Mark-Copy: Fast copying GC with less space overhead*

Narendran Sachindran and J. Eliot B. Moss
Department of Computer Science
University of Massachusetts
Amherst, MA 01003, USA
{naren,moss}@cs.umass.edu

ABSTRACT

Copying garbage collectors have a number of advantages over non-copying collectors, including cheap allocation and avoiding fragmentation. However, in order to provide completeness (the guarantee to reclaim each garbage object eventually), standard copying collectors require space equal to twice the size of the maximum live data for a program. We present a *mark-copy* collection algorithm (MC) that extends generational copying collection and significantly reduces the heap space required to run a program. MC reduces space overhead by 75–85% compared with standard copying garbage collectors, increasing the range of applications that can use copying garbage collection. We show that when MC is given the same amount of space as a generational copying collector, it improves total execution time of Java benchmarks significantly in tight heaps, and by 5–10% in moderate size heaps. We also compare the performance of MC with a (non-generational) mark-sweep collector and a hybrid copying/mark-sweep generational collector. We find that MC can run in heaps comparable in size to the minimum heap space required by mark-sweep. We also find that for most benchmarks MC is significantly faster than mark-sweep in small and moderate size heaps. When compared with the hybrid collector, MC improves total execution time by about 5% for some benchmarks, partly by increasing the speed of execution of the application code.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Memory management(garbage collection)*

General Terms

Algorithms, Design, Experimentation, Performance

*This material is based upon work supported by the National Science Foundation under grants CCR-0085792. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

Keywords

Java, copying collector, generational collector, mark-sweep, mark-copy

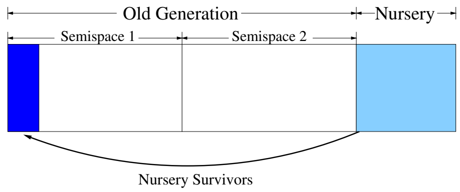
1. INTRODUCTION

Java is becoming increasingly popular as a programming language because of the advantages that it provides, including safety, object orientation, and portability. Garbage collection, which is an important feature of Java, relieves programmers from the burden of explicitly freeing allocated memory, making applications more reliable. However, garbage collection does have an overhead, which can be significant especially when the amount of space available is small.

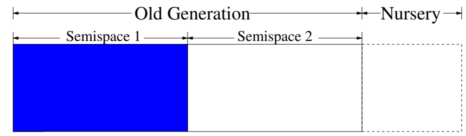
Copying garbage collectors operate by occasionally scanning the application heap (or portions of it) and copying the live (reachable) objects found into a new area in the heap. Since copying collectors always copy objects into contiguous regions, heaps managed by copying collectors exhibit little fragmentation. Also, since free space in the heap is contiguous, allocation is very cheap: it can be performed easily by incrementing a pointer across an unused portion of the heap. Another advantage of copying collectors is that they are relatively simple to implement. However, most copying collectors have a significant space overhead and cannot run in heaps smaller than twice the maximum live data size of a program.

Among current copying collectors, generational collectors [12, 20] are the most widely used. Generational copying collectors divide the heap into multiple regions called *generations*. Generations segregate objects in the heap by age. A two-generation copying collector (2G) uses two regions, an allocation region called a *nursery*, and a promotion region consisting of two semi-spaces called the *old generation*. There are two common types of 2G collectors. A *fixed-size nursery collector* (FG) maintains a constant size nursery, while a *variable-size nursery collector*, e.g., in the manner of Appel [3] (VG), allows the nursery to consume up to half the available space in the heap. VG collectors usually perform better than FG collectors, but at the expense of longer average pauses (periods during which application program execution is suspended in order to perform garbage collection).

