RC24794 (W0905-013) May 7, 2009
Computer Science

IBM Research Report

Efficient Memory Management for Long-Lived Objects

Ronny Morad[1], Martin Hirzel[2], Elliot K. Kolodner[1], Mooly Sagiv[3]

[1]IBM Research Division
Haifa Research Laboratory
Mt. Carmel 31905
Haifa, Israel

[2]IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
USA

[3]Tel-Aviv University
Israel

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

Efficient Memory Management for Long-Lived Objects

Ronny Morad
IBM Haifa Research Center
morad@il.ibm.com

Martin Hirzel
IBM Watson Research Center
hirzel@us.ibm.com

Elliot K. Kolodner
IBM Haifa Research Center
kolodner@il.ibm.com

Mooly Sagiv
Tel-Aviv University
msagiv@post.tau.ac.il

Abstract

Generational garbage collectors perform well for short-lived objects, but do not deal well with long-lived objects. Existing techniques for long-lived objects, such as pretenuring, eliminate work in the nursery; however, the collector still needs to deal with the long-lived objects in the older generations. We introduce a novel scheme employing regions that avoids both nursery and old generation costs.

Our scheme divides the heap into two sub-heaps: a garbage collected heap and a region heap. Most objects are allocated in the garbage collected heap, while long-lived objects are allocated in regions. The scheme maintains reference counts to the regions, and reclaims regions when their count drops to zero. The memory management mechanism is decoupled from the region selection policy. Region selection policies may range from manual to fully automatic. This paper presents a particular realization for our memory management scheme over an Appel-style generational garbage collector. This realization includes an automatic profile-based region selection technique. Evaluating our algorithm using a garbage collection simulator, it reduces copying by 10% on average when compared to Appel in tight heaps.

Categories and Subject Descriptors D3.4 [*Programming Languages*]: Processors—Memory management (garbage collection); Optimization

General Terms Performance, Measurement, Experimentation

Keywords Garbage Collection, Regions, Generations

1. Introduction

One of the key features of modern programming languages, such as Java, C#, and Python, is automatic memory management. Using automatic memory management the programmer only needs to allocate objects, and their reclamation when no longer necessary is performed by a garbage collector. This simplifies the design and code of programs and eliminates many memory related bugs. Garbage collectors interrupt the program execution occasionally in order to reclaim space.

Generational copying garbage collectors achieve good performance by dividing the heap into several spaces (usually two) termed generations [31, 44]. New objects are allocated in the nursery; when there is no more space in it, a minor collection scans the objects in the nursery and may promote surviving objects to the older generation. When there is no more space in the older generation, a major collection scans all reachable objects. Generational copying garbage collectors perform well, relying on the assumption that most objects die young and do not even survive a minor collection. However, long-lived objects may survive multiple major collections and be copied or traced each time.

Pretenuring is a technique that reduces copying in the nursery by directly allocating objects that are predicted to be long-lived in the old generation [9, 21, 4]. This eliminates the first copy. However, subsequent copies are still performed. Another technique is to allocate objects that are predicted to live throughout an entire program execution in a permanent space [9, 21, 4]. However, this does not address the issue of objects that are long-lived but not permanent. Such objects are likely to appear in a Web server: an object may be allocated at the beginning of a web transaction and live throughout the transaction.

We introduce a novel scheme employing regions that avoids both nursery and old generation costs. The scheme combines a generational garbage collector to handle the short-lived objects with regions to improve the handling of long-lived objects and achieve better overall performance. The scheme can work with a variety of generational and old space collectors, not necessarily copying collectors.

In our scheme the heap is divided into two sub-heaps: a garbage collected heap and a region heap. Most objects are allocated in the garbage collected heap while some objects are allocated in regions. Reference counts are maintained to regions in order to know when a region can be reclaimed. When a region is reclaimed then all the space it occupies is reclaimed at once. There is no partial region reclamation. This paper introduces two variations of the memory manager algorithm: one in which the reference counts are accurate after both major and minor collections, and a second in which the reference counts are accurate only after major collections. The former incurs some extra write barrier overhead, whereas the latter incurs no extra write barrier overhead.

Our method can employ a variety of region selection methods. One method is to let the programmer decide explicitly. Another possibility is to employ static analysis to determine the regions. Regions can also be assigned dynamically during program execution. For this work we developed a novel profile-based technique for deciding which objects to allocate in regions.

This paper introduces a realization of the memory management scheme based on an Appel-style generational garbage collector [2]. Blackburn et al. found that Appel's collector yields excellent throughput [6]. In the realization a region selector chooses which objects to allocate directly in regions based on their nested allocation site (also known as call-chain). The selector employs profiling in order to gather information on the nested allocation sites, and according to the information gathered it decides which sites are suitable for regions. The selector uses traces with accurate object death times generated by the Merlin algorithm [24].

Employing gcSim [26], a garbage collection simulator, we check the potential performance of the realization of our scheme. We use a medium size input for training (profiling and region selection) and a large size input for testing. The simulation results show that in a tight heap configuration our scheme gains a significant average improvement over the Appel collector: 10% less copying, 4.2% less scanning, 7.2% less major collections, and 6.3% less minor collections. The responsiveness of our method not only did not degrade, but even improved slightly. Using the same input for

training and testing, the results are even more significant: 22.3% less copying, 7.6% less scanning, 18% less major collections, and 10.9% less minor collections.

In summary this paper makes two primary contributions:

1. A novel memory management scheme, based on combining generational collection with regions, that deals with long-lived objects.
2. An effective technique for automatic region selection.

The remainder of this paper is organized as follows. Section 2 gives background information on generational garbage collection, region based memory management, and reference counting. Section 3 describes our memory management scheme, and Section 4 presents a realization of our scheme, including our region selection technique. Section 5 explains the experimental methodology, and Section 6 gives the results. Section 7 compares our work to related work, and Section 8 concludes.

2. Background

This paper is concerned with efficient automatic memory management. Memory management is the runtime system component that allows a program to allocate heap objects, for example, with "cons" in Lisp, "malloc" in C, or "new" in Java [45]. Automatic memory management means that the programmer does not need to explicitly "free" objects. This section briefly reviews some terminology for later use; for in-depth surveys, see [29, 42].

Copying garbage collection implements automatic memory management by copying survivors and discarding dead (i.e., unreachable) objects. It divides heap memory into two semispaces. Only one semispace is active for allocation at a time, the other semispace serves as *copy reserve*. Garbage collection starts when the active semispace is full. It first identifies the *root set*, the set of program variables that hold pointers to heap objects. The collector copies all objects that are transitively reachable from the root set to the copy reserve, and discards the originals. When the program resumes, it uses the other semispace for allocation.

Most language runtime systems today use *generational* garbage collectors, because they tend to yield the best throughput. Generational collectors segregate objects by age into generations [31, 44]. Younger generations are collected more often than older generations, which reduces overall collector work, because most objects become unreachable while they are still young. For concreteness, this paper discusses generational collectors with just two generations, and refers to them as *young space* and *old space*.

Generational collectors maintain a *remembered set* of locations (objects, slots, or small chunks of memory called cards) in the old space that have pointers to objects in the young space. This enables collecting the young space separately by traversing pointers starting from the remembered set in addition to the root set, and ignoring pointers to the old space. A *write barrier* adds old locations to the remembered set when the program stores pointers to young objects in them.

