage collector.

3 Reviewing the original reference-counting collector

Section 4 describes our age-oriented collector. For completeness, we start with a review of the sliding-views reference-counting collector [19]. The age-oriented collector is constructed by adding some simple modifications to this collector.

The sliding-views collector [19] is an on-the-fly collector. It is a reference-counting collector that eliminates many of the reference count updates by the following coalescing strategy. Consider a pointer slot p that is assigned the values $o_0, o_1, o_2, \ldots, o_n$ between two garbage collections. All previous reference counting collectors execute $2n$ reference count updates for these assignments: RC(o_0)$--$, RC(o_1)$++$, RC(o_1)$--$, RC(o_2)$++$, ..., RC(o_n)$++$. However, only two updates are required: RC(o_0)$--$ and RC(o_n)$++$.

Suppose the reference counts we have represent the heap view at the previous collection time and we would like to update them for the current collection time. In light of the observation above, it suffices to do the following. For each pointer p that was modified between the two collections:

1. find p's referent in the previous collection time (corresponding to o_0 above) and decrement its reference count, and
2. find p's referent in the current collection time (corresponding to o_n above) and increment its reference count.

It remains to devise a mechanism that records all pointers that were modified after the previous collection. Furthermore, this mechanism should provide, for each such pointer, its referent at the previous collection time and its referent at the current collection time. To achieve this, a program thread maintains a local buffer, denoted *Updates* buffer, in which all updated pointers are logged. For efficiency, all pointers of an updated object are logged rather than each single updated pointer. To make sure that each object is logged only once, a *dirty* bit per object is employed to signify whether the object is logged. During a collection, all objects are marked not dirty. Then, at the first time a thread modifies an object, it marks the object dirty and it logs all its pointers' previous referents in the *Updates* buffer. Further modification to the (dirty) object will not be recorded. When a new collection begins, the *Updates* buffer provides all the information required to update the reference counts: it lists all modified pointers, and keeps a record of their values before the first modification (these are the referents of these pointers in the previous collection time). In the current collection, the collector finds the current referent of the pointer on the heap.

A special case of modified objects are newly created objects. Such objects do not have referents at the previous collection time since they did not exist then. Newly created objects are created dirty (to prevent logging in the *Updates* buffer) and are logged (upon creation) in a special buffer, denoted the *YoungObjects* buffer. The collector increments the reference counts of their referents at the current collection time, but does not need to do any related decrements.

An example appears in Figure 1. It depicts the heap and the buffers in two subsequent collections, where the view of the former collection appears on the left side. The *YoungObjects* buffer contains the six objects that were created

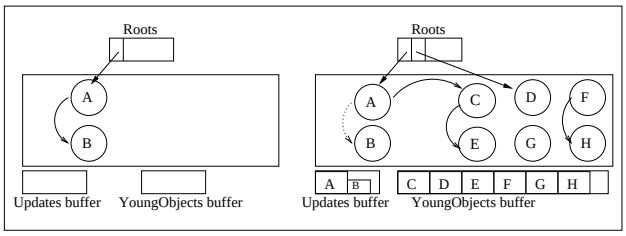

Fig. 1. An example: heap and buffers view in 2 subsequent collection.

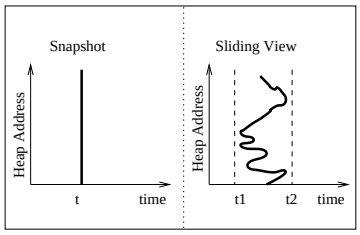

Fig. 2. A snapshot view at time t vs. a sliding view at interval $[t1,t2]$.

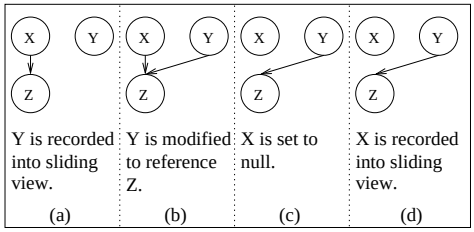

Fig. 3. An example in which the reachability of Object Z is missed by a sliding view.