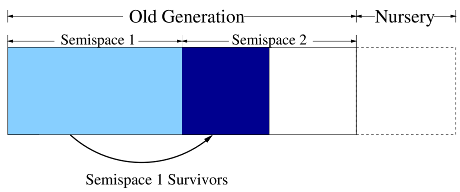
Figure 1 illustrates the functioning of a generational copying collector. The collector triggers garbage collection every time the nursery fills up (Figure 1(a)), and copies reachable nursery objects into a semi-space of the old generation (Semispace 1 in the figure). When the old generation grows to occupy about half the space available (Figure 1(b)), the collector copies reachable objects in the old generation into the other (free) semi-space (Figure 1(c) and (d)). Generational collection is based on the hypothesis that most objects live a very short time, while a small percentage live much longer. The short-lived objects are thus weeded out by frequent nursery



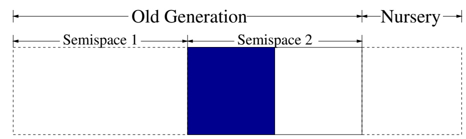
(a) Heap layout during nursery collection



(b) Heap layout before a full collection



(c) Heap layout during a full collection



(d) Heap layout after a full collection

Figure 1: Generational Copying Collection

collections and the space occupied by dead long-lived objects is reclaimed by the less frequent older generation collections.

Since a generational copying collector requires an entire semispace as a region into which to copy survivors during old generation collections (full collections), heap occupancy can never exceed half the heap space. It thus requires space equal to at least twice the maximum live size of a program to be able to operate. Generational copying collectors usually achieve good performance only at heap sizes that are at least 2.5–3 times the maximum live size. This restriction can be quite significant for programs that have large live sizes. Also, the space overhead usually rules out the use of generational copying collection for embedded systems, where the space available is very limited.

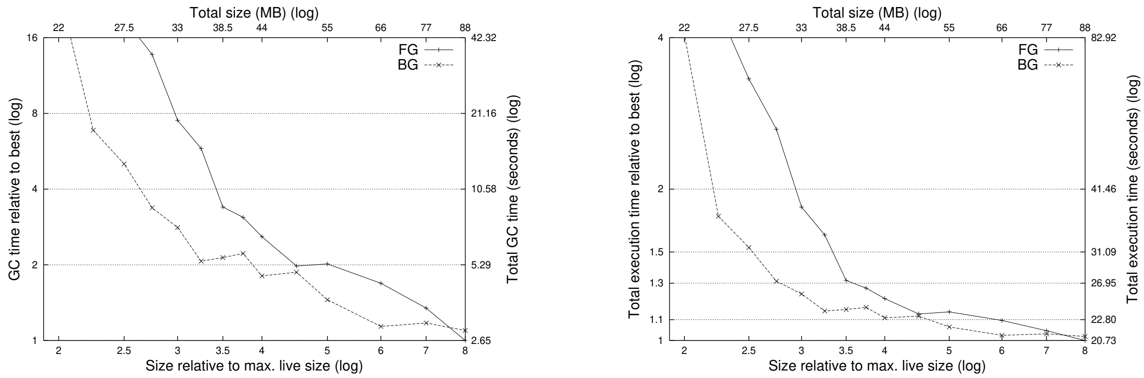
Recently, some schemes have been proposed that require less space than generational copying collectors. The Older First (OF) collector [19, 17, 18] exploits the fact that generational copying collectors prematurely copy the very youngest objects in the nursery. It lays out objects in the heap in order of decreasing age and slides a fixed size window across the heap starting from the oldest objects. At each collection, it copies reachable objects that are in the window, and then slides the window towards younger objects. Thus it usually avoids copying the youngest objects. When the window bumps into the allocation point (i.e., encounters the youngest objects), it is reset to the oldest end of the heap. OF generally reduces the amount of copying and outperforms generational copying collectors. Also, it requires additional space only equal to the size of one collection window, not an entire semi-space. However, OF is not *complete*: it will never reclaim cycles of garbage that are larger than the collection window.

The Beltway collectors [5] include two configurations that perform significantly better than a generational copying collector. The Beltway.X.X collector adds incrementality to the generational copying collector by dividing the generations (called *belts*) into fixed size increments. It collects only one increment at a time, hence the additional free space required at any time is equal to the increment size. However, for any increment value below 100%, the collector is not complete: it suffers from the same problem as OF. The Beltway.X.X.100 collector solves the completeness problem by adding a third belt with a single increment, performing a full collection when the third belt grows as large as half the heap space. Since the Beltway collectors determine the amount of space reserved for copying dynamically, the Beltway.X.X.100 collector does use more than half the heap space. However, since the third belt cannot grow to be larger than half the heap space, the Beltway.X.X.100 collector has the same problem as the generational copying collector: in the general case it cannot run in a heap smaller than twice the maximum live data size for a program.

