Age-Oriented Garbage Collection

HAREL PAZ
Technion - Israel Institute of Technology
and
EREZ PETRANK
Technion - Israel Institute of Technology

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 (weak) hypothesis to obtain better efficiency. We then instantiate this approach with reference counting, and obtain 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.

Categories and Subject Descriptors: D.3.4 [Programming Languages]: Processors- Memory management (garbage collection)

General Terms: Algorithms, Design, Languages, Performance
Additional Key Words and Phrases: Runtime systems, Memory management, Generational garbage collection, On-the-fly garbage collection

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. Generational garbage collection [Lieberman and Hewitt 1983; Ungar 1984] ameliorate 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 reclaiming those objects most likely to be garbage, i.e., young objects. Generational collectors segregate objects according to their age into two or more regions of the heap called generations, and run frequent collections of the young generation, where most objects are likely to be reclaimable. Keeping the young generation small, we get frequent short collections, that efficiently make room for further allocations. The older generation is collected seldom when space is exhausted, making the long pauses infrequent.

Author’s address: E. Petrank, Dept. of Computer Science, Technion - Israel Institute of Technology, Haifa 32000, Israel. 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.
If the generational hypothesis (i.e., that most objects die young) is indeed correct, we get several advantages. First, the original goal of 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).

Generational collectors tend to be more efficient, however, not always so, as they require some overhead. One major overhead is the manipulation of inter-generational pointers. These are pointers that point from one generation to another. If the young generation is collected while the old generation is not, then pointers from the old generation to the young generation 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. A large body of papers deal with reducing this overhead by using efficient methods (e.g., card marking).

The second major overhead of generational collection is the frequent initiation of young generation collections. Even though the young generation is small, and the collection is rather fast, it repeatedly involves garbage collection initiation, which means some synchronization with the program threads, the marking of all roots, etc. To eliminate this problem, several modern commercial JVM’s either give up generations altogether or use a large young generation. Using a large young generation implies longer pauses for collecting the young generation, but less frequent collections.

1.1 On-the-Fly Garbage Collection

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.

The study of on-the-fly garbage collectors was initiated by Steele and Dijkstra, et al. [Steele 1975; 1976; Dijkstra et al. 1978] and continued in a series of papers [Gries 1977; Ben-Ari 1982; 1984; Kung and Song 1977; Lamport 1976; Doligez and Leroy 1993; Doligez and Gonthier 1994; Levanoni and Petrank 2001; Domani et al. 2000; Bacon et al. 2001]. The advantage of an on-the-fly collector over a parallel collector and other types of concurrent collectors [Baker 1978; Ellis et al. 1988; Moon 1984; Printezis and Detlefs 2000; Boehm et al. 1991], is that it avoids the operation of stopping all the program threads. Such an operation can be costly, and it usually increases the pause times.

1.2 This work

In this work, we investigate the use of generations with on-the-fly garbage collectors. When using on-the-fly collectors the pauses are already very short and we are only
interested in generations for their possible positive influence on the throughput. More generally, we investigate how to best exploit the generational hypothesis in an environment where only the throughput matters (and pauses do not). Previous work provides evidence that generations may be useful in this setting [Domani et al. 2000; Azatchi and Petrank 2003]. However, in light of the disadvantages of generational collectors the question arises whether something better can be done to exploit the generational hypothesis. The focus of this paper is a proposal of such new approach denoted age-oriented garbage collection and an instantiation of this approach with reference counting.

An age-oriented collector partitions the objects in the heap into generations as a generational collector does (either by space segregation or by attaching an age attribute to each object). However, unlike generational collectors, an age-oriented collector does not allow frequent young collections. In fact, the young generation is never collected on its own but only together with the old generation. The heap is collected when allocation space is exhausted. When that happens, the whole heap is collected\(^1\). But then, the collector distinguishes between collecting old and young objects. By the generational hypothesis, many objects die young. Therefore, among the young objects we can expect high death rate. Using this hypothesis the collector of the old generation is selected to be one that does well with high live rate and the collector of the young generation is selected to be one that does well with high death rate.

To explain this approach better let us compare it to a standard generational collector. A generational collector frequently collects the young objects, and seldom collects the full heap. Sometimes, two different collectors are used: one handles the young generation only and the other handles the full heap. In our approach there are no young generation collections. All collections handle the whole heap and are run infrequently only when the allocation space is exhausted. In each such collection there are two collectors: one that handles the young generation only and one that handles the old generation only.

Like generational collection, the age-oriented approach is a framework that may be instantiated in many ways. In order to instantiate an age-oriented collection framework one should determine which collector is used for the old generation and which collector is used with the young generation. One should check if recording inter-generational pointers is required when integrating the two collectors and then decide which method might be used for that. One sensible example for a collector combination may be to use a mark and sweep collector for the old generation and a copying collector for the young generation. Another promising age-oriented collector, which we chose to develop and implement in this work, is one that uses reference counting for the old generation and mark and sweep for the young generation. Other combinations are possible, keeping in mind that the collector of the old generation should handle well long living objects, and the collector of the young generation should be able to reclaim most of the young objects efficiently.

The main advantage of age-oriented over generational collectors is that they do not

\(^1\)Note that a concurrent collector may be triggered earlier since the goal is to let it end the collection just as the program arrives at the first allocation failure. Still, the collection is triggered at the latest possible time.
not need to spend time initiating frequent young collections. Another advantage is that the mechanism of recording inter-generational pointers may sometimes be eliminated when the old and the young generations are collected simultaneously. The disadvantages of age-oriented collectors are that space is not frequently reused and that pauses are not kept short. The latter is not relevant for on-the-fly collectors and the effect of the first disadvantage is hard to predict without running measurements. To this end, we have designed and implemented a new on-the-fly age-oriented garbage collector with reference counting for the old generation and mark and sweep for the young objects. Reference counting is naturally suitable for a high live rate. First, it traverses through the dead objects and thus is not appropriate for high death rate. Second, it does not traverse the live objects, so it can handle well huge live heaps. The mark and sweep collector may benefit from high death rate since only the live objects are traversed2.

Our design of the age-oriented collector is based on the sliding views on-the-fly reference counting collector of [Levanoni and Petrank 2001] and on the generational combination of this collectors with the sliding views tracing collector [Azatchi et al. 2003] as described in [Azatchi and Petrank 2003]. Since these collectors do not move objects, there is no segregation between the old and the young objects [Demers et al. 1990; Domani et al. 2000; Azatchi and Petrank 2003]. Instead, each object has a designated bit signifying whether it is old or young. Another choice that was traditionally taken with on-the-fly generational collectors and we adopt is to employ a simple promotion policy in which any live young object (which survives a collection) is promoted.

The age-oriented collector designed according to the above guidelines, turned out to be a very efficient collector with very short pauses adequate for running on a modern multithreaded environment and modern SMP platforms. It is fully concurrent (on-the-fly): each program thread is stopped for a short time to cooperate with the collector, but the threads never need to be stopped at the same time. Our collector has the main deficiency of reference counting algorithm, i.e., its inability to reclaim cyclic structures. To overcome this problem, a tracing sliding view algorithm ([Azatchi et al. 2003]) is seldom run on the full heap as in the original reference counting algorithm of Levanoni and Petrank.

Our contributions are as follows. (1) The proposal of a new framework of collectors, denoted age-oriented collectors that uses the generational hypothesis in a different way than the traditional generational collector. The new framework is applicable whenever improving efficiency is important and short pause times do not matter. (2) A design of an instantiation of this framework for reference counting and mark and sweep. (3) An implementation of the age-oriented collector on the Jikes JVM and a performance comparison against the original reference counting base collector ([Levanoni and Petrank 2001]) and its generational variant ([Azatchi and Petrank 2003]).

We refer the reader to a detailed introduction to garbage collection algorithms in [Jones 1996].

2We remark that avoiding the sweep by using copying collectors may be even better for the young generation, but copying collectors do not naturally work well concurrently with program run.
1.3 Implementation and results
The age-oriented collector was implemented on Jikes [Alpern et al. 1999], a JVM system written entirely in Java (with some primitives for manipulating raw memory). The system was run on a 4-way IBM Netfinity server. We used the SPECjbb2000 benchmark and the SPECjvm98 benchmark suites. These benchmarks are described in detail in SPEC’s Web site [SPEC Benchmarks 2000].

In Section 7, we report the measurements we ran with our collector. It turns out that our algorithm achieves not only a significant throughput improvement over the original Levanoni and Petrank reference counting algorithm, but it is also noticeably more efficient than the generational extension made by Azatchi and Petrank. This throughput superiority does not affect the extremely low pause times of the original collector.