Section 3 presents a memory management scheme that can make use of any garbage collector. As a proof of concept, Section 4 describes a realization based on Appel's generational garbage collector [2]. *Appel's collector* uses a flexible size young generation: instead of a fixed space allowance, the young generation can always use all available space that is not needed for old objects or as copy reserve. This leads to less frequent collections and thus better program performance.

In *region based memory management*, the program allocates heap objects into regions individually, and reclaims all objects in a region simultaneously when none of them are needed anymore [42]. The advantage of this *all-or-nothing* policy is that there is no overhead for allowing individual object reclamation. The disadvantage is that there can be a long time lag between the time that an individual object becomes unreachable and the time when its memory is freed along with the surrounding region.

Reference counting keeps track of the number of references to each object during execution: every pointer write increases the count for the new target, and decreases the count for the old (overwritten) target. Reference counting garbage collectors reclaim objects when their reference counts drop to zero [14], and thus unlike other collectors, require no reachability traversal or copying. In its most basic form, reference counting suffers from two problems: it cannot reclaim cyclic garbage (since the objects in the cycle keep each other's reference counts positive), and it incurs a high overhead (due to bookkeeping at each pointer store). The bookkeeping overhead can be reduced by *deferred reference counting* ([15], no bookkeeping for stack variables, which are modified most frequently) and *ulterior reference counting* ([7], no bookkeeping for young objects, which are modified more frequently than old objects). The current paper counts references for entire regions instead of individual objects, collects cycles, and uses a technique similar to ulterior reference counting to reduce overhead.

3. The Memory Management Scheme

We choose to describe the memory management scheme for a collector with two generations, where objects surviving a minor collection are immediately promoted to the old generation. The scheme will also work for collectors that age objects in the young generation before promoting them.

The scheme divides the heap into two spaces: the garbage collected space and the region space. At allocation time, a region selector determines whether to allocate a new object in the garbage collected heap or in one of the regions. The division is flexible and changes adaptively during execution. In particular, region size and the total space allocated to regions can grow on demand as long as enough space is left for the collector to complete its work. For example, in a copying collector enough space should be reserved to enable all reachable objects to be copied.

The memory manager maintains reference counts to each of the regions. For efficiency, the reference counts do not include references from the young space or from the root set, analogous to ulterior reference counting [7]. The reference counts are updated by the write barrier and during collections. The memory manager also maintains a remembered set to the young space. The set includes references from the old space and from the region space.

Figure 1 depicts a possible state of this scheme. The region heap contains three regions: r0, r1, and r2. The rest of the space is used by the garbaged collected heap. The remembered set holds references to objects that point to the young space. Each region has a reference count which does not include references from the young space or from the root set.

When there is no room left in the young space, a minor collection is triggered. Surviving objects are promoted to the old space, and the reference counts of the regions are adjusted accordingly. When the collection is complete, it is safe to reclaim regions that have a zero reference count and no references from the root set. The space that was reclaimed is returned to the general heap and can be reused for either the garbage collected space or the region space. The next section shows how this can be done efficiently.

Figure 2 depicts a possible heap state right after a minor collection. Region r0 can be safely reclaimed. Region r2 cannot be reclaimed although it has a reference count of zero since a member of the root set refers to it.

There is no partial region reclamation; the reclamation of regions is performed according to an "all-or-nothing" policy. If the

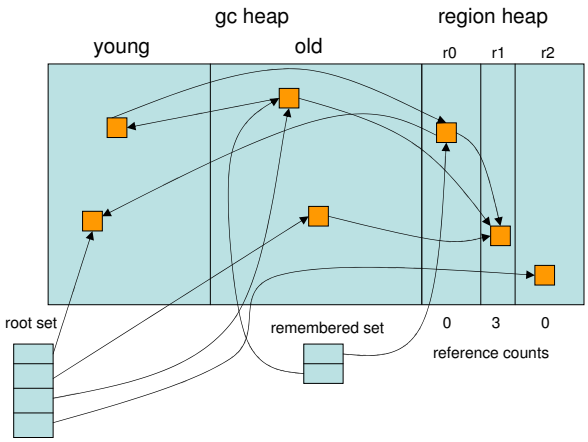

Figure 1. The memory management scheme. The region heap contains three regions: r0, r1 and r2. The rest of the space is used by the garbaged collected heap. The remembered set holds references to objects that point to the young space. Each region has a reference count which does not include references from the young space or from the root set.

reference count of a region is more than zero, then the region is not reclaimed, even though there may be unreachable objects in it.

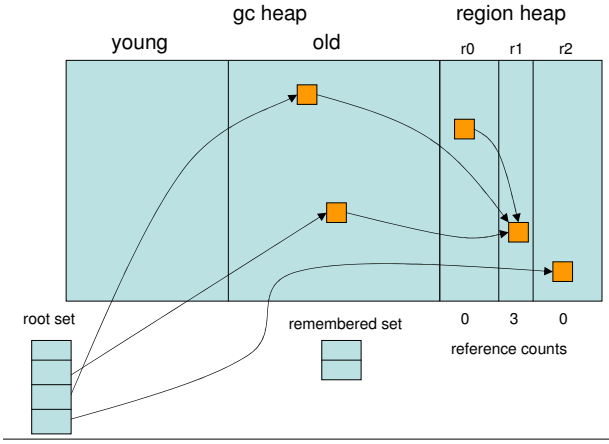

Figure 2. The heap right after a minor collection. Region r0 can be safely reclaimed. Region r2 cannot be reclaimed although it has a reference count of zero since a member of the root set refers to it.

When there is no more room in the old space, a major collection is triggered. During a major collection all the reachable objects are scanned, and the reference counts are recalculated for each of the regions. The recalculation avoids the need to sweep the old space after a major collection; this sweep would be needed to adjust the reference counts from old space objects that became garbage. When a major collection completes, the garbage collected heap contains only reachable objects and there are accurate reference counts for each of the regions. Similar to a minor collection, regions with zero reference counts and no references from the global root set (static variables and the stacks), can be safely reclaimed. Again, the reclaimed space is returned to the general heap and can be reused for either the garbage collected space or the region space. The recalculation of reference counts during a major collection ensures that cyclic garbage regions are also collected.

So far, we have outlined the first variant of the memory management algorithm. In a second variant, reference counts for the regions are calculated only during major collections (i.e., they are not maintained during normal program execution as the heap is mutated). In this variant, regions can only be reclaimed safely after major collections. This eliminates extra overhead on the write barrier; however, it provides less opportunities for region reclamation. We compare the two variants in our performance analysis.

3.1 Object Allocation

```
1:  allocate(objParams):
2:      region = selectRegion(objParams)
3:      if region == null // garbage collected heap
4:          gcHeap.allocate(objParams)
5:      else // region heap
            // try to allocate in the selected region
6:          if !region.allocate(objParams)
                // not successful. Initiate a garbage collection.
7:              gcHeap.collectGarbage(objParmas.size)
                // try to allocate again in the selected region
8:              if !region.allocate(objParams)
                    // not successful. Give up.
9:                  terminate "Out of Memory"
```

Figure 3. Allocation pseudo-code.