Our mark-copy (MC) algorithm extends generational copying collection, and allows control over the maximum size to which the old generation can grow, while still providing completeness. By removing the restriction on the usable fraction of the heap space, MC can run in very tight heaps. By using the available space more efficiently, MC can provide better execution times than other copying collectors. The efficient use of space makes MC very useful for reducing memory requirements, and suitable even for Java applications on embedded systems (so long as they do not have real time constraints). A fully incremental version of the MC algorithm can meet real time constraints. We describe such a collector in this paper but leave its implementation and evaluation to future work. We have implemented, and here evaluate, non-incremental and partially incremental versions of MC.

2. FIXED VS. BOUNDED NURSERIES

We first take a closer look at the space utilization of the FG collector compared with the VG collector. VG triggers full collection when the heap is half full. Although it is usually unlikely that all objects in the heap will survive (be reachable in the program), the collector uses this trigger to ensure that there is always enough space to copy surviving objects, even in the worst case. In contrast, while FG can use up to half the space available in the heap, it usually does not. It triggers a full collection when the heap occupancy exceeds $(H/2)-N$ where H is the heap size and N is the nursery size. This is to



(a) GC time (b) Execution time

Figure 2: Performance of an FG collector and a BG collector relative to the best performance for javac

ensure that there is enough space to collect the old generation even if all objects in the nursery survive collection. Thus, very small nurseries allow better utilization of space but perform collection more often. They also tend to have high survival rates (percentage of bytes copied), because in smaller nurseries, the nursery objects have not had as long to die. While larger nurseries have lower survival rates, they do not utilize space as well. For example, an FG collector with a nursery whose size is 10% of the total heap space will generally trigger a full collection when the old generation size grows to a little over 40% of the total space. This effective loss of space can be quite significant when the collector is running in a tight heap. Poor space utilization is one of the primary reasons for the poor performance of FG in tight heaps, as compared to the VG collector.

A simple modification of the FG collector demonstrates the effect that early triggering of full collections has on collector performance. When the available space for FG drops below the nursery size, instead of triggering a full collection, we reduce the nursery size similarly to what VG does. We continue to run the collector with progressively smaller nurseries until either the heap is half full or the nursery size drops below a threshold. We call this a *bounded-size nursery collector* (BG). It differs from VG in that the nursery size never exceeds a predefined bound. Figure 2 shows GC times and execution times for both BG and FG relative to the best time either collector achieves for the javac benchmark (Table 1 details our benchmark suite). We ran both collectors with nursery sizes equal to 20% of the total heap space. The BG collector's nursery lower bound was 512KB. The graphs show that by using the available heap space better and performing fewer full collections, BG can perform significantly better than FG in small and moderate size heaps. At larger heap sizes, the performance of the two collectors is similar since FG needs to perform very few full collections. We implemented both BG and VG versions of MC, omitting an FG version as not being worthwhile to consider.

3. MARK-COPY COLLECTION

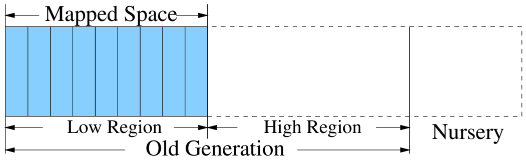
We now describe MC collection in more detail. We first look at the heap layout for MC and the collection algorithm. We then consider the remembered set overhead and space utilization of MC. Next we describe a variant of MC that eliminates the need for remembered sets by using an extra header word per object. Finally we describe how to make MC partially and fully incremental.

3.1 Heap Layout and Collection Procedure

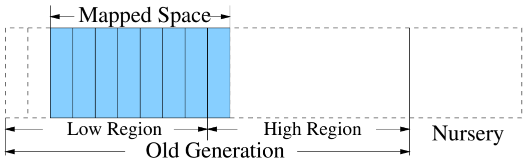
MC extends generational copying collection. Like generational copying collection, it can be implemented with a variable-size nursery (VG-MC), a bounded-size nursery (BG-MC), or a fixed-size nursery (FG-MC). As previously noted, we do not consider FG-MC here. While MC collection has no restriction on the number of generations, we consider only two generations.