1.4 Organization
In Section 2, we introduce the age-oriented framework and our proposed instantiation with reference counting and mark and sweep. In Section 3, an overview of the original reference counting collector is presented. Motivation for the new collector and an overview of the collector algorithm is introduced in Section 4. We provide the algorithmic details and pseudo-code in Section 5. Several Java implementation issues are discussed in Section 6 and performance results are described in Section 7. Related work is discussed in Section 8. We conclude in Section 9.

2. THE AGE-ORIENTED FRAMEWORK AND THE PROPOSED INSTANTIATION
In this section, a short description of the age-oriented framework is provided (the full description was presented in the introduction), and then our instantiation of the age-oriented framework with reference counting for the old objects and mark and sweep for the young objects is described. More details appear in the following sections.

An age-oriented collector is a collector that keeps information about the age of objects. When space is exhausted, an age-oriented collector collects the full heap, but treats objects according to their age. In order to provide an instantiation of this framework a few choices have to be made. In particular:

(1) **Which age information is kept.** A coarse choice (which is most popular) is to distinguish young objects from old. A fine choice (which is not common) is to keep the exact age for each object, measured by the space allocated prior to its creation.

(2) **How this information is used during the collection.** An age-oriented collector treats differently objects of different age. The typical age-oriented collector uses one collector to collect the old objects and a different collector to collect the young objects.

(3) **Collectors used.** In the common usage of ages, in which a different collector is used for each set of ages (generation), a choice should be made on which collector to use with each generation. By the generational hypothesis, we expect the old generation to contain many living objects and the young generation to contain many dead (unreachable) objects. The collectors should be chosen adequately for these behaviors.
**Interactions between collectors.** More choices to be made include the way one collector gets information about relevant information from spaces it does not collect. A typical such information is a list of inter-generational pointers. The choice of collectors may determine what kind of such information is required and a choice must be made on how this information is kept.

A simple age-oriented collector, which we implement, is one that distinguishes old objects in the heap from those objects that have been created since the previous garbage collection. We choose to collect the old objects with a reference counting collector and the new objects with a mark and sweep collector. Recall that the original motivation was to improve the throughput while ignoring the pause times as appropriate for concurrent collectors. Therefore, we choose to use modern on-the-fly collectors. In particular, we adopt the sliding views reference counting collector [Levanoni and Petrank 2001] and the sliding views mark and sweep collector [Azatchi et al. 2003]. This way, we gain the best of both worlds: the negligible pauses of the on-the-fly collectors with the high throughput obtained from taking ages into account effectively. The resulting collector is fast, non-disruptive, and appropriate for a modern platform employing an SMP hardware with a multithreaded system.

Interestingly, using our choice of collectors, we do not need to modify the (common) write-barrier of the original collectors. The age-oriented collector uses the information logged by the original write-barrier wisely to determine the young objects referenced by the old ones. Thus, no extra overhead for manipulating inter-generational pointers is needed.

### 3. REVIEWING THE ORIGINAL COLLECTOR

In Section 4 below, we describe our age-oriented collector. For completeness, we start with a short review of the sliding views reference-counting collector. The full description of this collector appear in the original paper [Levanoni and Petrank 2001]. The age-oriented collector employs some simple modifications to this collector allowing tracing over the young generation, as described in section 4.

Levanoni and Petrank [Levanoni and Petrank 2001] have presented an on-the-fly reference counting algorithm (the sliding views algorithm) with two interesting algorithmic properties: it saves many of the reference count updates, and it does not require synchronization in the write barrier. Thus, they obtain a non-disruptive efficient on-the-fly reference-counting collector. In this work, we use that collector and extend it to being age-oriented, thus, improving its efficiency.

The first contribution of the Levanoni-Petrank collector is the elimination of many redundant reference count updates. As a consequence, it presents a significant reduction in the overhead on transaction logging and counter updates (for common benchmarks), reducing the cost of the write barrier. 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: $RC(o_0) --, RC(o_1) ++, RC(o_1) --, RC(o_2) ++, \ldots, RC(o_n) ++$. However, only two are required: $RC(o_0) --$ and $RC(o_n) ++$. Furthermore, modern reference counting collectors let the program threads log updates in a transaction buffer. Using the above observation, we note that only one transaction needs to be logged,
whereas other collectors log all the n transactions. Measurements have shown that this improvement reduces the number of logs and counter updates by a factor of 100-1000 for standard Java benchmarks. This high saving is due to the fact that most updates are executed on newly created objects, and such updates require no logging.

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

Using the write barrier described above, the mutators record all heap objects whose pointer slots are modified between one collection to the next. The recorded information is the address of the modified object as well as the values of the object’s pointer slots before the current modification. As mentioned, a dirty flag is used to let only one record be kept for any modified object. The analysis shows that (seldom) races may cause more than one record be created for an object, but all such records contain the same information. The records are written into a local buffer with no synchronization.

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

A collection begins by taking a sliding-view of the heap. A sliding-view is essentially a non-atomic snapshot of the heap. It is obtained incrementally, i.e., the mutators are not stopped simultaneously. Such a non-atomic snapshot introduces a correctness danger: objects reachability may be wrongly computed by the collector because pointers keep changing in the heap. A solution to this problem is the snooping mechanism. While the view is being read from the heap, the snooping mechanism (via the write-barrier) records (locally) any object to which a new reference is created in the heap. Snooped objects are not reclaimed in the current collection cycle. It is shown in [Levanoni and Petrank 1999], that this mechanism ensures correctness.

To achieve cooperation between the collector and the program threads, handshakes are used. During a handshake, each thread is halted (separately, not simultaneously) for a short pause to cooperate with the collector. During a halt, data may be exchanged between the collector and the program threads. The Levanoni-Petrank collector employs four handshakes during the collection cycle.

The collection starts with the collector raising the Snoop flag of each thread, signaling to the mutators that it is about to start computing a sliding-view. During

\[\text{This actual property requires some care, see the original paper [Levanoni and Petrank 2001]}\]
the first handshake, mutator local buffers are retrieved and then are cleared. There are two such buffers for each mutator. YoungObjects \(i\) contains all objects created since the last collection by Mutator \(i\) and Updates \(i\) contains all objects modified by Mutator \(i\) since the last collection together with their previous sliding view non-null pointer slot values. Next, the dirty flags of the objects listed in the buffers (both the YoungObjects and the Updates buffer) are cleared while the mutators are running. Most of the clearing operations in this stage clear dirty bits created during the previous collection cycle, as intended. But some clearing are done to objects that have been dirtied concurrently by the running program threads. Such dirty bits should not be cleared. The collector proceeds by fixing this extra clearing. It reads the new local buffers from all mutators in a second handshake and sets the dirty bits for the (small number of) objects that have been dirtied and logged in the buffers concurrently with the collector clearing process. A third handshake with no specific operation is executed to make sure that the reinforced dirty bits are visible to all mutators. A fourth handshake is used to scan threads local states and objects directly reachable from the roots are marked as Roots.

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

Deciding which objects \(rc\) to increment, i.e., what are the children of an object \(O\) in the current sliding view is a bit more involved, since we cannot assume that all these children appear in some given list. If \(O\) has not been modified since the beginning of this collection then its pointer slots values may be read from the heap. If it has been modified, then the values of its pointer slots in the current sliding view must be obtained from the new updates buffers currently being written by the threads. Note that care is required here since threads are modifying the dirty bit while the collector is trying to determine if the object is currently dirty or not. This possible race is solved by the special way the collector checks these objects. The collector first checks if the object’s dirty flag is set. If it is set, then the values of this object may be obtained from the current local buffers\(^4\). Otherwise, we take a temporary replica of the object and check again if it is dirty. If not, then we may use the replica for finding the slots values. Otherwise, the object got dirty while the replica was taken, then we must read the buffer values.

A collection cycle ends with reclamation which recursively free any object with zero \(rc\) field which is not marked as local and is not snooped.

4. AGE-ORIENTED ALGORITHM OVERVIEW

We now describe the new collector. Assuming the weak generational hypothesis and in light of the actual measurements we ran, it holds that most objects created in the

\(^4\)An efficient way of obtaining these buffer values is described in section 5.1 below.
current cycle are not reachable. Thus, accounting for reference count updates due to pointer modifications in these objects is clearly wasteful. It is highly inefficient to go over the vast number of dead young objects, increment the \( rc \) of their direct descendants, only to find out in the next stages of the collection that these young objects are dead and then decrement the \( rc \) of all their descendants (before deleting them). The problem in the original reference counting collection is that we do not know in advance which of the young objects are dead, and which are reachable. An additional bothering problem is that unreachable young objects may form a cyclic structure. In this case, the update of the reference counts will not allow reclaiming them until the next tracing collection is invoked.