Figure 3 shows pseudo-code for the allocation sequence. Its input, *objParams*, denotes parameters known at allocation time, relevant to allocation and region selection. They include the frame pointer, thread id, object size, object type and nested allocation site.

Based on the input, the region selector chooses where to allocate the object (Line 2). The region selection technique is pluggable and is an independent part of the memory management scheme. Section 4 introduces an effective profile-based technique for region selection. If the region selector decides that this object is not part of the region space, then the object is allocated in the garbage collected space (Lines 3+4). If the region selector returns a region, then an attempt is made to allocate that object in the region.

If the allocator fails to allocate the object in that region (Line 6), because the region size cannot grow, then the memory manager forces a garbage collection (Line 7). It also provides the collector the size of the object so the collector can decide between a minor and a major collection.

Following the collection, the allocator makes another attempt to allocate the object in the region (Line 8) in the hope that enough space was reclaimed. If it fails, then the memory manager terminates with an insufficient memory error.

3.2 Write Barrier

```
 1:  pointerUpdated(srcObj, oldTgtObj, tgtObj):
 2:      updateReferenceCounts(srcObj, oldTgtObj, tgtObj)
 3:      updateRemSet(srcObj, oldTgtObj, tgtObj)

 4:  updateReferenceCounts(srcObj, oldTgtObj, tgtObj):
         // references from the young space are not counted
 5:      if !isYoung(srcObj)
             // avoid counting self references to regions
 6:          if !updateToSameRegion(oldTgtObj, tgtObj)
 7:              if oldTgtObj ≠ null and isRegion(oldTgtObj.space())
 8:                  decreaseRC(oldTgtObj.space())
 9:              if tgtObj ≠ null and isRegion(tgtObj.space())
10:                  increaseRC(tgtObj.space())
```

Figure 4. Write barrier pseudo-code.

Figure 4 shows pseudo-code for the write barrier sequence. Line 3 calls the normal write barrier of the generational garbage collector, which maintains the remembered set as described in

Section 2. Variation I of our memory manager requires extra work for adjusting the reference counts of the regions (Lines 4–10). For efficiency, it does not count references from the root set or from the young space, similar to deferred and ulterior reference counting respectively [15, 7]. Notice that in many cases either the old or new target are null and therefore no work is required besides the null check.

In the second variant of the scheme, the write barrier only updates the remembered set and introduces no extra overhead for maintaining region reference counts.

3.3 Minor Garbage Collection

The roots for minor collections are the global root set (static and stack roots), as well as the remembered set, which holds references from the old space and from the region space to the young space.

During a minor collection surviving objects are promoted to the old space. The reference fields of these objects are scanned in order to check whether there are regions that are referenced by these fields, and their counts are updated accordingly.

When the collection of the young space is complete, there are no objects remaining in it and the reference counts for the regions, which count the number of references from the old space and other regions, are accurate. The collector reclaims regions with zero reference counts and no references from the root set. The remembered sets are also discarded. Notice that if a region that is freed points to another region, then the pointed region reference count will not drop to zero at least until the next major collection, where its reference count is recalculated.

3.4 Major Garbage Collection

During a major collection all reachable objects are scanned, and reference counts are recalculated for the regions. The roots for a major collection are the global root set (static and stack roots).

At the beginning of a major collection the reference counts of the regions are reset. During the heap traversal, each time a reference that points into a region is traversed, the reference count of the region is incremented. Self references are not counted.

When the collection of the young and old spaces is complete, the collector reclaims regions with zero reference counts and no references from the root set. Such regions can be safely reclaimed. Notice that cycles between unreachable objects in regions will not be included in the reference counts, and thus, a region that has only unreachable objects will be reclaimed. The remembered sets are also discarded.

The recalculation of the reference counts incurs little extra overhead as all of the reachable objects in the young and old spaces need to be traced in any case. There is an extra cost to scan the reachable objects in the regions, which is less than the cost to copy them.

3.5 Second Variant of the Memory Manager Algorithm

Until now we described the first variant of the memory manager algorithm. The second variant does not maintain reference counts to the regions during heap mutation. Thus, the write barrier does not need to update reference counts and it is the same as for the generational garbage collector. In this case regions cannot be reclaimed after a minor collection. A major collection is the same as for the first variant; the collector recalculates reference counts and frees regions whose count is zero at the end of the heap traversal.

The main advantage of the second variant is that it incurs no additional overhead on the write barrier of the generational garbage collector. However, there are less opportunities for reclaiming regions since they can be collected only at major collections. Section 6 evaluates the performance of both variations.

3.6 Correctness

Safety: Objects in regions are not reclaimed prematurely. A region is reclaimed after a minor collection if its reference count is zero and no roots point to the region. If the reference count is zero, there are no pointers from the old generation and from other regions. The only other origin of pointers to the region would be the young generation, but that is empty right after a minor collection.

Liveness: If none of the objects in a region is reachable from the roots, it will be reclaimed no later than at the next major garbage collection. It will be reclaimed earlier if its reference count drops to zero, but that may not happen, because there may be pointers from dead objects in the old generation or in other regions that keep the reference count above zero even if the entire region is dead. But during a major garbage collection, all reference counts are recomputed based on a traversal of live (reachable) objects only. Therefore, if no object in the region is reachable, its reference count is zero and it gets reclaimed.

4. Realization of the Memory Management Scheme

This section describes a realization of the memory manager scheme based on an Appel-style generational collector [2]. Prior work has shown it is one of the best performing collectors [6]. First we describe how we organize the heap. Then we introduce our region selection technique and the heuristic on which it is based.

4.1 Heap Organization

We divide the heap into 4KB pages. The pool of pages is shared between the garbage collected heap and the region heap. The sharing of the same pool of pages for both the garbage collected heap and the region heap enables the efficient utilization of the heap, which we describe below.

The garbage collected heap has a current allocation page. The allocator for the garbage collected heap first tries to allocate on the current page; if there is no room, then it asks for additional pages from the pool to accommodate the object. Multiple pages may be required for large objects, which are allocated on contiguous pages. The last page requested becomes the current allocation page.

In order to be able to perform garbage collection successfully a copy reserve is required. This means that for every page used in the garbage collected heap there must be a corresponding free page in the pool to allow copying. If allocating an object would result in reducing the copy reserve below the level needed to complete a garbage collection successfully, then a collection is initiated.

A similar procedure performs allocation on the region heap. Each region has its own current allocation page. The regions grow as long as the copy reserve allows for successful garbage collection. If a region cannot grow because it would reduce the copy reserve (of the garbage collected heap) below its limit, a garbage collection is initiated. The regions do not need a copy reserve, since no copying is performed in the region space. When the reference count of a region drops to zero, the whole region is reclaimed, and its pages are returned to the pool, effectively increasing the size of the garbage collected heap.

4.2 The Region Selection Technique

Our region selection technique associates a region with a single nested allocation site. A nested allocation site is the call-chain that allocates an object. It is a sequence of method:offset, starting from the deepest method (allocating method) up to the main method. To avoid overfitting, we used a nested allocation site of size three: method1:offset1, method2:offset2, method3:offset3, where method1 is the allocating method. When a nested allocation site

is associated with a region, then all the objects that are allocated at this nested allocation site are allocated to that region.