MC divides the heap into two areas: a nursery, and an old generation that has two regions. The nursery is identical to a generational collector's nursery, but the old generation regions are broken down into a number of *windows*. The windows are of equal size, and each window corresponds to the smallest increment in the old generation that can be collected. The windows are numbered from 1 to n, with lower numbered windows collected before higher numbered windows. Given a heap of size H and an old generation with n windows, the amount of copy reserve space required is H/n, and the old generation can grow to size $H - H/n$.

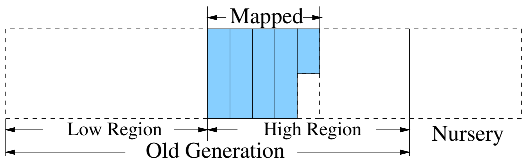
MC performs all allocation in the nursery, and promotes survivors of nursery collection into the old generation. Like a generational copying collector, MC uses a write barrier to record pointer stores that point from the old generation into the nursery. This is done to avoid having to scan the entire heap to find nursery survivors. After each nursery collection, the collector checks the amount of free space remaining in the heap. If it finds that free space in the heap has dropped to the size of a single window (a little more than that in practice), it invokes a full collection. As the first step in the full collection, the collector performs a full heap mark starting from the roots (stack(s), statics). While the marking is in progress, the collector calculates the total volume of live objects in each old generation window. At the same time, it constructs remembered sets for each of these windows. These remembered sets are unidirectional: they record slots in higher numbered windows that point to objects in lower numbered windows. The point is to record pointers whose target may be copied before the source, a condition that requires updating the source pointer when the collector copies the target object. At the end of the mark phase, the collector knows the exact amount of live data in each window. It has also constructed a remembered set for each window. The remembered set entries for each window have the following properties: they are live (not an overestimate of the live set); they are current (i.e., they really point into the window); and they are unique (i.e., the remembered set does not contain any duplicates).



(a) Heap layout before a full collection



(b) Heap layout after one copying pass



(c) Heap layout after three copying passes

Figure 3: Heap Layout during full MC collection, with a 50% survival rate in each window

Once the old generation mark phase is complete, the collector performs the copy phase. The copy phase is broken down into a number of *passes*, each pass copying a subset of the windows in the old generation. The collector has the option either of copying the old generation windows one after the other without performing any allocation in between, or of interleaving copying with nursery allocation. We focus on the first technique here, as our goal is increased throughput and not lower pause times, and discuss the incremental possibilities in Section 3.4. Since the collector knows the exact volume of live data in each old generation window and also the total free space available in the heap, it may collect multiple windows in a single pass. However, since the remembered sets are unidirectional, the windows must be collected strictly in order of window number, and not in any arbitrary order. One way to overcome this restriction is to build bidirectional remembered sets, which store pointers into each window from objects in all other windows. However, this would increase the space overhead of the remembered sets and complicate their management. We do not support bidirectional remembered sets in our current implementation.