The age-oriented collector solves these problems directly by tracing the reachable young objects and ignoring the rest. Thus, reference counts are not updated due to modified pointers of unreachable young objects and cyclic structures are reclaimed before becoming a nuisance to the reference counting collector.

In the beginning of the collection, the original reference counting collector marks the snooped objects and the objects pointed by the roots as locals. The age-oriented collector buffers into a special separate buffer each such object which is also a young-object. Next, as in the original collector, \( rc \) adjustments of objects logged in the \( Updates \) buffer is performed: the collector decrements the \( rc \) of theirs slots values in the previous sliding-view and increments the \( rc \) of theirs current sliding-view slots values. The age-oriented collector uses this phase to detect more roots for the tracing of the young generation. In particular, each young object whose \( rc \) is incremented is also buffered (unless it was already buffered before). Therefore, at the end of this stage, each young object which has a non-zero \( rc \) or marked local, is buffered. These are exactly the root objects for the young generation tracing collector. Next, the age-oriented collector traces all buffered young objects. In order to trace an object, we use ideas similar to [Azatchi et al. 2003]. Its descendants are determined according to the current sliding view. Namely, if the object is not dirty, then we may read it to find its descendants. However, if it was modified since the beginning of the current collection (and is dirty) we use its recorded slots in the update buffers, representing its state in the current sliding view. For each traced object, the \( rc \) of its descendants is incremented, and each descendant that is young and not yet traced, is traced recursively. When the trace terminates, each object marked as local or having a non-zero \( rc \) is considered alive, and we can safely reclaim each (young or old) object which does not fulfill neither of these conditions.

The partitioning to young and old generations is logical. An object is young if it appears in the created buffer. A new object is put into this buffer upon creation and the buffer is cleared during garbage collection. Thus, each created object belongs to the young generation, and if it survives a collection it is considered old in the next collection.

4.1 A race condition

Using the above strategy with the Levanoni-Petrank on-the-fly collector is susceptible to a (rare) race condition. To form the sliding views without stopping the mutators simultaneously, the collector reads the mutators’ buffers (\( YoungObjects \) and \( Updates \)) while the mutators run. Thus, while the buffers are being read by the collector from one thread, objects are being created and modified by other
(program) threads. The problem occurs when a new object is being created by a
mutator during the time the collector is performing the first handshake and collect-
ing the local buffers from the mutators one by one.

Here is the problematic scenario. Suppose a collection is started and the collector
reads the YoungObjects (and Updates) buffers of mutator \( T_1 \). After this happens,
any new object created by \( T_1 \) is considered young for the next collection, and
not for the current one. Indeed, there is no danger in erroneously reclaiming this
object. However, we show that a specific race may cause its descendants to be
erroneously reclaimed in the next collection. Suppose that \( T_1 \), after delivering its
buffers to the collector in the first handshake, creates a new object \( O_1 \) and logs
it in its YoungObjects buffer. This log will be read only in the next garbage
collection. Suppose also that another thread \( T_2 \), whose buffers were not yet read
by the collector, modifies an old object \( O_2 \) to reference \( O_1 \). The record of this
modification is logged into \( T_2 \)’s Updates buffer and is read in the current collection.
(This record is processed in the current collection and will not appear in the next
collection.) Note the source of confusion: because \( T_2 \)’s buffers are read after \( T_1 \)’s
buffers are read, \( O_1 \) is new in the next collection but the creation of a pointer to
it from an old object is already processed by the collector in the current collection.
During the current collection, the reference count of the created object \( O_1 \) will
be incremented and it will not be collected until this reference count is properly
decremented. However, we expect newly created objects that are referenced from
the old generation to be traced. Tracing is normally triggered by the appearance of
created objects in the updates buffer, signifying an inter-generational pointer. But
with the race described, the record that was created in the updates buffer is already
processed in the current collection and will not appear again in the updates buffer
when the next collection starts and the object \( O_1 \) will be processed as a new object.
The outcome of this race is that the descendants of object \( O_1 \) will not be traced in
the next collection, which may lead to a premature collection of objects.

To ensure that all necessary tracing is executed properly, we add a specific treat-
ment for objects with the above properties. We identify them by their being young
with non-zero \( rc \) in the beginning of a collection (due to a race in the previous
collection). Note that this can only happen because of this race. Normally, \( rc \)’s
of young objects are zero since reference count updates have not been executed
since they were created. Thus, during the Clear-Dirty-Marks procedure we check if a
young object has \( rc > 0 \) and if so, this young object is pushed into the mark stack.
We ran this check during the Clear-Dirty-Marks procedure since the young buffer
is traversed at that time. From correctness perspective this check may be run at
any time after the race is no longer possible (i.e., after the second handshake of the
previous cycle) and before updating \( rc \)’s of young objects in the current collection
cycle.

4.2 Advantages
Processing \( rc \) adjustments only for reachable young objects is expected to save a
large fraction of \( rc \) increments, as most young objects are not reachable. Moreover,
the releasing process is much easier, instead of tracing through the dead objects
to perform \( rc \) decrements as in a standard reference counting algorithm, the age-
oriented may sweep the heap fast.
The original algorithm’s write-barrier is used with no modifications, as it fits naturally to the new age-oriented collector. No extra treatment of inter-generational pointers is required during the program run. The roots for the collection are young object which have a non-zero \( rc \) and young objects which are marked local. No additional data structures for maintaining the inter-generational pointers are required.

The new collector maintains the advantages of the Levanoni-Petrank collector. In particular, it is adequate for a multithreaded environment and a multiprocessor platform, it has excellent throughput, as the write barrier requires no synchronization operations, and it has the short pauses of an on-the-fly collector.

As any reference counting collector, this age-oriented algorithm cannot reclaim cyclic data structures. To reclaim such structures, the tracing sliding view algorithm of [Azatchi et al. 2003] is run once in a while on the full heap. Its use is rare since cyclic structures in the young generation are collected immediately.

The measurements (in Section 7) show that the age-oriented collector outperforms the original Levanoni-Petrank collector. Comparison against a similar generational collector [Azatchi and Petrank 2003] demonstrates superiority in this case too. This provides a good indication that this framework is a useful framework to be considered in future work.

5. THE GARBAGE COLLECTOR DETAILS

In this section, pseudo-code and explanations regarding the new on-the-fly age-oriented collector are provided. In order to stress the additions to the original reference counting collector, we adopt a convention of adding an asterisk to any line in the age-oriented algorithm code that differs from the original Levanoni-Petrank collector.

5.1 The log-pointer

The original reference counting algorithm, as well as the new age-oriented collector require maintaining a dirty bit signifying whether an object has been modified since the most recent collection started. During the first modification of an object in a cycle, its pointers are recorded in the \( updates \) buffer and its dirty bit is set. We follow [Levanoni and Petrank 2001; Azatchi et al. 2003] by choosing to dedicate a full word to keep the dirty bit. Indeed, this consumes space, but it allows keeping information about the dirty object. In particular, this word is used to keep a pointer into the thread’s local buffer where this object’s pointers have been logged. A zero value (a null pointer) signifies that the object is not dirty (and not logged). We call this word the \( LogPointer \). Justification to this choice is provided in the original sliding views papers and is not repeated here.

5.2 Mutator cooperation

The mutators need to execute garbage-collection related code on three occasions: when updating an object, when allocating a new object and during handshakes. This is accomplished by the \( Update \) (Figure 1) procedure, the \( New \) (Figure 2) procedure and the handshake mechanism, respectively. The \( Update \) and \( New \) operations never interleave with a handshake. Namely, cooperation with a handshake waits until a currently executed \( Update \) or \( New \) operation finishes. The \( Update \) and \( New \)
Procedure Update\( (\text{obj}: \text{Object}, \text{offset}: \text{int}, \text{new}: \text{Object}) \)
begin
1. if \( \text{obj}.\text{LogPointer} = \text{NULL} \) then // object not dirty
2. \( \text{TempPos} := \text{CurrPos} \)
3. // take a temporary replica of the object
4. foreach field \( \text{ptr} \) of \( \text{obj} \) which is not NULL
5. \( \text{Updates}_i[+\text{TempPos}] := \text{ptr} \)
6. // is it still not dirty?
7. if \( \text{obj}.\text{LogPointer} = \text{NULL} \) then
8. // add pointer to object
9. \( \text{Updates}_i[+\text{TempPos}] := \text{address of } \text{obj} \)
10. // committing the replica
11. \( \text{CurrPos} := \text{TempPos} \)
12. // set dirty
13. \( \text{obj}.\text{LogPointer} := \text{address of } \text{Updates}_i[\text{CurrPos}] \)
14. \( \text{write}( \text{obj}, \text{offset}, \text{new}) \)
15. if \( \text{Snooped}_i \) and \( \text{new} \neq \text{NULL} \) then
16. \( \text{Snooped}_i := \text{Snooped}_i \cup \{ \text{new} \} \)
end

Fig. 1. Mutator code: Update Operation

operations are exactly equal to those used in the original sliding views reference
counting algorithm [Levanoni and Petrank 2001].