1.	$Roots := programRoots \cup SnoopedObjects$
2.	for each object logged in $Updates$ do
3.	- decrement rc of its previous sliding-view descendants
4.	- increment rc of its current sliding-view descendants
5.	for each object logged in $YoungObjects$ do
6.	increment rc of its current sliding-view descendants
7.	reclaim objects with zero rc which do not belong to *Roots* recursively

Fig. 4. Reference-Counting: Collection Cycle

after the last collection. Between the two collections a pointer in A was modified to reference C. Hence, A was logged in the *Updates* buffer, together with its previous referent B (which appears next to A in a smaller font). The collector uses this information in the following way. It iterates over the objects logged in the *Updates* buffer and finds A. It decrements the reference count of B, which is A's descendant in the previous collection, and it increments the reference count of C, A's descendant at the current collection time. It then iterates over the six objects in the *YoungObjects* buffer. It increments the reference counts of their descendants at the current collection time. For example, for the object F the reference count of H is incremented.

Virtually, the above algorithm uses a snapshot of the heap. A snapshot at time t is a copy of the content of each object in the heap at time t. To get an on-the-fly collector, the program threads are not stopped simultaneously and thus a snapshot cannot be used. Instead, a collection works with a *sliding view* of the heap. A sliding view of the heap is associated with a time interval $[t1,t2]$ (rather

```
1.  Roots := programRoots ∪ SnoopedObjects
2.  youngGenerationRoots := YoungObjects ∩ Roots
3.  for each object logged in Updates do
4.      - decrement rc of its previous sliding-view descendants
5.      - increment rc of its current sliding-view descendants, while adding
          young objects whose rc is incremented into youngGenerationRoots
6.  trace young objects reachable from youngGenerationRoots, while
7.      incrementing the rc of each object traced
8.  reclaim young objects with zero rc which do not belong to Roots
9.  reclaim old objects with zero rc which do not belong to Roots recursively
```

Fig. 5. Age-Oriented: Collection Cycle

than a single point in time). It provides the content of each object in the heap at an arbitrary time t, satisfying $t_1 \leq t \leq t_2$. In contrast to a snapshot, objects are not all viewed at the same time. Figure 2 depicts the difference between a sliding view and a snapshot. Using a sliding view for collection introduces a correctness danger: objects reachability may not be reflected correctly in the view. Figure 3 shows such example, where the reachability of Z is missed in the sliding view, although it is reachable. A solution to this problem is a *snooping* mechanism. The *snooping* mechanism (via the write-barrier) records any object to which a new reference is created in the heap during the time interval [$t1$,$t2$]. Snooped objects are considered roots, and are not reclaimed in the current collection cycle.

The main phases of the sliding views algorithm (a simplified version) are presented in Figure 4. Further details are irrelevant for this paper and can be found in the original paper [19].

4 The Age-oriented collector

This section presents our age-oriented collector. Full details (including pseudo-code) are omitted for lack of space, and appear in our technical report [22]. Our age-oriented collector extends the reference counting collector of [19] by using it for the old generation and adding a tracing collection for the young generation. The tracing collection is in the spirit of the tracing collector in [3].

The original reference counting collector of [19] iterates over all the young objects recorded in the *YoungObjects* buffer, incrementing the reference counts of their descendants, only to find out later that most of them are dead (assuming the weak generational hypothesis). Thus, it then decrements the reference counts of all their descendants (before deleting them). The source of this inefficiency is that the collector does not know in advance which of the young objects are dead, and which are reachable. The age-oriented collector avoids this problem by wisely detecting the roots of the young generation and tracing only the small number of reachable young objects, updating the reference counts of reachable young objects and their descendants during the trace.