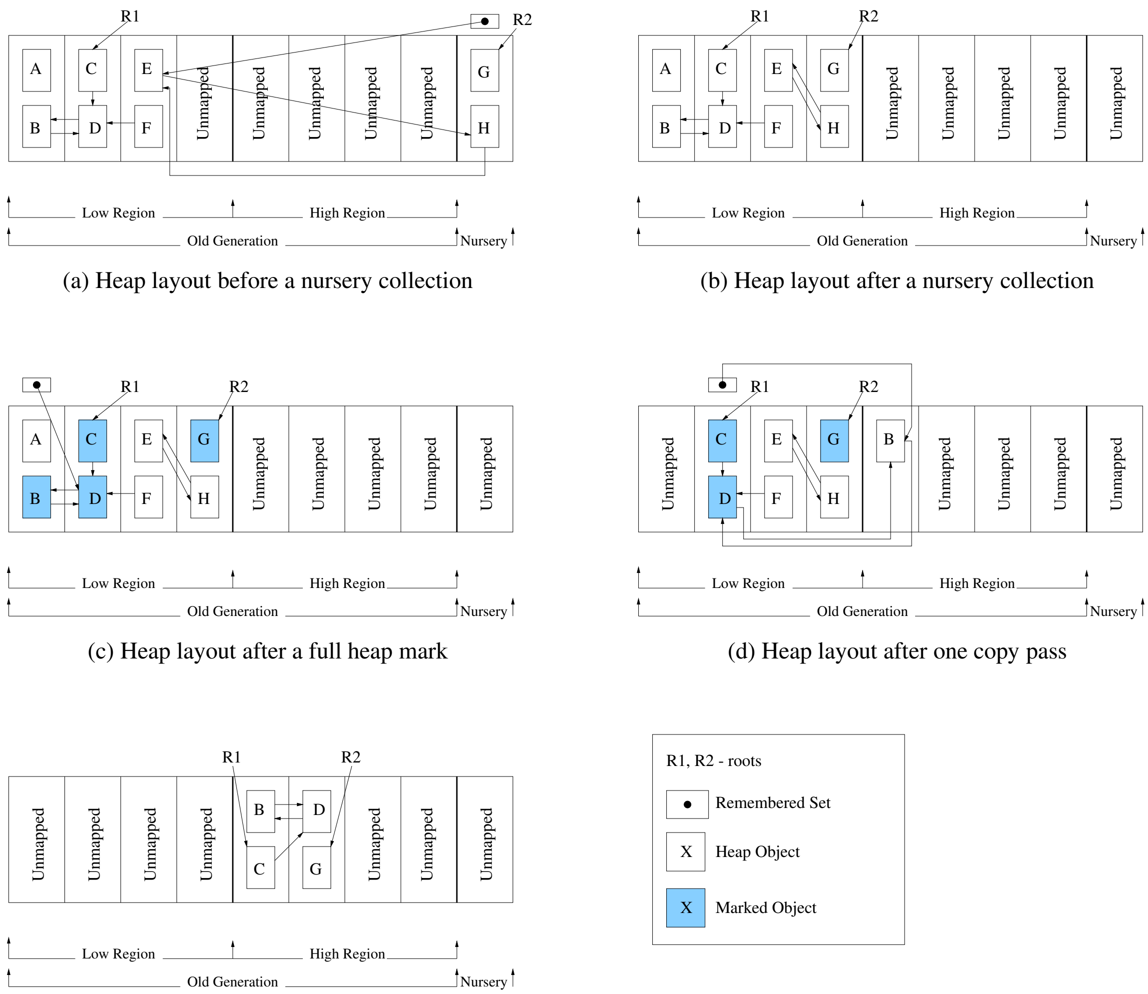
Figure 3 shows the virtual memory layout of the collector performing copying in a heap whose old generation regions are divided into nine windows each. Since each old generation region is divided into nine windows, the old generation can occupy up to 90% of the total heap space. Before copying commences, the nine windows in the low region are all full and the heap has enough free space for a single window in the high region (Figure 3(a)). In this particular case, we assume that the amount of live data in each low region window is exactly 50%. The first copy pass scans live objects in the first two windows in the low region and copies them into the first window of the high region. The roots for the collection are the stack(s), statics, and remembered set slots. When copying is complete, the total amount of mapped virtual memory equals the total heap space. At this point, the space consumed by the first two windows in the low region is released (unmapped), as shown in Figure 3(b). This means that the mapped space after the completion of one copying pass is 80% of the maximum total heap space. This now allows objects from four windows in the low region to be copied in the next pass. Finally in the third pass, the objects from the last three windows are copied, leaving the heap as shown in Figure 3(c).