5.2.1 Write barrier. Procedure Update (Figure 1) is activated at pointer
assignment and its main task is to record the object whose pointer is modified
(i.e., log objects values at the sliding views). We stress that the write barrier (the
Update protocol) is only used with heap pointer modification. Modifications of local
pointers in the registers or stack are not monitored. Going through the pseudo-
code, we see that each object’s \( \text{LogPointer} \) is optimistically probed twice (lines
1 and 7) so that if the object is dirty (which is often the case), then the write barrier
is extremely fast. If the object was not logged (i.e., the \( \text{LogPointer} \) of an object
is \( \text{NULL} \)) then after the first probe, the objects values are recorded into the local
\( \text{Updates}_i \) (lines 3-5). The second probe at line 7 ensures that the object has not
yet been logged (by another thread). If \( \text{LogPointer} \) is still \( \text{NULL} \) (in the second
probe), then the recorded values are committed as the buffer pointer is modified
(line 11). In order to be able to distinguish later between objects and logged values,
in line 9 we actually log the object’s address with the least significant bit set on
(while values are logged with least significant bit turned off). Then, the object’s
\( \text{LogPointer} \) field is set to point to these values (line 13). After logging has occurred,
the actual pointer modification happens. Finally, from the time a collection begins
until marking the roots of the mutators, the snoop flag is on. At that time, the
new target of the pointer assignment is recorded in the local \( \text{Snooped}_i \) buffer. This
happens in lines 15-16. The variables \( \text{Updates}_i, \text{CurrPos}, \text{Snoop}_i \), and \( \text{Snooped}_i \)
are local to the thread.

We do not elaborate on the properties of the write-barrier, on why it works in a
multithreaded environment, etc. We focus on the modifications required to obtain
an age-oriented collector. A thorough discussion of the write barrier appears in the
Procedure **New**(size: Integer, obj: Object)
begin
1. Obtain an object obj of size size from the allocator.
2. YoungObjects[++] := address of obj
3. obj.LogPointer = address of YoungObjects[++]
4. return obj
end

Fig. 2. Mutator code: Allocation Operation

original paper [Levanoni and Petrank 2001].

5.2.2 Creating a new object. Procedure **New** (Figure 2) is used when allocating an object. After Thread $T_i$ creates an object, the object’s address is logged into a local (to the mutator) buffer called YoungObjects$_i$. This buffer includes the addresses of the objects which would be considered as young objects in the next collection cycle. This logging will tell us which objects are young. For the reference counting this means that reference counts of descendants of these objects (if they survive the next collection) must be updated. There is no need to record their children slots values as they are all null at creation time. In addition, the LogPointer of a newly allocated object is modified to record its log address (so that future assignments to this object, won’t activate the write-barrier and log it again in the updates buffer).

5.2.3 The handshake mechanism. Our handshake mechanism is the same as the one employed by the Doligez-Leroy-Gonthier collector [Doligez and Leroy 1993; Doligez and Gonthier 1994]. The mutator threads are never stopped together for cooperating with the collector. Instead, threads are suspended one at a time for the handshake. The stopping of the thread is not allowed while it is executing the write barrier or while it is creating a new object. While a thread is suspended, the collector executes the relevant actions for the handshake and then the thread is resumed. The collector repeats this process until all threads have cooperated. At that time, the handshake is completed.

5.3 Phases of the collection
The collector algorithm runs in phases as follows.

— **Start snooping**: raising the Snoop$_i$ local flag of each mutator, which activates the snooping mechanism.

— **First handshake**: during this handshake each mutator is stopped and its log buffers (YoungObjects$_i$ and Updates$_i$), are accumulated by the collector.

— **Clear dirty marks**: The collector clears the dirty marks of all objects previously recorded in the buffers.

— **Second handshake**: during this handshake each mutator is stopped in order to reinforce logs (to its current Updates$_i$ buffers) which were cleared during the clear dirty marks phase.

— **Third handshake**: no operation (empty) handshake to make sure that the proper dirty marks are visible by all mutators.
Procedure Collection-Cycle
begin
1. Initiate-Collection-Cycle // 1st handshake
2. Clear-Dirty-Marks
3. Reinforce-Clearing-Conflict-Set // 2nd and 3rd handshake
4. Mark-Roots // 4th handshake
5. Update-Old-Reference-Counters
*6. Trace-Young
*7. Reclaim-Young-Garbage
8. Reclaim-Old-Garbage
9. Prepare-Next-Collection
end

Fig. 3. Collector code- Collection Cycle

—Fourth handshake: during this handshake each mutator is stopped and the local roots of each mutator are marked. Also, the Snoopi local flag is cleared.

—Update reference counts of old objects: the collector adjusts the rc fields of old objects’ descendants.

—Trace young objects: the collector traces the live young objects (while incrementing their rc).

—Reclaim young garbage: during this phase, the collector reclaims dead young objects. (Objects release in this phase is extremely fast, since no recursive deletion is required.)

—Reclaim old garbage: during this phase, the collector reclaims old objects (and their descendants using recursive deletion) which have a zero rc and which are not referenced by the system roots.

—Prepare next collection: the collector prepares its buffers to the next collection.

5.4 Collector code

Collector’s code for cycle k is presented in Procedure Collection-Cycle (Figure 3). Let us briefly describe each of the collector’s procedures.

Procedure Initiate-Collection-Cycle (Figure 4) raises first the (local to mutator) Snoopi flag, signaling the mutators that they should start snooping all stores into heap slots. Then the first handshake is carried out to gather the local buffers of the threads.

Procedure Clear-Dirty-Marks (figure 5) clears all dirty marks that were set by mutators prior to responding to the first handshake. This stage takes place while the mutators are running. When handling the YoungObjects buffer, we also push into markStack each object which has a non-zero rc. Objects pushed into markStack would be traced later. A young object can have a non-zero rc due to a race condition (as explained in 4.1). Other young objects (which have a zero rc) are optimistically considered dead (and thus are not marked).

Note that lines 5-7 (marked with asterisk) are relevant only to the age oriented collector, as the original algorithm treats young objects similarly to old objects.
Age-Oriented Garbage Collection

Procedure *Initiate-Collection-Cycle*

begin
1. for each thread \( T_i \) do
2. \( \text{Snoop}_i := \text{true} \)
3. // first handshake
4. for each thread \( T_i \) do
5. suspend thread \( T_i \)
6. // copy (without duplicates).
7. \( \text{Updates} := \text{Updates} \cup \text{Updates}_i \)
8. \( \text{Updates}_i := \emptyset \) // clear buffer.
9. \( \text{YoungObjects} := \text{YoungObjects} \cup \text{YoungObjects}_i \)
10. \( \text{YoungObjects}_i := \emptyset \) // clear buffer.
11. resume thread \( T_i \)
end

Fig. 4. Collector code- Initiate-Collection-Cycle

Procedure *Clear-Dirty-Marks*

begin
1. for each object \( \text{obj} \in \text{Updates} \) do
2. \( \text{obj}.\text{LogPointer} := \text{NULL} \)
3. for each object \( \text{obj} \in \text{YoungObjects} \) do
4. \( \text{obj}.\text{LogPointer} := \text{NULL} \)
*5. if \( \text{obj}.\text{rc} > 0 \) then
*6. \( \text{obj}.\text{live} := \text{true} \)
*7. push \( \text{obj} \) into \text{markStack}
end

Fig. 5. Collector code- Clear-Dirty-Marks

**Procedure Reinforce-Clearing-Conflict-Set** (figure 6) implements the reinforcement step and assures that it is visible to all mutators. A second handshake takes place, during which thread buffers are read into \( \text{ClearConf lictSet} \). Then, LogPointres of objects logged into \( \text{ClearConf lictSet} \) are reinforced to point their current position in \( \text{Updates}_i \). Finally, the third handshake of the cycle takes place with no action in it. The reason for that handshake is that a thread can fall behind another thread by at most one handshake\(^5\). Thus, threads that have responded to the fourth handshake will not be interfered by operation carried out by threads during the clearing or reinforcement stages.