The main phases of the age-oriented collector (ignoring irrelevant on-the-fly issues) are presented in Figure 5. As with generational collectors, one needs to identify all young objects directly referenced by the program roots and by old objects. We denote these objects *youngGenerationRoots*. The age-oriented collector obtains these roots for free from the data structure of the original collector. An old object that references a young object must have been modified after the previous collection, as the young object did not exist earlier. All modified objects are logged in the *Updates* buffer. After locating the roots, the tracing of the young generation uses the current sliding views as explained in Section 3. Dead young objects are freed via sweep on the $YoungObjects$ buffer and dead old objects are freed as usual by recursive freeing of the reference counting algorithm.

Example. We use Figure 1 to present the principles of the age-oriented collector. The previous sliding view is depicted on the left side, and the current sliding view is depicted on the right. The roots are depicted above the heap and the old generation (containing A and B) is visibly separated on the left side of the heap from the young generation, which is depicted on the right side of the heap. When the age-oriented collector scans the objects logged in the *Updates* buffer (line 3 in the pseudo-code of Figure 5), it finds A. It decrements the reference count of B, its descendant in the previous sliding view (line 4 in the pseudo-code), and increments the reference count of C, its current sliding-view values (line 5). The incremented values that belong to the young generation (C) are considered roots for the young generation tracing (line 5). An additional young generation root is D which is directly referenced by the program roots (line 2). Hence, the age-oriented collector traces the young generation from C and D (line 6). In comparison, the original reference counting collector would have iterated over the six young objects incrementing the reference counts of their current sliding view, only to find out later that the work spent on F, G and H was redundant.

As with any reference counting collector, this age-oriented algorithm cannot reclaim cyclic data structures in the old generation (cyclic structures in the young generation are collected immediately). To reclaim such structures, the tracing sliding view algorithm of [3] is run infrequently on the full heap.

Since the on-the-fly collector we build on [19] does not move objects, the partitioning to young and old generations is logical (as in [10, 14, 4]). A bit per object indicates whether the object is young or old. If a young object survives a collection, it is considered old in the next collection.

The new collector retains the characteristics of the original collector. In particular, it is adequate for a multithreaded environment and a multiprocessor platform, it retains the short pauses of the original collectors, and it has the potential to be efficient (which indeed is shown in the measurements below).

5 Platforms, Benchmarks, and Measurements

An Implementation for Java. The age-oriented collector was implemented in the Jikes RVM [1] (using the baseline compiler of version 2.0.3), a research Java virtual machine. The collector is suitable for any other JVM as well.

Platform and benchmarks. We run measurements on a 4-way IBM Netfinity 8500R server with a 550MHz Intel Pentium III Xeon processor and 2GB of physical memory. The benchmarks used were the SPECjvm98 benchmark suite and the SPECjbb2000 benchmark (described in [24]). The multithreaded SPECjbb2000 benchmark is more important, as the SPECjvm98 are mostly single-threaded and our algorithm, being on-the-fly, is targeted at multithreaded programs running on multi-processors. SPECjbb2000 runs in a single JVM in which threads represent terminals in a warehouse. It is run with one terminal per warehouse, thus, the number of warehouses signifies the number of threads.

Testing procedure. We used the benchmark suite using the test harness, performing standard automated runs of all the benchmarks in the suite. Each benchmark was run five times for each of the JVM's involved (each implementing a different collector). The average of this 5 runs is reported. Finally, each JVM was run on varying heap sizes. For the SPECjvm98 suite, we started with a 24MB heap size and extended the sizes by 8MB increments until a final large size of 96MB. For SPECjbb2000 we started from 256MB heap size and extended by 64MB increments until a final large size of 704MB. It should be noted that Jikes requires larger sizes than other JVMs because the same heap is used both for the application and for the data structures of the JVM itself.

The compared collectors. The age-oriented collector was tested against 3 collectors. First, against the original reference counting collector [19], denoted *the original collector*. Second, against the generational collector of [4], denoted *the generational collector*. And finally, against the Jikes parallel stop-the-world mark and sweep collector. Recall that the second (generational) collector of [4] is a collector that builds on exactly the same two collectors of [19, 3], but it combines them in the standard generational manner, whereas we combined them according to the age-oriented framework.