Figure 4 shows a detailed example of a full collection using MC. For this example we assume that all objects allocated in the heap have the same size, and that the heap can accommodate at most 10 objects. The heap consists of an old generation with 4 windows. Each of these windows can hold exactly 2 objects. R1 and R2 are root pointers. Figure 4(a) shows a nursery collection, which results in objects G and H being copied into the old generation. G is copied because it is reachable from a root, and H is copied because it is reachable from an object (E) in the old generation. At this point, the old generation is full (Figure 4(b)). MC first performs a full heap mark and finds objects B, C, D, and G to be live. During the mark phase it builds a unidirectional remembered set for each window. After the mark phase (Figure 4(c)), the remembered set for the first window contains a single entry (D→B). All other remembered sets are empty, since there are no pointers into the windows from live objects in higher numbered windows. (If we used bidirectional remembered sets, the second window would contain an entry to record the pointer from B to D.) In the first copy pass, there is enough space to copy two objects. Since the first window contains one live object (B) and the second window contains two live objects (C, D), only the first window can be processed in this pass. MC copies B to a high region window and then unmaps the space occupied by the first window (Figure 4(d)). It also adds a remembered set entry to the second window, to record the pointer from B to D (since B is now in a higher numbered window than D, and B needs to be updated when D is moved). The old generation now contains enough free space to copy 3 objects. In the next copying pass, MC copies the other 3 live objects and then frees up the space occupied by windows 2, 3, and 4 (Figure 4(e)).

The mark phase of the MC collector serves two purposes:

- By calculating the free space in each window, the mark phase minimizes the number of passes the copy phase needs to make in order to copy all the data. For example, for an old generation with nine windows and an overall survival rate of 10%, MC needs only one copying pass. This is because the copy reserve of 10% is enough to accommodate all the collection survivors. However, in the worst case, when the survival rate is over 90%, it will need nine passes to copy the data. One should, however, note that for survival rates over 50% a standard copying collector would not even be able to run the program in this space.
- By building remembered sets, the mark phase ensures that the copy phase needs to perform only a single scan over the old generation objects in spite of having to perform the copying in multiple passes.

From the above description, we can see that the amount of space reserved for copying can be made extremely small by increasing the number of old generation windows. However, the minimum copy reserve space needs to be at least as large as the largest object allocated in the heap. By allowing control over the copy reserve, the collector is able to run in very tight heaps. It also makes the memory footprint of the collector much more predictable than that of a typical copying collector. For a heap of size H, the footprint of a generational copying collector varies between $H/2$ and H, i.e., by



(a) Heap layout before a nursery collection

(b) Heap layout after a nursery collection

(c) Heap layout after a full heap mark

(d) Heap layout after one copy pass

(e) Heap layout after two copy passes

Figure 4: MC - Full collection example

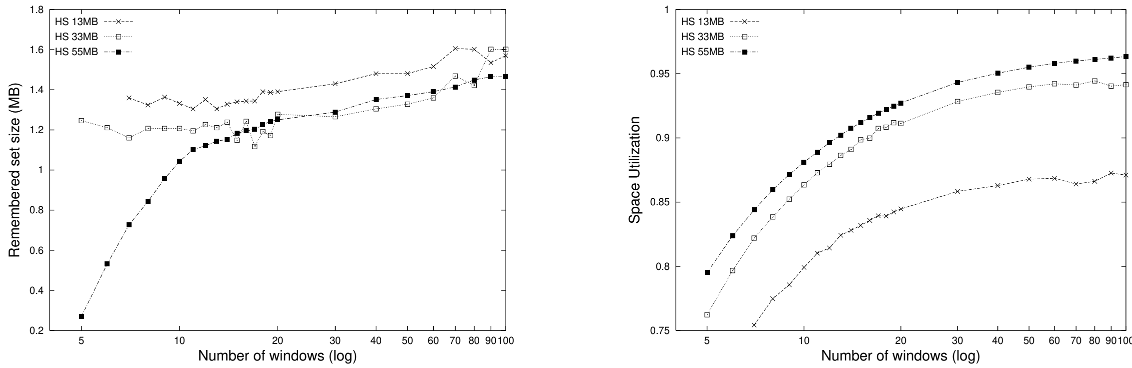
a factor of 2. However, given an old generation that has n windows, the footprint of the MC collector varies between $H - H/n$ and H, a factor of $n/(n-1)$. Since we can control the value of n, we can minimize the variation (subject to the maximum object size).

Also, MC defers full collections until the free space drops down to one window. Standard copying collectors perform full collections when the heap becomes half full. Hence, when MC and other copying collectors are given the same amount of space, MC will perform many fewer full collections. This decreased full collection frequency gives objects in the old generation a longer time to die, resulting in much less copying than regular copying collection. Another way to think of this is that MC increases the *effective* heap size.