**Procedure Mark-Roots** (Figure 7) carries out the forth (third in Jikes) and last handshake during which the local \( \text{Snoop}_i \) flag is turned off and the thread local roots are accumulated into the \( \text{Roots} \) (global) buffer (the set of objects directly reachable from thread \( T_i \) is denoted \( \text{State}_i \)). Next, the \( \text{Snooped}_i \) buffer of each thread (containing snooped objects), is accumulated into \( \text{Roots} \), and then cleared.

\(^5\)In real implementation on Jikes this scenario cannot happen as explained in [Azatchi and Petrank 2003]. Thus, we do not have this empty handshake in our Jikes implementation, although we present it in our pseudo-code.
Procedure Reinforce-Clearing-Conflict-Set
begin
1. ClearConflictSet := ⊘
2. // second handshake
3. for each thread $T_i$ do
4. suspend thread $T_i$
5. ClearConflictSet :=
6. ClearConflictSet $\cup Updates_i[1...CurrPos_i]$
7. resume thread $T_i$
8. for each object $obj \in ClearConflictSet$ do
9. if $obj.LogPointer = NULL$ then // need to reinforce
10. $obj.LogPointer :=$
11. address of $obj$'s replica in $Updates_i$
12. // third handshake- empty handshake
13. for each thread $T_i$ do
14. suspend thread $T_i$
15. resume thread $T_i$
end

Fig. 6. Collector code- Reinforce-Clearing-Conflict-Set

Procedure Mark-Roots
begin
1. // fourth handshake
2. for each thread $T_i$ do
3. suspend thread $T_i$
4. $Snoop_i := false$
5. $Roots := Roots \cup State_i$ // copy thread local state.
6. resume thread $T_i$
7. for each thread $T_i$ do
8. // copy and clear snooped objects set
9. $Roots := Roots \cup Snooped_i$
10. $Snooped_i := ⊘$
*11. for each object $obj \in Roots$ do
*12. if $obj.live = false$ then
*13. $obj.live := true$
*14. push $obj$ into $markStack$
end

Fig. 7. Collector code- Mark-Roots

(for the next collection). Thus, during this procedure, the true root set of this collection cycle is being marked.

Lines 11-14, present a code for marking roots of the young generation. It is relevant only to the age-oriented collector: each young object belonging to $Roots$, which was considered until now as dead (i.e., has a zero $rc$, and thus was not marked as $live$ in procedure Clear-Dirty-Marks), is marked $live$ (as it belongs to the roots set and thus should be treated as alive) and is pushed into the $markStack$ (so it would be traced later).
Procedure Update-Old-Reference-Counters
begin
1. for each object obj whose replica rep in Updates do
2. // decrement previous values of the object obj
3. for each slot s in the replica of rep do
4. previous-value := read(s)
5. previous-value.rc --
6. if previous-value.rc = 0 then
7. add previous-value to ZCT
8. // increment reference count of sliding-view descendants
9. Increment-Descendants-RC(obj)
end

Fig. 8. Collector code- Update-Old-Reference-Counters

Procedure Increment-Descendants-RC(obj: Object)
begin
1. // Check if object has been modified
2. if obj.LogPointer = NULL then
3. // No - read its descendants from heap.
4. replica := copy(obj)
5. // Check again if copied replica is valid.
6. if obj.LogPointer != NULL then
7. // Object has been modified while being read. Get replica from buffers.
8. replica := getOldObject(obj.LogPointer)
9. else // Object has been modified. Use buffers to obtain replica.
10. replica := getOldObject(obj.LogPointer)
11. // Increment rc’s of descendants.
12. for each slot s in replica of obj do
13. curr := read(s)
14. curr.rc++
15. if curr.live = false then
16. curr.live := true
17. push curr into markStack
end

Fig. 9. Collector code- Increment sliding-view values

Procedure Update-Old-Reference-Counters and procedure Increment-Descendants-RC (figures 8-9) adjust the reference counters corresponding to the modified objects. It examines the objects which were modified since the previous collection cycle. Those objects must be logged to the Updates buffer. The rc of their children slots values in the previous sliding view should be decremented, whereas the rc of their current children slots values should be incremented. During rc adjustments, every object’s whose rc is decremented to 0, is inserted into the ZCT.

The procedure Increment-Descendants-RC (figure 9) performs rc increments of current sliding-view children slots values of a given object. An object may be
Procedure Trace-Young
begin
*1. while markStack is not empty
*2. obj := pop(markStack)
*3. Increment-Descendants-RC(obj)
end

Fig. 10. Collector code-Trace-Young

Procedure Reclaim-Young-Garbage
begin
*1. for each object obj ∈ YoungObjects do
*2. if obj.live = false then
*3. return obj to the general purpose allocator.
end

Fig. 11. Collector code-Reclaim-Young-Garbage

modified by mutators while the replica is taken. If this is the case, then its children slots at the current sliding view can be found by looking at the current collection cycle log entry which is pointed by the dirty flag (the LogPointer points to the logging location of this object). Lines 15-18, are related only to the age-oriented algorithm. In these lines, each object (whose rc was just incremented) which is not marked as live is marked as live and pushed into markStack. Note, that those objects are young object for which we have found evidence of being alive only now.

Procedure Trace-Young (figure 10) traces the live young objects while adjusting the rc of all their descendants. The roots of this tracing are the young objects for which we have found evidence of being alive (all those objects were pushed during the collection into markStack). Those objects are traced, while we also increment the rc of their current sliding-view slots values (using the procedure Increment-Descendants-RC presented in figure 9). No decrement is needed as those objects were created since the last collection.

Note that this procedure is related only to the age-oriented algorithm, while in the original algorithm we process the increments related to all the young objects.

Procedure Reclaim-Young-Garbage (figure 11) releases the young objects which are not marked as live. Note that releasing those objects is extremely fast, since we do not need to decrement rc of theirs descendant as done in traditional reference-counting systems (and thus no recursive deletion is needed).

Procedure Reclaim-Garbage (figure 12) releases unreachable old objects (and their descendants if needed). At first, we iterate through the ZCT buffer. Each object which has a positive rc field or is marked as Roots is deleted from the ZCT. Otherwise, its live flag is turned-off and it is collected by the Collect procedure (figure 13), which takes care of decrementing the rc of an object’s sliding-view values before releasing it (and performs recursive releases if necessary). Note, that after the process of reclaiming garbage is over, we clear the ZCT buffer.

Procedure Prepare-Next-Collection (Figure 14) inserts objects referenced solely by Roots (and thus having a zero rc), into the ZCT. In the next collection, those
Procedure Reclaim-Old-Garbage
begin
1. for each object \( obj \in ZCT \) do
2. \( \text{if } obj.\text{rc} > 0 \lor obj \in \text{Roots} \) then
3. \( ZCT := ZCT \setminus \{ obj \} \)
4. for each object \( obj \in ZCT \) do
   *5. \( obj.\text{live} := \text{false} \)
6. Collect\((obj)\)
7. \( ZCT := \emptyset \)
end

Fig. 12. Collector code- Reclaim-Old-Garbage

Procedure Collect\((obj : \text{Object})\)
begin
1. \( \text{if } obj.\text{LogPointer} \neq \text{NULL} \) then
2. Invalidate-Log-Entry\((obj.\text{LogPointer})\)
3. \( \text{replica} := obj.\text{LogPointer} \)
4. for each slot \( s \) in \( \text{replica} \) of \( obj \) do
5. \( \text{curr} := \text{read}(s) \)
6. \( \text{curr.} \text{rc} := \text{false} \)
7. \( \text{if } \text{curr.} \text{rc} = 0 \land \text{curr} \notin \text{Roots} \) then
8. Collect\((\text{curr})\)
9. \( \text{else} \)
10. for each slot \( s \) of \( obj \) do
11. \( \text{curr} := \text{read}(s) \)
12. \( \text{curr.} \text{rc} := \text{false} \)
13. \( \text{if } \text{curr.} \text{rc} = 0 \land \text{curr} \notin \text{Roots} \) then
14. Collect\((\text{curr})\)
15. return \( obj \) to the general purpose allocator.
end

Fig. 13. Collector code- Collect

Procedure Prepare-Next-Collection
begin
1. for each object \( obj \) in \( \text{Roots} \)
2. \( \text{if } obj.\text{rc} = 0 \) then
3. add \( obj \) to \( ZCT \)
4. \( \text{// clear buffers for next collection} \)
5. \( \text{Roots} := \emptyset \)
6. \( \text{Updates} := \emptyset \)
7. \( \text{YoungObjects} := \emptyset \)
end

Fig. 14. Collector code- Prepare-Next-Collection
objects would be examined when iterating through the ZCT. Next, the procedure cleans the global Roots, YoungObjects and Updates buffers.