There are two stages to the technique. First an offline stage profiles the application. The profiling collects relevant statistics, which we detail below, regarding the nested allocation sites. Based on these statistics, a heuristic selects the nested allocation sites to be associated with regions. The second stage is the production run, where the chosen sites allocate in regions.

4.2.1 Floating Garbage Ratio

The heuristic is based on a measure of floating garbage in the region associated with a nested allocation site, which we call the floating garbage ratio and explain below.

When an object is allocated in a region it consumes space equal to its size. As time advances, more objects are allocated to the region, and the space that the region consumes is equal to the sum of the sizes of the objects that were allocated in that region. When an object becomes unreachable, the object continues consuming space until its region is freed. Thus, at any point in time, the memory consumption of a region is equal to the amount of space occupied by reachable objects plus the amount of space occupied by unreachable objects. At the time the region is freed, its memory consumption drops to zero.

The memory consumption of a region can be drawn as a graph where the y-axis is the memory consumption in bytes, and the x-axis is the program execution time measured in bytes allocated since the start of the program. There are three relevant graphs: one for the memory consumption of the reachable objects, the second for the memory consumption of the unreachable objects, and the third for the sum of the consumption of the reachable and unreachable objects (the total consumption of the region).

We define *RB* as the space-time product of reachable bytes in a region; it is the area beneath the graph of reachable objects. We define *AB* as the space-time product of the allocated bytes (sum of the reachable and unreachable) in a region; it is the area beneath the graph of the allocated objects. The floating garbage ratio of a region, called *FGR*, is computed as $(AB - RB)/AB$.

Figure 5 shows possible memory consumption graphs for a prospective region and illustrates the concepts. The *RB* for the region is 6.5 kb^2. The *AB* for the region is 10.5 kb^2. The *FGR* is 38%.

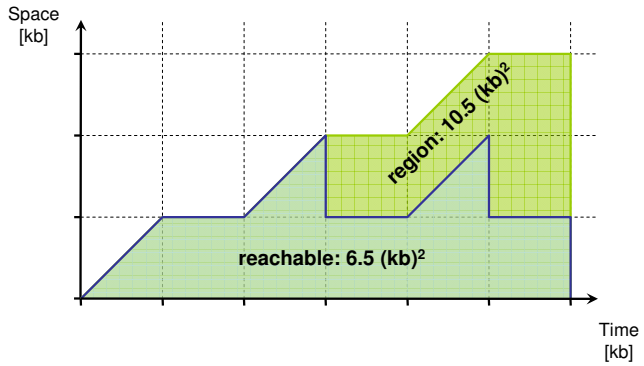

Figure 5. Floating garbage. The floating garbage ratio is 38%.

The *RB* and the *AB* for all nested allocation sites are calculated in linear time in a single pass over a Merlin trace [24]. A Merlin trace is a trace that contains the exact times when all objects first become unreachable, which is the earliest time at which they can be deallocated.

4.2.2 The Heuristic

Our goal is to select nested allocation sites that will perform better when they are managed in regions rather than by the garbage collector. We observe that regions perform best when their floating garbage ratio is zero. A floating garbage ratio of zero means that all the objects in a region die together, and the region is freed as soon as its objects die. We also observe that we should avoid allocating short-lived objects in regions as these are the objects that a generational collector handles best.

The above arguments lead to the following criteria for choosing a nested allocation site for allocation in a region:

1. The floating garbage ratio for the region associated with the site should be as small as possible, in particular smaller than *fgr_threshold*.
2. The objects allocated at the site should be long-lived, in particular, the average lifetime of the objects allocated is greater than *lifetime_threshold*. Some nested allocation sites may allocate objects with very different lifetimes. Thus, we only select sites with a low standard deviation in the object lifetime, in particular less than *stddev_threshold*.
3. The site should allocate as much space as possible. Choosing all the nested allocation sites that meet the first two criteria might require too many regions. Since the heap is divided into pages, it might become too fragmented. Thus, among all regions that meet the first two criteria, we select those that allocate the largest amount of space. We select no more than *region_num* regions.

The particular values of the constants that we used in our experiments are:

1. *fgr_threshold* $= 0.01$. This value avoids missing nested allocation sites that have a negligible yet positive floating garbage ratio.
2. *lifetime_threshold* $= 0.3 * (high\_watermark)$, where the *high_watermark* is the largest reachable heap during the execution. The value $0.3 * (high\_watermark)$ has the same order of magnitude as a semispace of a young generation, and hence, is a reasonable distinguisher for old objects.
3. *stddev_threshold* $= 0.3 * (average\_lifetime)$, where *average_lifetime* is the average lifetime of the objects allocated at a site. Picking a low value compared to the average lifetime leads to homogeneous intra-region longevity.
4. *region_num* $= 10$. Usually, most benefit comes from a handful of nested allocation sites. Each region corresponds to one nested allocation site, and 10 regions are enough to get the benefit yet few enough that fragmentation is not a concern.

5. Methodology

The methodology of this paper consists of the following steps: generate a training trace for each benchmark using a small input; select regions based on the training trace; generate a measurement trace using a larger input; and simulate the garbage collectors on the measurement trace to collect performance metrics.

The traces come from a trace generator implemented in version 2.2.0 of Jikes RVM, an open-source Java virtual machine [1]. Traces are chronological recordings of object allocations, pointer updates, and object deaths. Allocations and pointer updates enable garbage collection simulation, and object deaths enable simulator validation and region selection. The death time of an object is the exact time when the object becomes unreachable as computed by the Merlin algorithm [24]. The trace generator was originally developed to evaluate connectivity-based garbage collection [26], and

Program	Suite	Input (training)	Input (measured)	Total allocation objects	Total allocation MB	High water	Turn-over
jess	jvm98	`-s10`	`-s100`	11,750,329	359.6	1.5	232.3
ipsixql	Colorado	`1 2`	`3 2`	8,816,975	261.0	3.2	81.3
jack	jvm98	`-s10`	`-s100`	13,528,967	395.5	5.0	79.6
bh	Olden	`-b 50 -s 10`	`-b 500 -s 10`	1,106,090	33.8	0.5	70.9
javac	jvm98	`-s10`	`-s100`	18,518,049	530.1	9.0	58.7
pseudojbb	jbb	1 warehouse, 7,000 trans.	1 warehouse, 70,000 trans.	18,606,121	502.8	26.7	18.8
mtrt	jvm98	`-s10`	`-s100`	6,868,468	163.4	8.7	18.8
compress	jvm98	`-s10`	`-s100`	9,608	105.3	6.7	15.7
health	Olden	`-l 5 -t 50 -s 1`	`-l 5 -t 500 -s 1`	1,778,253	37.7	2.6	14.7
xalan	Colorado	`1 2`	`3 2`	7,054,505	390.4	32.7	12.0
db	jvm98	`-s10`	`-s100`	3,763,760	90.8	8.6	10.5
mpegaudio	jvm98	`-s10`	`-s100`	25,102	1.0	0.5	2.3
null	(none)	–	–	1,488	0.1	0.0	2.9

Table 1. Traces used in this evaluation.

we augment it to record nested allocation sites instead of just flat allocation sites.

We implement our memory management scheme inside of the gcSim garbage collection simulator.[1] It consists of implementations of various collectors, supported by models of the root set, the heap, and individual objects and a block manager (the heap is organized as a number of fixed-sized blocks).