MC collectors do have some disadvantages compared with generational copying collectors. Each full collection for MC will scan every live object in the heap twice, once while marking and once while copying. The mark phase requires additional space for a mark stack and there is a copying remembered set overhead. Also, for an old generation consisting of n windows, up to n passes (but generally many fewer) over the stacks and statics may be required. This could be significant if there are a large number of threads or the thread stacks are very deep. This overhead could be reduced by storing these slots in the remembered sets during the mark phase. However, doing so would require additional space. While we do not offer details here, we found that scanning stacks and statics did not make a large contribution to GC time for our benchmark suite.

3.2 Remembered Sets

The MC collector builds remembered sets at two different points during program execution. While the mutator is running, the write barrier inserts slots into the nursery remembered set. This is identical to what a generational copying collector does and occupies the same amount of space. During the mark phase of a full collection, MC constructs unidirectional remembered sets to record pointers between windows in the old generation. These remembered sets are not required by a generational copying collector and are an additional space overhead for MC. However, the nursery and old generation remembered sets do not co-exist in a non-incremental MC



(a) Remembered set size

(b) Space utilization

Figure 5: Effect of the number of old generation windows on the space usage of MC for javac

collector, since MC performs old generation marking only after a nursery collection completes. We look at the overhead of old generation remembered sets here.

Normally we would expect the remembered set space overhead to increase as the number of old generation windows increases. This does not necessarily happen, for a couple of reasons. First, the set size depends on the locations of the objects in the old generation, i.e., if objects that reference each other lie close together in the old generation, then increasing the number of windows will not have a significant impact. Second, an increase in the number of windows increases the amount of usable space in the heap. This in turn changes the points at which collection occurs, and hence the amount of live data in the heap during full collection. Since the remembered set entries are accurate and depend on the amount of live data in the heap, a change in the volume of live data could reduce the remembered set size.

Figure 5(a) shows the variation in the maximum overall remembered set size as one varies the number of windows, for heap sizes which are 1.2, 3, and 5 times the maximum live size (13MB, 33MB, 55MB), for javac. The horizontal axis represents the number of windows and is drawn to a logarithmic scale. We look at the remembered set sizes for an old generation with number of windows ranging from 5 to 100. Although we do not use more than 20 windows in the old generation with the non-incremental collector, it is important to see what the remembered set overhead is with a large number of windows, since an incremental collector might need to use 100 or more windows to ensure a low pause time.

The remembered set sizes drop initially in the smaller heaps, and then generally increase as the number of windows is increased. The size in the smallest heap is about 1.35MB with 7 windows and about 1.57MB with 100 windows. The size in the largest heap is about five times larger with 100 windows than with 5. However, if we look at the remembered set size as a fraction of the total heap space, it is not very large even when we use 100 windows (12% in the smallest heap and 2.6% in the largest heap).

Figure 5(b) shows the space utilization for MC as we increase the number of old generation windows. We calculate the space utilization for a given number of windows n using the formula $(H - (H/n) - R)/H$, where H is the heap size, H/n is the copy reserve size, and R is the maximum remembered set size. For all heap sizes, we obtain significant gains in space utilization until the number of old generation windows grows to about 40. This is because the copy reserve is reduced by a larger amount than the corresponding increase in remembered set size. Increasing the number of windows beyond 50 causes the space utilization to improve slightly or even to deteriorate a little.

3.3 MC without Remembered Sets

Although the remembered set overhead for MC is not usually very high, in the worst case remembered sets could grow to be as large as the heap. We describe in this section a variant of MC that uses an extra header word per object in the old generation (MCHW), eliminating the need for remembered sets and thus bounding the worst case space utilization. However, unlike MC, MCHW can be used only as a non-incremental collector.

The heap layout for MCHW is identical to the layout for MC. MCHW performs allocation in the nursery and promotes survivors to the old generation. MCHW adds an extra header word to each object that is copied into the old generation. When the free space in the heap drops down to the size of a single window, it invokes a full collection. Like MC, MCHW performs a full collection in two phases.