5.1 Comparison with Related On-the-Fly Collectors

SPECjbb2000. In Figure 6 we report the throughput results for the generational collector and the age-oriented collector against the original collector with the SPECjbb2000 benchmark. With 1-3 warehouses, the collectors do not differ much, as they run concurrently on a spare processor (on our 4-way machine), and usually manage to handle all their work while mutators are running. With 4-8 warehouses, the collector shares a processor with the program threads (yet, given a higher priority). Thus, the efficiency of the collector influences the throughput of the whole system. The results show that the age-oriented collector substantially outperforms the generational collector, which already performs better than the original collector. The superiority of the age-oriented collector is usually higher with (relatively) small heaps where more garbage collections are required. The generational collector is less efficient on tight heaps, since full collections cannot be postponed much. The improvements of the age-oriented are less visible with larger heaps simply because there are fewer collections, and less time spent on collections.

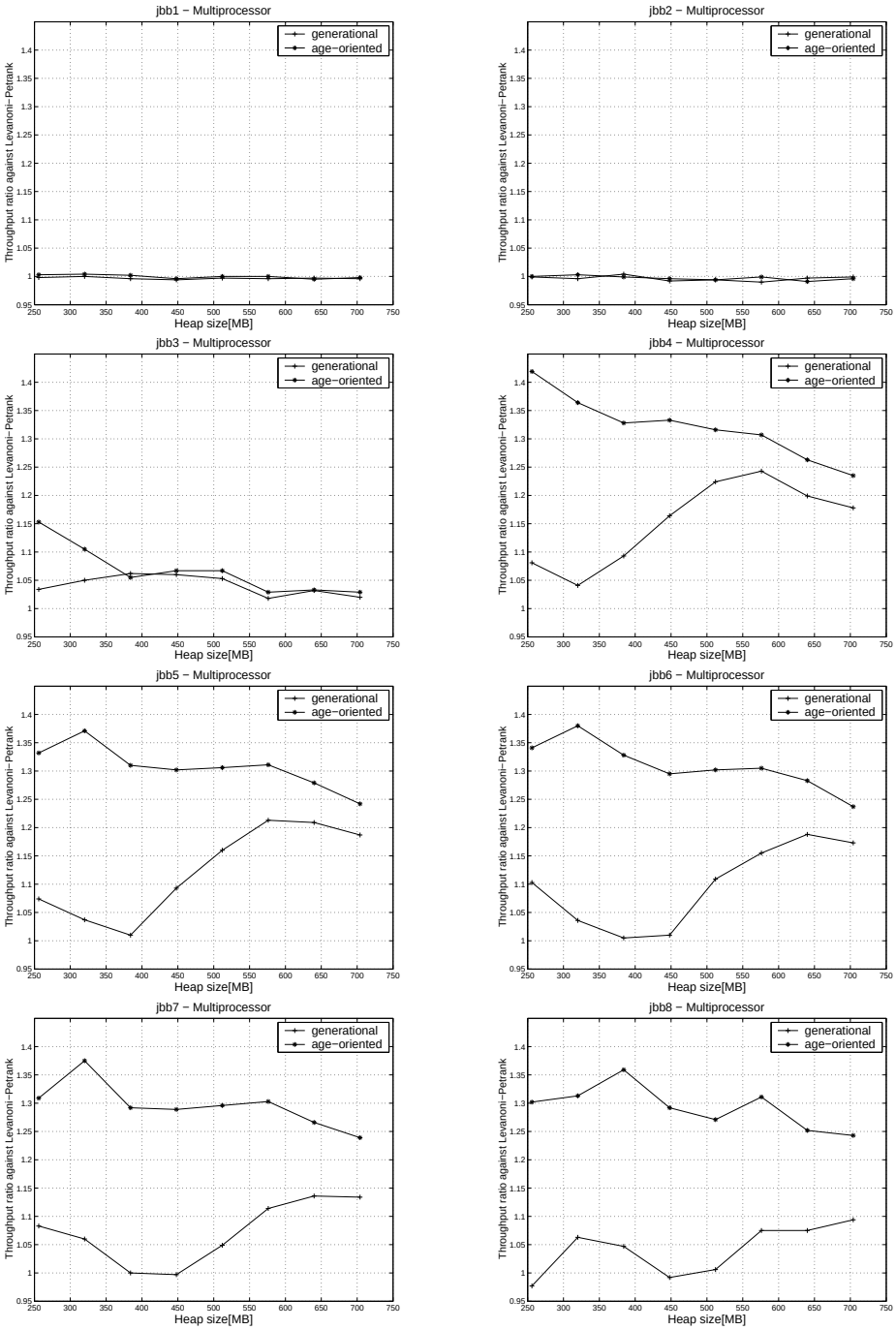

Fig. 6. SPECjbb2000 on a multiprocessor: throughput ratio of the generational and the age-oriented collector for 1-8 warehouses. The higher the ratio, the better the measured collector performs compared to the original reference counting collector. jbb$i$ stands for running with i terminals, i.e., i program threads.

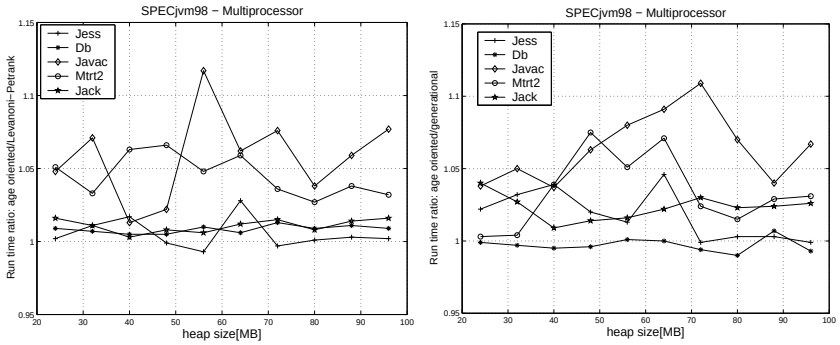

Fig. 7. SPECjvm98 on a multiprocessor: run-time ratio of the age-oriented collector compared to the original collector (left) and compared to the generational collector (right). The higher the ratio, the better the age-oriented collector performs compared to the other collector.