6. AN IMPLEMENTATION FOR JAVA

We have implemented our algorithm in Jikes [Alpern et al. 1999], a Java virtual machine (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 implementing the handshakes of the garbage collection. In 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.

6.1 Memory allocator

Our implementation employs the non-copying allocator of Jikes, which is based on the allocator of Boehm, Demers, and Shenker [Boehm et al. 1991]. This allocator is well suited for collectors that do not move objects. Small objects are allocated from per-processor segregated free-lists build from 16KB blocks divided into fixed-size slots. Large objects are allocated out of 4KB blocks with first-fit strategy. This allocator keeps the fragmentation low and allows efficient reclamation of objects.

6.2 Sweeping the young generation on the Jikes platform

The pseudo-code described in Section 5 presents our collector in a general manner, appropriate for any platform. We now explain an implementation detail, which improve the throughput. In particular, a nice trick allows reclaiming unmarked objects efficiently. A similar method is used with other Jikes available collectors.

The Jikes allocator associates a block table entry with each occupied block. In the original implementation of the Levanoni-Petrank collector, an unmarked entry in the block table, signifies that the corresponding slot in the block is not allocated. When an object was allocated from a certain block, its corresponding entry in this block table was marked. That way, if this object would not be evacuated in the next collection, the allocator would identify this slot as occupied in the next time it allocates objects from this block. When the collector would evacuate an object, it would zero the relevant entry in the block table.

Since most of the young objects are freed during the next collection, the above implementation marks the entry of an allocated young object, only to usually unmark it when found unreachable in the coming collection. The alternative strategy we adopted (which is used also by the Jikes parallel mark and sweep collector) is not to mark slots’ entries during their allocation. Instead, during a collection, each young object (slot) which survives the collection is marked. The main advantage is that we do not need to mark or unmark unreachable young objects (slots). The fraction of objects that survive the young collection (and thus require marking in the block table) is a small fraction of the total objects allocated. Since our collector is on-the-fly, we should make sure that the allocator does not re-allocate an
object that was already allocated but not marked in the block table. We do that 
by allocating only from blocks which do not need to processed by the collector or 
those that the collector have already processed (i.e., the collector have marked the 
survived young objects from those blocks).

The outcome of this optimization is that there is no action to take when releasing
an un-reachable young object in the age-oriented algorithm. Therefore, there is
no need to iterate over the YoungObjects buffer in the procedure Reclaim-Young-
Garbage (figure 11), and thus this procedure is redundant.

For fairness of comparison, we have implemented this strategy not only on the
age-oriented collector, but also on the original Levanoni-Petrank algorithm, and its
generational extension.

7. MEASUREMENTS

Platform and benchmarks. We have taken measurements on a 4-way IBM
Netfinity 8500R server with a 550MHz Intel Pentium III Xeon processor and 2GB
of physical memory. The benchmarks we used were the SPECjvm98 benchmark
suite and the SPECjbb2000 benchmark. These benchmarks are described in de-
tail in SPEC’s Web site [SPEC Benchmarks 2000]. We feel that the multithreaded
SPECjbb2000 benchmark is more interesting, as the SPECjvm98 are more ap-
propriate for clients and our algorithm is targeted at servers. We also feel that there is
a dire need in academic research for more multithreaded benchmarks. In this work,
as well as in other recent work (see for example [Bacon et al. 2001; Domani et al.
2000]) SPECjbb2000 is the only representative of large multithreaded applications.

Testing procedure. We used the benchmark suite using the test harness, per-
foming 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).

Finally, to understand better the behavior of our collector under tight and relaxed
conditions, we tested it 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 used larger heaps, starting from 256MB heap
size and extending by 64MB increments until a final large size of 704MB.

The compared collectors. We tested our age-oriented concurrent collector
against 2 collectors: the Levanoni-Petrank reference-counting collector ([Levanoni
and Petrank 2001]), and the generational collector of Azatchi and Petrank ([Azatchi
and Petrank 2003]). Both collectors were also implemented in Jikes. We would
denote hereafter the Levanoni-Petrank collector as the original collector, and the
Azatchi-Petrank collector as the generational collector.

7.1 Server performance

We start with the SPECjbb2000 benchmark. SPECjbb2000 requires multi-phased
run with increasing number of warehouses. Each phase lasts for two minutes with
a ramp-up period of half a minute before each phase. The benchmark provides
a measure of the throughput and we report two throughput ratios improvements:
the first is the ratio of the age-oriented collector and the original Levanoni-Petrank
collector, while the second is the ratio between the generational collector and the
original collector. Thus, the higher the ratio, the better the collector performs, and
any ratio larger than 1 implies that the measured collector outperforms the original reference counting (Levanoni-Petrank) collector.

The measurements are reported for a varying number of warehouses and varying heap sizes in figures 15-22. In each figure, we report the measurement for a specific number of warehouses, i.e., a specific number of threads. The behavior of the collectors should be separated into two cases.

The first case is with 1-3 warehouses. In this case, since our machine has four processors, any of the three on-the-fly collectors runs on a spare processor. In this setting, the collectors do not differ much. If the collectors could handle all their work while mutators are running (except for handshakes), all 3 collectors would have achieved the same throughput (as they share similar allocator and write-barrier). This is indeed what we get with 1-2 warehouses. A throughput improvement (of usually 5%) begins only with 3 warehouses and especially on tight heaps, were both generational and age-oriented collectors outperform the original collector. The age-oriented does a little better. The reason for this improvement is that the original collector must deal now with work supplied by 3 mutators (i.e., more work for the collector), and thus mutators sometimes halt waiting for the collector to terminate its work.

The second case refers to 4-8 warehouses, where collectors do not run on a spare processor but rather share a processor with the program threads. Note, nevertheless, that we gave the collector (in this case) the highest priority, so that when a collection is triggered the collector would always get a dedicated processor. Thus, when the number of warehouses is four and up, the efficiency of the collector becomes more important: a collector should not only be able to handle all its work while mutators are running, but also as the collector becomes more efficient, a collection would consume less time, thus letting mutators use a larger fraction of the fourth processor (and therefore increasing the throughput). The results shows that the age-oriented collector and the generational collectors outperform the original collector (where the age-oriented collector usually obtains a performance improvement of 25%-40% over the original collector). The age-oriented collector substantially outperforms the generational collector. As with 3 warehouses, the superiority
Fig. 16. SPECjbb2000 on a multiprocessor: throughput ratio for 2 warehouses.

Fig. 17. SPECjbb2000 on a multiprocessor: throughput ratio for 3 warehouses.

Fig. 18. SPECjbb2000 on a multiprocessor: throughput ratio for 4 warehouses.
Fig. 19. SPECjbb2000 on a multiprocessor: throughput ratio for 5 warehouses.

Fig. 20. SPECjbb2000 on a multiprocessor: throughput ratio for 6 warehouses.

Fig. 21. SPECjbb2000 on a multiprocessor: throughput ratio for 7 warehouses.
of the age-oriented collector is usually higher with (relatively) small heaps where the collector efficiency is more significant.

**SPECjvm98 measurements.** The SPECjvm98 benchmarks provide a measure of the elapsed running time, which we report. Here, the smaller the better. In figure 23, we report the running time ratio between the age-oriented collector and the original collector, and in figure 24, we report the running time ratio between the age-oriented collector and the generational collector. For clarity of presentation, we report the inverse ratio, so that higher ratios still show better performance of our collector over the original (or the generational) collector, and ratios larger than 1 imply our collector outperforming the original (or the generational) collector.

We do not present measurements for the _222_mpegaudio and the _201_compress benchmarks. _222_mpegaudio does not perform meaningful allocation activity. The _201_compress heavily depends on a tracing collector as it creates substantial garbage cycles, and runs out of memory if its cycles are not reclaimed. Hence, when being run with the original collector, the backup tracing collector is applied constantly, so its measurements are not relevant for a comparison to a reference counting collector.

When running the SPECjvm98 benchmarks on a multiprocessor, we allow a designated processor to run the collector thread. Here again the collector runs concurrently with the program thread and good concurrency is the main factor in the comparison. Results show that our collector performs slightly better (usually wins by few percentages) than both the original collector and the generational collector. However, we would like to stress that such measurements are in favor of the original collector, since it does not show that our collector consumes much less resources of the spare processor than the original collector. Similar measurements on a uniprocessor look much different (as will be discussed below).