Jikes RVM is written mostly in Java, and runtime system components, such as the optimizing JIT compiler, allocate Java objects in addition to the objects allocated by the benchmark itself. The traces come from runs of Jikes RVM with adaptive optimization system enabled, and start after one benchmark execution has completed, so the compiler causes little additional allocation. The tracer denotes all objects reachable at the beginning of the second run as *boot image* objects. The simulator places boot image objects in a special region that is not part of the garbage collected heap. Boot image objects are part of the remembered set for minor garbage collections. Major garbage collections traverse but do not copy boot image objects. Region reference counts include pointers from boot image objects.

Table 1 describes the traces from our benchmark programs, which come from a variety of benchmark suites (SPECjvm98[2], SPECjbb2000[3], Colorado[4], and Olden[5]). The "null" benchmark consists of the empty main method; it is not interesting in and of itself, but it puts results for other benchmarks into perspective. Columns "Input (training/measured)" show the command line arguments for the two traces that we collect for each benchmark. Two benchmarks (null and mpegaudio) allocate very little memory, so the results section excludes these benchmarks.

Columns "Total allocation", "High watermark", and "Turnover" characterize the measurement traces excluding the boot image. Total allocation is the sum of all allocations in the trace, high watermark is the maximum amount of reachable data at any time during the trace, and turnover is the ratio of total allocation over high watermark. The garbage collection simulator bounds the heap size to a multiple of max live (for example, 3×), and therefore, programs with higher turnover exert more garbage collection pressure. The rows are sorted by decreasing turnover, putting the most interesting benchmarks at the top.

[1] `http://www-plan.cs.colorado.edu/hirzel/gcSim/`

[2] `http://www.specbench.org/osg/jvm98`

[3] `http://www.specbench.org/osg/jbb2000`

[4] `http://www-plan.cs.colorado.edu/henkel/projects/colorado_bench/`

[5] `http://www-ali.cs.umass.edu/~cahoon/olden`

Using a simulator to evaluate our memory management scheme has advantages over a full implementation: it allows us to abstract from implementation details, it allows us to compare our scheme to other collectors in a controlled environment, and it allows us to experiment with different variations on our scheme and to evaluate their performance before committing to a full-fledged implementation. There are also drawbacks to not using a full implementation in a Java virtual machine. The most important is that simulation can not give us concrete timing numbers. Another drawback is that we have no cache-level locality numbers. However, previous work has shown that simulators can be useful for GC research [37, 16, 26].

6. Results

Using the simulator we analyze the performance of the realization of our memory management scheme. We compare it to an Appel-style collector both on overall garbage collection performance and responsiveness, analyze its sensitivity to heap size, and check its sensitivity to the input used on the training runs.

6.1 Comparison to Appel

We compare the two variants of our algorithm to an Appel-style collector. Variation I reclaims regions both after major and minor collections, and Variation II reclaims regions only after major collections. The metrics for comparison are the number of bytes copied, the number of bytes scanned, and the number of major and minor collections. These metrics have a strong correlation to collector performance; copying and scanning are the primary costs for a copying collector and there is also a non-negligible fixed cost for each major and minor collection. Copying is more expensive than scanning, and therefore the bytes copied result should have a bigger weight when evaluating performance.

We compare using a tight heap size: 2.3 times the high watermark of reachable memory for each benchmark. A tight heap size is most interesting because the time that a program spends in garbage collection is bigger, and any improvement obtained is more significant.

Table 2 shows the results for each benchmark: the number of megabytes copied and scanned with Appel, and the percentage improvement over Appel using our memory manager. The last line of the table shows the average percentage improvements.

Table 3 shows the number of major and minor collections, and the percentage improvement for each. The last line of the table also shows the average percentage improvements.

These results show that on average our realization performs better than Appel on all four metrics. Surprisingly, Variation II, which has less opportunity to free regions (only on major collections) performs slightly better than Variation I on average. In particular Vari-

Program	Appel		Variation I		Variation II	
	copied	scanned	copied	scanned	copied	scanned
jess	52.3	127.0	36.9%	11.2%	39.0%	21.7%
ipsixql	284.8	97.4	7.8%	3.9%	7.8%	4.9%
jack	73.6	43.6	13.3%	10.2%	13.9%	9.2%
bh	11.7	22.0	17.0%	4.9%	15.0%	-4.1%
javac	370.6	145.6	7.1%	1.3%	7.1%	3.7%
pseudojbb	457.9	58.0	1.0%	-2.0%	1.0%	-0.8%
mtrt	49.0	24.0	7.6%	10.2%	7.7%	11.2%
compress	98.9	4.0	8.3%	2.5%	8.3%	2.5%
health	8.7	8.5	2.3%	-4.0%	1.7%	-6.1%
xalan	473.1	36.1	0.0%	-0.7%	0.0%	0.0%
db	106.1	43.9	9.7%	3.7%	9.7%	4.4%
Average			10.1%	3.7%	10.1%	4.2%

Table 2. Copying and scanning at heap size 2.3×.

Program	Appel		Variation I		Variation II	
	major	minor	major	minor	major	minor
jess	29	1,713	31.0%	8.9%	34.5%	21.1%
ipsixql	77	327	6.5%	0.6%	6.5%	4.0%
jack	12	430	8.3%	11.6%	8.3%	9.3%
bh	21	280	14.3%	3.6%	14.3%	-9.3%
javac	35	533	5.7%	-9.4%	5.7%	1.5%
pseudojbb	14	228	0.0%	-10.1%	0.0%	-4.4%
mtrt	4	178	0.0%	25.8%	0.0%	28.1%
compress	13	16	0.0%	25.0%	0.0%	25.0%
health	2	73	0.0%	-5.5%	0.0%	-9.6%
xalan	13	116	0.0%	-3.4%	0.0%	2.6%
db	10	146	10.0%	-2.7%	10.0%	1.4%
Average			6.9%	4.0%	7.2%	6.3%

Table 3. Number of garbage collections at heap size 2.3×.

ation II performs 10% less copying and 4.2% less scanning than Appel on average. It also reduces the number of major collections by 7.2% and the number of minor collections by 6.3% relative to Appel.

Though our algorithm works better on average, the improvement is uneven. Some programs, like *jess*, improve significantly (39% copying, 22% scanning), while others such as *pseudojbb* don't seem to improve at all (1% copying, -1% scanning). The performance results on *pseudojbb* seem to be related to the stability of the behavior of the nested allocation sites between the training input and the measuring input in *pseudojbb*. The results for *pseudojbb* when the input for the training run is the same as the measurement run (self prediction results that we present in section 6.4) show a significant improvement.

6.2 Responsiveness

We also compare the responsiveness of our realization to Appel. We measure responsiveness by the maximum amount of copying and scanning in a single collection cycle.

Table 4 shows the results for Appel, Variation I, and Variation II for a heap size of 2.3. The results show that on average both variants reduce the maximum copying when compared to Appel, but increase the maximum scanning. Given that copying is more expensive than scanning and for all of the benchmarks, except *bh*, there are also many more bytes copied than scanned, we believe that on average our method is at least as responsive as Appel and likely slightly better. Notice that there is a big variation in the results between the benchmarks: some, like *jess*, are much more responsive with our algorithm, and some, like *bh* and *pseudojbb* are less responsive.