SPECjvm98. Figure 7 presents comparison of the age-oriented collector with the original collector and with the generational collector over all the suite's benchmarks[6]. When running the SPECjvm98 benchmarks on a multiprocessor, the collector thread can run on a designated processor and hardly influence the throughput[7]. The results show that the age-oriented collector performs slightly better than both the original collector and the generational collector. Large variations in performance are especially noticeable with _213_javac. The reason for these fluctuations is that _213_javac creates many garbage cycles in the old generation. All three collectors (the age-oriented, the generational and the original) rely on a backup tracing collector to collect these cycles. Collection of these cycles is triggered at irregular times resulting in the observed fluctuations.

5.2 Comparison to a Stop-the-World Collector

Using an on-the-fly collector leads to extremely short pause times, but has a throughput cost. To measure this cost, we have compared the performance of the age-oriented collector against the Jikes parallel stop-the-world mark and sweep collector. In this comparison, the multithreaded SPECjbb2000 was run on a 4-way platform, and SPECjvm98 benchmarks were run on a uniprocessor. The results, appearing in figure 8, show that unless the heap is tight (and then the mutators exhaust the heap before the concurrent collector is done) the overhead incurred by running the collector concurrently is up to 10%. Obtaining short pauses normally require a pay in the throughput. A 10% throughput reduction is considered a small pay for a two orders of magnitude reduction in the pause

[6] Measurements of _222_mpegaudio and _201_compress are not presented. _222_mpegaudio does not perform meaningful allocation activity. _201_compress heavily depends on a tracing collector as it creates substantial garbage cycles, so its measurements are not relevant for a comparison to a reference counting collector.