Table I addresses this exact issue. We have measured for both original and age-oriented collectors the time each one actually work, i.e., the amount of time in which the spare processor is in use. Our results demonstrate that the original collector works 2.41-18.67 times more than the age-oriented collector. Since SPECjvm98 runs use a designated processor, those numbers influence the throughput mainly if
the original collector could not perform all its work concurrently with the mutators (causing the mutators to stop waiting for memory space). That is why when running SPECjbb2000 with 4 warehouses or more, our graphs show a much substantial throughput superiority: in those cases more CPU time for collector means less CPU time for mutators (on the CPU in which the collector is ran). Results of SPECjvm98, where the collector does not have a dedicated processor, are provided in subsection 7.3, where SPECjvm98 benchmarks are run on a uniprocessor. Those results support our claim, as it shows that our collector performs much better than the original.

7.2 Collector characteristics

7.2.1 Dead objects measurements. The age-oriented collector optimistically assumes that most young objects are dead, thus hoping to save rc adjustments of their descendants. Table II measures this assumption. The second column presents the fraction of young objects that are actually dead when a collection is triggered. For those object there is no need to increment the rc of their sliding-view values, and
Benchmarks | collector work ratio
--- | ---
**jess** | 1:18.67
**db** | 1:5.47
**javac** | 1:2.41
**jack** | 1:13.84
**mtrt** | 1:8.26

Table I. collector work ratio: work time ratio between the age-oriented collector and the original collector.

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>percent dead young from all young</th>
<th>percent dead young from all dead</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>jess</strong></td>
<td>99.24</td>
<td>99.82</td>
</tr>
<tr>
<td><strong>db</strong></td>
<td>90.7</td>
<td>99.9</td>
</tr>
<tr>
<td><strong>javac</strong></td>
<td>77.03 (99.42)</td>
<td>88.72 (99.99)</td>
</tr>
<tr>
<td><strong>jack</strong></td>
<td>98.31</td>
<td>99.98</td>
</tr>
<tr>
<td><strong>mtrt</strong></td>
<td>95.19</td>
<td>99.91</td>
</tr>
<tr>
<td><strong>jbb-1</strong></td>
<td>99.49</td>
<td>99.97</td>
</tr>
<tr>
<td><strong>jbb-2</strong></td>
<td>98.88 (99.69)</td>
<td>99.33 (99.87)</td>
</tr>
<tr>
<td><strong>jbb-3</strong></td>
<td>97.97 (99.31)</td>
<td>98.71 (99.73)</td>
</tr>
<tr>
<td><strong>jbb-4</strong></td>
<td>95.79 (99.72)</td>
<td>97.24 (99.63)</td>
</tr>
<tr>
<td><strong>jbb-5</strong></td>
<td>91.71 (99.20)</td>
<td>95.50 (99.54)</td>
</tr>
<tr>
<td><strong>jbb-6</strong></td>
<td>90.72 (98.89)</td>
<td>94.76 (99.45)</td>
</tr>
<tr>
<td><strong>jbb-7</strong></td>
<td>86.81 (98.91)</td>
<td>93.18 (99.36)</td>
</tr>
<tr>
<td><strong>jbb-8</strong></td>
<td>84.73 (98.75)</td>
<td>92.76 (99.27)</td>
</tr>
</tbody>
</table>

Table II. dead young objects: fraction of dead young objects out of all young objects and out of all dead objects. Result in parenthesis measured with larger heaps.

their release is extremely rapid since as a consequence there is no need to decrement the \( rc \) of their descendants. The third column presents the fraction of objects which are released using this rapid mode (i.e., it measures fraction of young objects released out off all the released objects).

The measurements were taken while the SPECjvm98 benchmarks were run with a 64MB heap size (except for 213 javac which was run also with a 128MB heap size; the results of that run appear in parenthesis) where the collector ran on a separate spare processor. SPECjbb2000 benchmark was run with two different heap sizes: a 256MB heap size and a 704MB heap size (the results of the 704MB runs appear in parenthesis). In SPECjbb2000 runs, as before, with up to 3 warehouses the collector ran on a separate collector, while with 4 and up warehouses the collector shared a processor with the program threads.

SPECjvm98 measurements show that except for 213 javac, more than 90% of the young object are found dead by our collector, and more than 99.8% of the dead objects are released in the last way. For 213 javac with 64MB, the measurement are less promising since it produces heavy garbage cycles which our reference-counting collector does not collect (and thus one invocation of a tracing collector is needed in the middle of 213 javac’s run), thus producing many small collections, in which the fraction of snooped young objects is high (comparing to the number of young objects created since the last collection) and also many young objects are still alive,
Table III. fraction of reference-counting operations made by the age-oriented collector comparing to the original collector.

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>percent work compared to original</th>
</tr>
</thead>
<tbody>
<tr>
<td>jess</td>
<td>1.55</td>
</tr>
<tr>
<td>db</td>
<td>5.64</td>
</tr>
<tr>
<td>javac</td>
<td>51.86 (1.78)</td>
</tr>
<tr>
<td>jack</td>
<td>5.12</td>
</tr>
<tr>
<td>mtrt</td>
<td>31.65</td>
</tr>
<tr>
<td>jbb-1</td>
<td>1.95</td>
</tr>
<tr>
<td>jbb-2</td>
<td>2.95 (0.89)</td>
</tr>
<tr>
<td>jbb-3</td>
<td>6.24 (1.61)</td>
</tr>
<tr>
<td>jbb-4</td>
<td>14.90 (2.29)</td>
</tr>
<tr>
<td>jbb-5</td>
<td>26.91 (3.62)</td>
</tr>
<tr>
<td>jbb-6</td>
<td>30.60 (3.27)</td>
</tr>
<tr>
<td>jbb-7</td>
<td>39.88 (2.94)</td>
</tr>
<tr>
<td>jbb-8</td>
<td>45.21 (4.08)</td>
</tr>
</tbody>
</table>

since they did not have enough time to die. When we run javac with 128MB heap, those garbage cycles do not cause excessive collections, and therefore the percentages measured are much better.

SPECjbb2000 measurements with 256MB support the above claim: as the number of warehouses rises, collections become more frequent, causing both fractions measured to descend (although even with 8 warehouses, measures show that 84.83% of the young objects created since the last collection are dead, and 92.76% are still released using the rapid path). The measurements with 704MB improve those fractions, as expected.

All in all, the percentage of dying objects in the young generation is very high, as expected.

7.2.2 number of RC updates. Table III shows the fraction of rc updates made by the age-oriented collector comparing to the original reference counting collector. The second column presents the fraction of rc updates executed by the age-oriented collector compared to the number of rc updates executed by the Levanoni Petrank collector. Note that the Levanoni-Petrank collector is already saving a lot of the reference count updates using the dirty bit in the write barrier. The saving over a traditional reference counting algorithm is (much) higher.

As before, the measurements were taken while the SPECjvm98 benchmarks were run with a 64MB heap size (except for javac which was run also with a 128MB heap size; the results of that run appear in parenthesis) where the collector ran on a separate spare processor. SPECjbb2000 benchmark was run with two different heap sizes: a 256MB heap size and a 704MB heap size (the results of the 704MB runs appear in parenthesis). In SPECjbb2000 runs, as before, with up to 3 warehouses the collector ran on a separate collector, while with 4 and up warehouses the collector shared a processor with the program threads.

7.2.3 Profiling measurements. We have profiled the different phases of both the original Levanoni-Petrank algorithm and the age-oriented algorithm. The mea-
Age-Oriented Garbage Collection

Table IV. profiling of the Levanoni-Petrank collector

<table>
<thead>
<tr>
<th>Collector phases</th>
<th>jess</th>
<th>db</th>
<th>javac</th>
<th>mtrt</th>
<th>jack</th>
<th>jbb</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clear-Dirty-Marks and</td>
<td>3.06</td>
<td>4.17</td>
<td>5.08</td>
<td>6.46</td>
<td>4.15</td>
<td>4.29</td>
</tr>
<tr>
<td>Reinforce-Clearing-Conflict-Set</td>
<td>0.85</td>
<td>0.88</td>
<td></td>
<td>1.61</td>
<td>3.76</td>
<td>1.25</td>
</tr>
<tr>
<td>Mark-Roots</td>
<td>42.34</td>
<td>46.03</td>
<td></td>
<td>40.2</td>
<td>41.57</td>
<td>40.46</td>
</tr>
<tr>
<td>Update-Reference-Counters-Young</td>
<td>0.37</td>
<td>0.85</td>
<td>8.15</td>
<td>0.65</td>
<td>0.41</td>
<td>1.5</td>
</tr>
<tr>
<td>Reclaim-Garbage-Young</td>
<td>51.95</td>
<td>47.13</td>
<td>68.43</td>
<td>47.99</td>
<td>48.7</td>
<td>49.56</td>
</tr>
<tr>
<td>Reclaim-Garbage-Old</td>
<td>0.87</td>
<td>0.38</td>
<td>3.5</td>
<td>0.95</td>
<td>3.35</td>
<td>2.22</td>
</tr>
<tr>
<td>Prepare-Next-Collection</td>
<td>0.46</td>
<td>0.46</td>
<td>1.73</td>
<td>0.73</td>
<td>2.24</td>
<td>0.68</td>
</tr>
</tbody>
</table>