Program	Appel		Variation I		Variation II	
	copied	scanned	copied	scanned	copied	scanned
jess	1.6	1.2	30.2%	-13.6%	25.7%	-13.6%
ipsixql	3.1	1.3	5.4%	-0.5%	5.0%	-0.5%
jack	4.8	1.5	33.9%	-0.7%	30.3%	-0.7%
bh	0.4	1.1	-5.1%	-0.2%	-11.1%	-0.2%
javac	8.7	3.5	2.6%	-0.1%	2.5%	-0.1%
pseudojbb	25.1	1.8	-1.6%	-37.1%	-1.6%	-38.8%
mtrt	8.0	2.3	13.8%	-7.2%	13.5%	-7.2%
compress	6.4	1.0	0.0%	-0.0%	0.0%	-0.0%
health	2.1	1.0	0.9%	0.2%	1.8%	0.2%
xalan	32.2	2.1	0.0%	-2.5%	0.0%	-3.1%
db	7.7	2.2	4.8%	27.4%	4.8%	23.6%
Average			7.7%	-3.1%	6.5%	-3.7%

Table 4. Maximum amount of copying and scanning for a single collection cycle.

6.3 Using Different Heap Sizes

We compare how sensitive the results are to heap size. We measure the same metrics as before, adding measurements for 3 and 5 times the high watermark.

Tables 5 and 6 show the results for heap size 3×, and Tables 7 and 8 show the results for heap size 5×.

Program	Appel		Variation I		Variation II	
	copied	scanned	copied	scanned	copied	scanned
jess	29.0	81.2	27.2%	28.1%	27.0%	19.5%
ipsixql	193.0	64.2	4.8%	1.9%	4.8%	2.0%
jack	62.6	35.7	13.9%	10.3%	13.7%	11.5%
bh	8.6	17.8	12.6%	14.6%	12.8%	17.4%
javac	209.9	82.2	4.8%	0.2%	4.8%	-2.2%
pseudojbb	268.3	38.2	0.5%	-0.3%	0.5%	0.6%
mtrt	25.6	16.8	6.0%	17.7%	6.6%	18.9%
compress	119.8	3.8	1.1%	-0.0%	1.1%	-0.0%
health	6.6	6.8	1.6%	8.4%	1.5%	8.4%
xalan	263.9	17.9	0.0%	-0.9%	0.0%	-0.7%
db	60.8	30.5	4.2%	0.2%	4.3%	0.1%
Average			7.9%	7.6%	7.9%	7.2%

Table 5. Copying and scanning at heap size 3×.

Program	Appel		Variation I		Variation II	
	major	minor	major	minor	major	minor
jess	12	1,115	25.0%	31.3%	25.0%	20.4%
ipsixql	40	202	5.0%	0.5%	5.0%	1.0%
jack	8	338	12.5%	8.6%	12.5%	10.7%
bh	12	232	16.7%	18.1%	16.7%	22.0%
javac	15	266	6.7%	-8.6%	6.7%	-21.1%
pseudojbb	6	128	0.0%	0.0%	0.0%	4.7%
mtrt	1	145	0.0%	35.9%	0.0%	37.9%
compress	13	6	0.0%	0.0%	0.0%	0.0%
health	1	42	0.0%	-4.8%	0.0%	-4.8%
xalan	5	53	0.0%	-5.7%	0.0%	-5.7%
db	4	51	0.0%	-35.3%	0.0%	-9.8%
Average			6.6%	4.0%	6.6%	5.6%

Table 6. Number of garbage collections at heap size 3×.

Both variations of our algorithm perform better in tighter heaps. This is because there are more collections in tighter heaps. Even in large heaps our algorithm continues to perform better than Appel on average. For the heap size 5 times the high watermark Variation II performs on average 3.7% less copying and 3.5% less scanning than Appel. It also performs on average 3% less major collections and 3.6% less minor collections. Also, Variation II continues to

Program	Appel		Variation I		Variation II	
	copied	scanned	copied	scanned	copied	scanned
jess	14.8	35.0	12.8%	-0.3%	7.6%	21.9%
ipsixql	109.4	35.6	1.9%	0.2%	1.9%	-0.5%
jack	49.3	21.1	4.5%	0.7%	4.3%	-0.6%
bh	6.2	12.4	11.9%	16.0%	10.3%	29.4%
javac	106.0	41.8	0.6%	-1.4%	0.7%	-1.1%
pseudojbb	157.6	23.8	0.4%	-0.0%	0.4%	1.9%
mtrt	12.3	4.7	-5.7%	-11.7%	-5.7%	-11.7%
compress	86.8	2.7	10.5%	-4.5%	10.5%	-4.5%
health	4.1	3.3	1.0%	-1.2%	1.0%	-1.2%
xalan	198.9	13.4	-0.0%	-1.5%	-0.0%	-1.0%
db	34.5	23.2	1.3%	-1.1%	1.3%	-1.1%
Average			4.3%	0.2%	3.7%	3.5%

Table 7. Copying and scanning at heap size 5×.

Program	Appel		Variation I		Variation II	
	major	minor	major	minor	major	minor
jess	3	452	0.0%	-1.1%	0.0%	27.7%
ipsixql	13	98	0.0%	2.0%	0.0%	-2.0%
jack	3	159	0.0%	2.5%	0.0%	0.0%
bh	5	156	20.0%	20.5%	20.0%	40.4%
javac	4	104	0.0%	-5.8%	0.0%	-3.8%
pseudojbb	2	52	0.0%	0.0%	0.0%	9.6%
mtrt	0	13	%	-7.7%	%	-7.7%
compress	5	6	0.0%	-16.7%	0.0%	-16.7%
health	0	8	%	0.0%	%	0.0%
xalan	2	26	0.0%	-15.4%	0.0%	-7.7%
db	1	18	0.0%	-5.6%	0.0%	-5.6%
Average			3.0%	-2.1%	3.0%	3.6%

Table 8. Number of garbage collections at heap size 5×.

perform better than variation I on average, although the differences for the bigger heaps become smaller.

In a study performed by Hertz et al. [23] garbage collectors perform better with large heap sizes, in which collections are less frequent. However, sometimes a program already occupies most of the memory of the machine, and a large heap is not an option. In tight heaps a significant portion of the execution time may be due to garbage collection. Therefore, our method might exhibit significant runtime improvements in smaller heaps.

6.4 Self Prediction vs. True Prediction

The previous results are based on true prediction, in other words, they use a medium size input for training and selecting the regions, and a larger input for measuring the metrics. This is in contrast to self prediction, which uses the same input both for training and for measuring the metrics. Self prediction is expected to yield better results, because the system is trained correctly, but is only realistic in online scenarios where the system is, in fact, training on its current inputs. We perform this comparison for Variation II using a heap size factor of 2.3, and show the results in Tables 9 and 10.

The results show a very significant improvement for self prediction. Self prediction performs 22.3% less copying and 7.6% less scanning on average than Appel. Compare this to 10.1% and 4.2% for true prediction. Self prediction also reduces the number of major collections by 18% and the number of minor collections by 10.9% relative to Appel. Compare this to 7.2% and 6.3% for true prediction.