[7] Further uniprocessor results are given in our technical report [22].

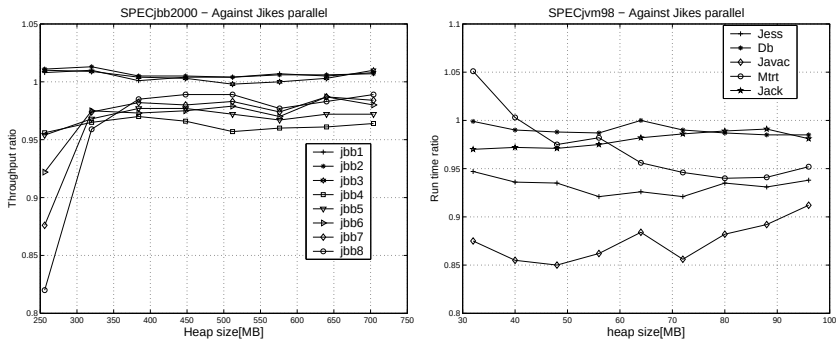

Fig. 8. SPECjbb2000 on a multiprocessor (left) and SPECjvm98 on a uniprocessor (right): comparison against Jikes parallel mark and sweep collector. The higher the ratio, the better the age-oriented collector performs compared to Jikes collector.

times (see pause time measurements in Section 5.3 below). The tight conditions highlight the advantage of parallel collectors in this setting. Parallel collectors always exploit all CPUs, while our on-the-fly collector uses only one processor while all program threads wait for free space to allocate. An exception is seen with the _213_javac benchmark. This benchmark creates cycles that are promoted to the old generation and die there. Since the age-oriented collector employs reference-counting with the old-generation, it does not collect these garbage cycles, causing frequent garbage collection invocations.

5.3 Pause times

Table 1 presents the maximum pause times of the age-oriented collector and Jikes parallel collector. Pauses were measured with a 64MB heap for SPECjvm98 benchmarks, and a 256MB heap for SPECjbb2000 with 1, 2, and 3 warehouses. For this number of threads, no thread gets swapped out, and so pauses are due to the garbage collection only. If we run more program threads, large pause times (whose lengths depend on the operating system scheduler) appear because threads lose the CPU to other threads.

The maximum pause time of 2.1ms, measured for the age-oriented collector, is two orders of magnitude shorter than that of Jikes parallel collector. The length of the age-oriented pause time is dominated by the time it takes to scan the roots of a single thread (occurring in one of the handshakes). This operation also dominates the pause time of the previous on-the-fly collectors [19, 4], and thus their pause times are similar (see [19, 4] for specific measurements of pause times for these collectors). Hence, the age-oriented collector achieves a significant throughput improvement over the original reference counting collector and over the generational collectors, while retaining the short pause times.

It is important to note that the pauses induced by the collector do not happen frequently. If pauses of 2ms occurred once every 3ms, then pause times would loose their meaning and we should look at mutator's minimum utilization

(MMU). However, in our case, the pauses form a negligible part of the collection cycle, and are split far apart from each other.

Maximum pause time(ms)	compress	jess	db	javac	mtrt	jack	jbb-1	jbb-2	jbb-3
Age-oriented	1.0	1.7	1.1	2.1	1.4	1.2	1.1	1.4	1.9
Jikes Parallel	195	261	188	643	225	376	322	417	511

Table 1. Maximum pause time in milliseconds

6 Related work

Generational garbage collection was introduced by Lieberman and Hewitt [20], and the first published implementation was by Ungar [27]. Both algorithms aimed to reduce the running time of most collections by focusing on the young objects.