Table V. profiling of the age-oriented collector

<table>
<thead>
<tr>
<th>Collector phases</th>
<th>jess</th>
<th>db</th>
<th>javac</th>
<th>mtrt</th>
<th>jack</th>
<th>jbb</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clear-Dirty-Marks and</td>
<td>53.01</td>
<td>27.72</td>
<td>11.53</td>
<td>37.87</td>
<td>57.57</td>
<td>41.25</td>
</tr>
<tr>
<td>Reinforce-Clearing-Conflict-Set</td>
<td>15.05</td>
<td>7.0</td>
<td>7.14</td>
<td>5.7</td>
<td>9.79</td>
<td>11.38</td>
</tr>
<tr>
<td>Mark-Roots</td>
<td>13.75</td>
<td>51.26</td>
<td>42.91</td>
<td>47.84</td>
<td>17.56</td>
<td>21.94</td>
</tr>
<tr>
<td>Update-Reference-Counters-Young</td>
<td>6.58</td>
<td>4.32</td>
<td>19.07</td>
<td>4.23</td>
<td>5.24</td>
<td>7.51</td>
</tr>
<tr>
<td>Reclaim-Garbage-Young</td>
<td>1.29</td>
<td>5.93</td>
<td>1.26</td>
<td>0.51</td>
<td>1.41</td>
<td>0.43</td>
</tr>
<tr>
<td>Reclaim-Garbage-Old</td>
<td>2.33</td>
<td>0.34</td>
<td>13.76</td>
<td>0.53</td>
<td>1.49</td>
<td>10.29</td>
</tr>
<tr>
<td>Prepare-Next-Collection</td>
<td>7.92</td>
<td>3.36</td>
<td>4.43</td>
<td>3.27</td>
<td>6.89</td>
<td>6.96</td>
</tr>
</tbody>
</table>

measurements were taken while the SPECjvm98 benchmarks were run with a 64MB heap size and the SPECjbb2000 benchmark (with 1,2,3 warehouses) was run with a 256MB heap size.

Table IV presents the profiling measurements of the original Levanoni-Petrank algorithm. Those measurements indicate that:

— a very large portion of the collector work (usually around 90%) is dedicated to the handling of the *YoungObjects* buffer, which includes incrementing the *rc* of the sliding-view values of this buffer’s objects and the release of the objects included in this buffer which turned out to be garbage (i.e., had a zero *rc*) after all reference-counting adjustments were made.

—the handling of *Updates* buffer (*rc* adjustments of objects logged in the *Updates* buffer, and the release of such objects) usually consumes negligible resources.

The clear throughput superiority of the age-oriented collector, is better understood after seeing those profiling measurements: our age-oriented collector has focused on reducing the overhead of the 2 major collector phases (adjusting *rc* of young objects’ sliding-view slots and releasing young objects). Table V summarizes the profiling measurements of the age-oriented algorithm. Since the two major phases (which comprise 78%-95% of the original collector work) were dramatically improved, the other phases which comprise 5%-22% of the original collector work, now comprise of 42%-85% of the original collector work. That actually shows how well the age-oriented collector does.

### 7.3 Client performance

Although the age-oriented collector is targeted at servers running on SMP platforms, we also measured its performance against the original algorithm on a unipro-
Fig. 25. SPECjvm98 on a uniprocessor: run-time ratio compared to the Levanoni-Petrank collector.

The behavior of the collector on a uniprocessor may demonstrate its efficiency. We measured the age-oriented collector on a uniprocessor with the SPECjvm98 benchmark suite and the results appear in figure 25. Our measurements show that the age-oriented algorithm is substantially better than the original one in all tests as throughput ratios are shown to improve from 10% to 50%. Note that on uniprocessor, the age-oriented collector achieves a better throughput ratio than on multiprocessor (figure 23), because runs on uniprocessor measure also the collections time, where the age-oriented collector is substantially more efficient than the original one.

8. RELATED WORK

Generational garbage collection was introduced by Lieberman and Hewitt [Lieberman and Hewitt 1983], and the first published implementation was by Ungar [Ungar 1984]. Both algorithms were aimed to reduce the running time of most collections by focusing on the young objects.

Appel [Appel 1989] presented a two-generation collector with variable young generation size: all its free space is devoted to the young generation. When the young generation becomes full (and thus both generations consume all usable memory), 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.

Demers, et al. [Demers et al. 1990] presented an algorithm for using generational collector without moving the objects. Their motivation was to adapt generations for conservative garbage collection. Instead of partitioning the heap physically and keeping the young objects in a separate place, it is suggested to partition the heap logically, and to keep a bit per object, indicating whether the object is young or old.

Other interesting variants of generational collection include the Train algorithm [Hudson and Moss 1992], the older first collector [Stefanović et al. 1999], and Beltways [Blackburn et al. 2002].

The age-oriented collectors focus on on-the-fly collectors, and thus this work may be thought of as a continuation and improvement over previous incorporations
of generational collectors into on-the-fly collectors. Such previous reports are by Domani, Kolodner, and Petrank [Domani et al. 2000], and by Azatchi and Petrank [Azatchi and Petrank 2003]. Domani, Kolodner and Petrank adopt the idea of Demers, et al. [Demers et al. 1990] to partition the young and the old objects logically (as objects cannot be moved). The Doligez-Leroy-Gonthier Mark and sweep collector [Doligez and Leroy 1993; Doligez and Gonthier 1994] is used both for the collection of the young generation and the collection of the full heap.

Azatchi and Petrank [Azatchi and Petrank 2003] have proposed using reference counting for the old generation and mark and sweep for the young generation. They built on the reference counting collectors of Levanoni and Petrank [Levanoni and Petrank 2001] and on the mark and sweep collector of Azatchi et al. [Azatchi et al. 2003]. Their measurements showed good performance for this collector combination, which is similar to what we use for the age-oriented collector. Note that they use a generational collector and not an age-oriented one. Our collector turns out to do better. Since we never collect the young generation separately, our mechanism for finding inter-generational pointers was much simpler and the age oriented collector could employ the original write barrier of Levanoni and Petrank.

Blackburn and McKinley [Blackburn and McKinley 2003] implemented a uniprocessor stop-the-world generational collector with reference counting for the old generation and copying for the young. Using copying for the young generation is a natural choice for stop-the-world collection, as it makes better use of the high death rate. Adopting copying for an on-the-fly collector is rather difficult (the only known such construction appears in [Hudson and Moss 2001]). The goal in [Blackburn and McKinley 2003] was to use the generational collector to shorten the pauses a stop-the-world reference counting may incur, while obtaining good throughput. They used a clever mechanism to 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. Indeed, their collector demonstrated controlled pause times with good throughput. These pause times are larger than those obtained by on-the-fly collectors.

9. CONCLUSIONS

We have proposed the age-oriented framework of garbage collectors making use of the generational hypothesis without running frequent time-consuming young generation collections. An age-oriented collector runs only full heap collections (when the heap is exhausted) but it treats objects according to their age. An age-oriented collector does not maintain the short pauses of a generational collector but it allows higher throughput. Such a collector is useful when throughput is important and pauses are not. A particular such occasion happens when using on-the-fly collectors; these collectors achieve extremely short pauses even without employing generations.

A partial incorporation of generations with an on-the-fly collector, used only for immutable objects was used by Doligez, Leroy and Gonthier [Doligez and Leroy 1993; Doligez and Gonthier 1994]. 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 then propose an instantiation of an age-oriented collector where the reference counting algorithm is used to collect the old objects and the mark and sweep algorithm is used to collect the young objects. We use the on-the-fly sliding views collectors as the base for the design. This age-oriented collector is then implemented on Jikes and measurements are presented. The obtained collector turns out to maintain the short pauses of the original collectors but improve significantly over their efficiency. It outperforms both the original reference counting collector as well as the generational variant of running frequent tracing young generation collections with infrequent reference counting for the full heap.

10. ACKNOWLEDGEMENT

We thank Elliot (Hillel) Kolodner for helpful discussions.

REFERENCES