Region selection is more accurate for self prediction. This suggests that we can further improve our region selection technique, for example, by using additional parameters for the heuristic, or by employing an adaptive method that selects the regions at runtime according to parameters also measured at runtime. A combination of offline training and online adaptation may also perform better.

Program	Appel		Variation I		Variation II	
	copied	scanned	copied	scanned	copied	scanned
jess	52.3	127.0	39.0%	21.7%	42.9%	17.2%
ipsixql	284.8	97.4	7.8%	4.9%	7.8%	3.7%
jack	73.6	43.6	13.9%	9.2%	11.7%	9.1%
bh	11.7	22.0	15.0%	-4.1%	16.8%	-17.0%
javac	370.6	145.6	7.1%	3.7%	15.7%	5.3%
pseudojbb	457.9	58.0	1.0%	-0.8%	24.2%	10.1%
mtrt	49.0	24.0	7.7%	11.2%	25.2%	14.8%
compress	98.9	4.0	8.3%	2.5%	8.2%	2.4%
health	8.7	8.5	1.7%	-6.1%	6.0%	-5.0%
xalan	473.1	36.1	0.0%	0.0%	8.1%	8.4%
db	106.1	43.9	9.7%	4.4%	79.0%	35.0%
Average			10.1%	4.2%	22.3%	7.6%

Table 9. Copying and scanning with self prediction.

Program	Appel		Variation I		Variation II	
	major	minor	major	minor	major	minor
jess	29	1,713	34.5%	21.1%	37.9%	15.1%
ipsixql	77	327	6.5%	4.0%	6.5%	-1.5%
jack	12	430	8.3%	9.3%	8.3%	10.7%
bh	21	280	14.3%	-9.3%	9.5%	-26.4%
javac	35	533	5.7%	1.5%	11.4%	-9.0%
pseudojbb	14	228	0.0%	-4.4%	21.4%	3.1%
mtrt	4	178	0.0%	28.1%	25.0%	17.4%
compress	13	16	0.0%	25.0%	0.0%	25.0%
health	2	73	0.0%	-9.6%	0.0%	12.3%
xalan	13	116	0.0%	2.6%	7.7%	4.3%
db	10	146	10.0%	1.4%	70.0%	68.5%
Average			7.2%	6.3%	18.0%	10.9%

Table 10. Number of garbage collections with self prediction.

7. Related work

Section 7.1 surveys research on region memory management, Section 7.2 discusses work related to our memory manager, and Section 7.3 discusses work related to our region selector.

7.1 Memory Management with Regions

In region based memory management, the program allocates heap objects into regions individually, and reclaims all objects in a region simultaneously when none of them are needed anymore.

The original work on regions used *region inference*, a static analysis that decides which objects go into which regions and when to reclaim regions [40, 42, 43]. There are region inference algorithms for ML [41], Java [10], and even C [30]. Similar to our scheme, region inference yields automatic memory management, where the programmer does not need to reclaim memory by hand. Unfortunately, region inference is not always precise enough to free regions in a timely manner, leading to wasted memory. Programmers can work around this problem by profiling memory usage and making their program more region friendly, but then, they lose some of the benefit of automatic memory management. In contrast, our technique is fully automatic and does not depend on precise static analysis.

An alternative approach to region inference is *manually specified* regions, where the programmer decides by hand which objects go into which regions and when to reclaim regions [17, 11, 18]. This approach puts a large burden on the programmer. To find mistakes, the programmer can elect to use manually specified regions with reference counts, in which case region deletion is ignored if

the region's count is non-zero. That means that cycles between unreachable objects in regions will never be reclaimed. In contrast, our technique does not require programmer annotations, and uses garbage collection in addition to reference counts to reclaim memory completely and efficiently.

Some prior work *combines regions with garbage collection*. In RTSJ (the real-time specification for Java), programmers manually specify regions for some objects, and the remaining objects are garbage collected [8]. Similarly, Cyclone gives programmers the choice of manually placing some objects in statically scoped regions and others in the "heap-region", which is garbage collected [19]. Our approach differs from RTSJ and from Cyclone in that we do not require any programmer annotation. Hallenberg, Elsman, and Tofte put all objects into regions, and perform a complete copy of all reachable objects in regions to reclaim the memory of unreachable objects [20]. In contrast, our approach enables partial garbage collections, and puts some objects into the generational heap instead of in regions.

Qian and Hendren present an *adaptive* region-based allocator for Java. They use a write barrier to track which objects escape the method that allocated them. If none of the objects from an allocation site escape, they are kept in a region which is reclaimed when the method returns. This scheme is perhaps the most similar to ours: it requires no static analysis and no programmer annotation, and combines profile-directed regions with garbage collection. The main difference is that while Qian and Hendren use regions for short-lived objects, our memory manager focuses on long-lived objects. We achieve that by decoupling the heuristics for region selection from the mechanism for region reclamation.

7.2 Efficient Memory Management for Long-Lived Objects

Generational garbage collection [31, 44] derives its good performance from short-lived objects, but is less efficient for long-lived objects. Our scheme tackles this issue with regions; this section discusses alternative optimizations for long-lived objects.

Older-first garbage collection finds an "age window" in which garbage collection is most effective, and uses a write barrier to allow a partial garbage collection of just the objects in that window, excluding both younger and older objects [12, 37, 36]. This prevents wasted work when young objects need some time before most of them die. However, for many programs, the young generation is already large enough that young objects have enough time to die, so the heap layout constraints and write-barrier overhead of older-first collection do not pay off. In contrast, we use generational garbage collection for those objects for which it works well. For long-lived objects in regions, our approach spends as little effort as possible while still reclaiming them promptly.

Pretenuring is an addition to standard generational garbage collection that allocates objects that are predicted to live long directly into the old generation [9, 21, 4]. Like our technique, pretenuring is profile-directed and focuses on long-lived objects. However, with pretenuring, long-lived objects exert more pressure on the old generation. If the garbage collector uses copying for the old generation, the pretenured objects require additional space for copy-reserve, and it takes additional time to copy them. If the garbage collector uses mark-sweep for the old generation, the pretenured objects require time and space for free-list and sweeping. A hybrid causes a mixture of these costs [32]. Blackburn et al. compared the performance of copying vs. mark-sweep for old objects in a generational collector [5]. They concluded that copying tends to increase collector time due to copy reserve, but decrease mutator time due to improved locality. Our region-based scheme requires no copy reserve for the regions, yet objects in regions have allocation-order locality.

7.3 Lifetime Characterization and Modeling

This paper introduces a region selector that uses heuristics on the lifetime characteristics from a training run with the Merlin tracer [24]. This approach follows a long tradition of using empirical object lifetime measurements to drive manual optimizations or recommend new algorithms [22, 3, 39, 33, 16, 34, 35, 25, 27, 28]. While our approach benefits from the insight of these studies, it uses lifetime profiles in a more direct manner by automatically selecting regions for the production run.

Another approach to lifetime modeling is to find combinations of radioactive decay models that resemble object survival characteristics [38, 13]. This technique strives at gaining insights from mathematical models, whereas the current paper uses simpler models and judges their quality by how much they improve performance.