Appel [2] presented a generational collector with variable young generation size: all its free space is devoted to the young generation. When the young generation becomes full, it collects the young generation, copying surviving objects to the older generation, and reducing the young generation size by this space. Major collections are executed only when the old generation occupies the entire heap. We push this idea further by proposing to always collect the old generation together with the young generation to make room for a large young generation.

Demers, et al. [10] presented a generational collector which does not move objects, hence appropriate for conservative garbage collection. They partition the heap logically (instead of physically separating between generations) by keeping a bit per object indicating whether it is young or old. We adopt this idea. However, their collector is not concurrent.

The study of on-the-fly garbage collectors was initiated by Steele and Dijkstra, et al. [25, 26, 11] and continued in a series of papers culminating in [14, 5, 17, 19, 3]. The advantage of an on-the-fly collector over a parallel collector and other types of concurrent collectors [6, 23, 9, 15, 16, 18], is that it avoids the operation of stopping all the program threads and incurs very short pauses.

Incorporations of generational collectors into on-the-fly collectors were done by Domani et al. [14], and by Azatchi and Petrank [4][8]. Both works employed fixed-sized young generation and both showed that combining generations with on-the-fly collectors may be useful. Domani et al. used the Doligez-Leroy-Gonthier mark and sweep collector [13, 12] both for the collection of the young generation and the collection of the full heap. The generational collector of [4] used the same basic collectors that we use here for the age-oriented collector. Results show that using these collectors for an age-oriented collection is more efficient than using them for a generational collection.

Blackburn and McKinley [8] implemented a uniprocessor stop-the-world generational collector with reference counting for the old generation and copying for

[8] A partial incorporation of generations with an on-the-fly collector 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.

the young. Their goal was to shorten the pauses a stop-the-world reference counting incurs, while obtaining good throughput. They run part of the old generation collection together with the young collection in order to avoid the need for a full collection that requires a long pause. The (controlled) pause times they obtain are an order of magnitude larger than those obtained by on-the-fly collectors.

7 Conclusion

We have proposed a framework of garbage collectors called age-oriented collectors. These collectors exploit the generational hypothesis in a different manner than standard generational collectors. Instead of running frequent young generation collections, the entire heap is collected infrequently, but young objects are treated differently from old objects. An age-oriented collector does not need to record inter-generational pointers, and avoids the overhead of initiating frequent young generation collections. The most fitting use of age-oriented collectors is with on-the-fly collectors and particularly when the old generation is collected via reference counting.

We have designed and implemented an instantiation of an age-oriented collector, based on the reference counting collector of [19] and the tracing collector of [3], in which reference counting collects the old objects and mark and sweep collects the young objects. This age-oriented collector was implemented on the Jikes RVM. Our measurements show that this collector maintains the short pauses of the original collectors and significantly outperforms both the original reference counting collector as well as the generational variant.

Acknowledgement We thank Elliot (Hillel) Kolodner for helpful discussions, and Michael Philippsen for the numerous helpful remarks on improving the readability of this manuscript.

References

1. Bowen Alpern, C. R. Attanasio, Anthony Cocchi, Derek Lieber, Stephen Smith, Ton Ngo, John J. Barton, Susan Flynn Hummel, Janice C. Sheperd, and Mark Mergen. Implementing Jalapeño in Java. In *ACM Conference on Object-Oriented Systems, Languages and Applications*, 34(10), pages 314–324, 1999.
2. Andrew W. Appel. Simple generational garbage collection and fast allocation. *Software Practice and Experience*, 19(2):171–183, 1989.
3. Hezi Azatchi, Yossi Levanoni, Harel Paz, and Erez Petrank. An on-the-fly mark and sweep garbage collector based on sliding view. In OOPSLA [21].
4. Hezi Azatchi and Erez Petrank. Integrating generations with advanced reference counting garbage collectors. In *Proceedings of the 12th International Conference on Compiler Construction, CC 2003*, volume 2622 of *LNCS*, pages 185–199, 2003.
5. David F. Bacon, Clement R. Attanasio, Han B. Lee, V. T. Rajan, and Stephen Smith. Java without the coffee breaks: A nonintrusive multiprocessor garbage collector. In *Proceedings of Conference on Prog. Lang. Design and Impl.*, 2001.
6. Henry G. Baker. List processing in real-time on a serial computer. *Communications of the ACM*, 21(4):280–94, 1978.

7. Stephen M. Blackburn, Richard Jones, Kathryn S. McKinley, and J. Eliot B. Moss. Beltway: Getting around garbage collection gridlock. In *Proceedings of SIGPLAN 2002 Conference on Prog. Lang. Design and Impl.*, pages 153–164, 2002.
8. Stephen M. Blackburn and Kathryn S. McKinley. Ulterior reference counting: Fast garbage collection without a long wait. In OOPSLA [21].
9. Hans-Juergen Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. *ACM SIGPLAN Notices*, 26(6):157–164, 1991.
10. Alan Demers, Mark Weiser, Barry Hayes, Daniel G. Bobrow, and Scott Shenker. Combining generational and conservative garbage collection: Framework and implementations. In *17 ACM Symp. on Prin. of Prog. Lang.*, pages 261–269, 1990.
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 *21 ACM Symp. on Principles of Prog. Lang.*, 1994.
13. Damien Doligez and Xavier Leroy. A concurrent generational garbage collector for a multi-threaded implementation of ML. In *the Twentieth ACM Symp. on Principles of Prog. Lang.*, pages 113–123. January 1993.
14. Tamar Domani, Elliot Kolodner, and Erez Petrank. A generational on-the-fly garbage collector for Java. In *Proceedings of SIGPLAN 2000 Conference on Programming Languages Design and Implementation.*
15. Toshio Endo, Kenjiro Taura, and Akinori Yonezawa. A scalable mark-sweep garbage collector on large-scale shared-memory machines. In *Proceedings of High Performance Computing and Networking (SC'97)*, 1997.
16. Christine Flood, Dave Detlefs, Nir Shavit, and Catherine Zhang. Parallel garbage collection for shared memory multiprocessors. In *Usenix Java Virtual Machine Research and Technology Symposium (JVM '01)*, April 2001.
17. Richard L. Hudson and J. Eliot B. Moss. Sapphire: Copying GC without stopping the world. In *Joint ACM Java Grande — ISCOPE 2001 Conference.*
18. Elliot K. Kolodner and Erez Petrank. Parallel copying garbage collection using delayed allocation. In *Parallel Processing Letters*, volume 14, June 2004.
19. Yossi Levanoni and Erez Petrank. An on-the-fly reference counting garbage collector for Java. In *ACM Conf. on Object-Oriented Systems, Lang. & Appl.*, 2001.
20. Henry Lieberman and Carl E. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, 1983.
21. *OOPSLA'03 ACM Conf. on Object-Oriented Systems, Lang. & Applications*, 2003.
22. Harel Paz and Erez Petrank. Age-oriented garbage collection. Technical Report CS-2003-08, Technion, Israel, October 2003. http://www.cs.technion.ac.il/-users/wwwb/cgi-bin/tr-info.cgi?2003/CS/CS-2003-08.
23. Tony Printezis and David Detlefs. A generational mostly-concurrent garbage collector. In Tony Hosking, editor, *Proceedings of the Second International Symp. on Memory Management*, volume 36(1) of *ACM SIGPLAN Notices*, October 2000.
24. SPEC Benchmarks. Standard Performance Evaluation Corporation. http://www.spec.org/, 1998,2000.
25. Guy L. Steele. Multiprocessing compactifying garbage collection. *Communications of the ACM*, 18(9):495–508, September 1975.
26. Guy L. Steele. Corrigendum: Multiprocessing compactifying garbage collection. *Communications of the ACM*, 19(6):354, June 1976.
27. David M. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices*, 19(5):157–167, April 1984.