8. Conclusions

This paper offers a novel memory management scheme that deals efficiently with long-lived objects. The scheme divides the heap into two sub-heaps: a garbage collected heap and a region heap. Most objects are allocated in the garbage collected heap, while some long-lived objects are allocated in the region heap. The scheme is general and can be realized with a variety of garbage collectors and region selection mechanisms. In particular, the region selection mechanism may vary from manual to fully automatic. The paper introduces an effective automatic profile-based region selector. The region selector measures various metrics on nested allocation sites in order to choose which sites to allocate in regions. We present a realization of the memory management scheme over an Appel-style generational garbage collector and using our novel region selector.

We evaluated our scheme by using a garbage collection simulator. Our experiments show that on a suite of benchmarks run in a tight heap configuration our scheme gains a significant average improvement: 10% less copying, 4.2% less scanning, 7.2% less major collections, and 6.3% less minor collections. The responsiveness of our method not only did not degrade, but even improved slightly. Using self-prediction (measuring the scheme on the same trace used to select the regions) the average improvement is even more significant: 22.3% less copying, 7.6% less scanning, 18% less major collections, and 10.9% fewer minor collections.

References

[1] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 2000.

[2] A. W. Appel. Simple generational garbage collection and fast allocation. *Software – Practice and Experience (SPE)*, 1989.

[3] D. A. Barrett and B. G. Zorn. Using lifetime predictors to improve memory allocation performance. In *Programming Language Design and Implementation (PLDI)*, 1993.

[4] S. Blackburn, S. Singhai, M. Hertz, K. S. McKinley, and J. E. B. Moss. Pretenuring for Java. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2001.

[5] S. M. Blackburn, P. Cheng, and K. S. McKinley. Myths and realities: The performance impact of garbage collection. In *Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2004.

[6] S. M. Blackburn, R. Jones, K. S. McKinley, and J. E. B. Moss. Beltway: Getting around garbage collection gridlock. In *Programming Language Design and Implementation (PLDI)*, 2002.

[7] S. M. Blackburn and K. S. McKinley. Ulterior reference counting: Fast garbage collection without a long wait. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2003.

[8] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, D. Hardin, and M. Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, 2000.

[9] P. Cheng, R. Harper, and P. Lee. Generational stack collection and profile-driven pretenuring. In *Programming Language Design and Implementation (PLDI)*, 1998.

[10] S. Cherem and R. Rugina. Region analysis and transformation for Java programs. In *International Symposium on Memory Management (ISMM)*, 2004.

[11] M. V. Christiansen and P. Velschow. Region-based memory management in Java. Master's thesis, Department of Computer Science (DIKU), University of Copenhagen, 1998.

[12] W. D. Clinger and L. T. Hansen. Generational garbage collection and the radioactive decay model. In *Programming Language Design and Implementation (PLDI)*, 1997.

[13] W. D. Clinger and F. V. Rojas. Linear combinations of radioactive decay models for generational garbage collection. *Science of Computer Programming*, 2006.

[14] G. E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM (CACM)*, 1960.

[15] L. P. Deutsch and D. G. Bobrow. An efficient, incremental, automatic garbage collector. *Communications of the ACM (CACM)*, 1976.

[16] S. Dieckmann and U. Hölzle. A study of allocation behavior of the SPECjvm98 Java benchmarks. In *European Conference for Object-Oriented Programming (ECOOP)*, 1999.

[17] D. Gay and A. Aiken. Memory management with explicit regions. In *Programming Language Design and Implementation (PLDI)*, 1998.

[18] D. Gay and A. Aiken. Language support for regions. In *Programming Language Design and Implementation (PLDI)*, 2001.

[19] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *Programming Language Design and Implementation (PLDI)*, 2002.

[20] N. Hallenberg, M. Elsman, and M. Tofte. Combining region inference and garbage collection. In *Programming Language Design and Implementation (PLDI)*, 2002.

[21] T. Harris. Dynamic adaptive pre-tenuring. In *International Symposium on Memory Management (ISMM)*, 2000.

[22] B. Hayes. Using key object opportunism to collect old objects. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 1991.

[23] M. Hertz and E. D. Berger. Quantifying the performance of garbage collection vs. explicit memory management. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2005.

[24] M. Hertz, S. M. Blackburn, J. E. B. Moss, K. S. McKinley, and D. Stefanović. Generating object lifetime traces with Merlin. *Transactions on Programming Languages and Systems (TOPLAS)*, 2006.

[25] M. Hirzel, A. Diwan, and J. Henkel. On the usefulness of type and liveness accuracy for garbage collection and leak detection. *Transactions on Programming Languages and Systems (TOPLAS)*, 2002.

[26] M. Hirzel, A. Diwan, and M. Hertz. Connectivity-based garbage collection. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2003.

[27] M. Hirzel, J. Henkel, A. Diwan, and M. Hind. Understanding the connectivity of heap objects. In *International Symposium on Memory Management (ISMM)*, 2002.

[28] H. Inoue. *Anomaly Detection in Dynamic Execution Environments*. PhD thesis, University of New Mexico, 2005.

[29] R. Jones and R. Lins. *Garbage collection: Algorithms for automatic dynamic memory management*. John Wiley & Son Ltd., 1996.

[30] C. Lattner and V. Adve. Automatic pool allocation: Improving performance by controlling data structure layout on the heap. In *Programming Language Design and Implementation (PLDI)*, 2005.

[31] H. Lieberman and C. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM (CACM)*, 1983.

[32] P. McGachey and A. L. Hosking. Reducing generational copy reserve overhead with fallback compaction. In *International Symposium on Memory Management (ISMM)*, 2006.

[33] N. Röjemo and C. Runciman. Lag, drag, void, and use — heap profiling and space-efficient compilation revisited. In *International Conference on Functional Programming (ICFP)*, 1996.

[34] R. Shaham, E. K. Kolodner, and M. Sagiv. On the effectiveness of GC in Java. In *International Symposium on Memory Management (ISMM)*, 2000.

[35] R. Shaham, E. K. Kolodner, and M. Sagiv. Heap profiling for space-efficient Java. In *Programming Language Design and Implementation (PLDI)*, 2001.

[36] D. Stefanović, M. Hertz, S. M. Blackburn, K. S. McKinley, and J. E. B. Moss. Older-first garbage collection in practice: Evaluation in a Java virtual machine. In *Workshop on Memory System Performance (MSP)*, 2002.

[37] D. Stefanović, K. S. McKinley, and J. E. B. Moss. Age-based garbage collection. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 1999.

[38] D. Stefanović, K. S. McKinley, and J. E. B. Moss. On models for object lifetime distributions. In *International Symposium on Memory Management (ISMM)*, 2000.

[39] D. Stefanović and J. E. B. Moss. Characterization of object behaviour in Standard ML of New Jersey. In *LISP and Functional Programming*, 1994.

[40] M. Tofte. A brief introduction to regions. In *International Symposium on Memory Management (ISMM)*, 1998.

[41] M. Tofte and L. Birkedal. A region inference algorithm. *Transactions on Programming Languages and Systems (TOPLAS)*, 1998.

[42] M. Tofte, L. Birkedal, M. Elsman, and N. Hallenberg. A retrospective on region-based memory management. *Higher-Order and Symbolic Computation*, 2004.

[43] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 1997.

[44] D. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Software Engineering Symposium on Practical Software Development Environments (SESPSDE)*, 1984.

[45] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In *International Workshop on Memory Management*, 1995.