During the mark phase, MCHW marks all objects reachable from roots. While performing the marking, it maintains a *live offset array* that stores the sum of the sizes of the live objects found so far in each window. It also maintains a bitmap that indicates the locations of the live objects. While marking a reachable object, the collector computes a *logical address* for the object. The logical address is a combination of the window to which the object belongs and the live offset of the object within the window. The logical address is stored in the extra header word reserved for the object and it indicates the location to which the object will be copied in the next phase. Every time a slot referencing the object is found, the contents of the slot are replaced with the logical address of the object. Figure 6 shows the logical address layout used for MCHW. The lowest bit is set to 1 to indicate that the value stored is a logical address. (Objects are word aligned and the machine is byte addressed, so ordinary object pointers always have zeroes in the two low order bits.) The middle bits store the live offset, and the high order bits store the window number.

Once the mark phase is complete, the live offset array contains the sum of sizes of the live objects in each window. The collector now calculates a cumulative live offset (CLO), which for each window is the sum of the offsets of all preceding windows. Using this information, the new address for any heap object is calculated

Window	Live Offset	1

Figure 6: Logical address layout for MCHW

using the logical address (LA) of the object and the start address of the copy space (CSA) using the formula:

$$CSA + CLO[Window(LA)] + LiveOffset(LA)$$

The MCHW collector requires that the entire copy space be contiguous (in virtual memory). Otherwise it will need to perform an extra pass over the live objects to determine the correct cumulative live offset (since objects may span two windows). Once the offsets are calculated, the collector starts the copy phase. It first scans the roots and updates pointers into the heap using the above formula. It then scans the bitmap, processing one window at a time. For each live object in the window being processed, it updates any pointers that the object contains using the above formula, and then copies the object to the to-space location computed from its logical address. After processing all live objects in a window, it unmaps the space occupied by the window and then processes the next. The copy phase terminates when all windows have been processed.

As can be seen from the description, MCHW does bound the worst case space utilization. It requires one additional word per object plus space for a bitmap. The size of the bitmap is approximately 3% of the heap space when objects are word aligned (one bit per 32-bit word). However, since it adds an extra word to each object, the layout of the old generation is not as compact as that for MC, which could cause negative locality effects and also reduce the effective heap size.

3.4 Incremental Collection

Until now we have dealt only with batched collection of windows in the old generation. Although the collector performs the copying in increments, it performs multiple passes one after the other until all the copying is finished. This scheme is meant to optimize throughput and does not necessarily give good maximum pause times. The maximum pause time is typically close to that of the generational copying collectors. A fully incremental version of BG-MC trades some loss in throughput for reduced maximum pause times. We now describe how we can construct a fully incremental version, MCI, of the BG-MC collector, describing a partially incremental version along the way.

BG-MC consists of three different collection phases: nursery collection, old generation marking, and old generation copying. In order to make the collector fully incremental, we need to bound the amount of copying and marking work done in each phase. The amount of copying performed during nursery collection is bounded because the size of the nursery is bounded. The copying and marking work done in the other two phases can also be bounded. We first describe how the collector can perform incremental copying while using a non-incremental mark phase (MCIC, IC for *incremental copying*). We then describe how to make the mark phase incremental (MCI).

Incremental old generation copying: When the marking of the old generation is complete, we know the exact amount of data that will be copied out of each of the old generation windows. Using this information, we divide the old generation copying work across one or more nursery collections. We first group the old generation windows based on the amount of live data in each window. Each *group* consists of one or more adjacent old generation windows, with the condition that the total amount of live data in a group is less than or equal to the size of a single old generation window.

Once the grouping is done, we copy objects within the first group and give the freed memory (less one window) to the nursery allocator (i.e., we can use up to half of those windows for nursery allocation, subject also to the BG bound on nursery size). We then resume program execution and the program runs until the nursery fills up, which we then collect. At this point, we copy the next group of windows in the old generation. We repeat this process of alternating program execution with nursery then old generation group collection until the entire old generation is copied, thus spreading the old generation copying work across several nursery collections. In the case that an old generation collection does not free enough space, we repeatedly collect old generation groups until we free enough memory for the nursery. A problem MCIC can run into is that a large number of full windows may lie next to each other in the heap. This causes the collector to perform a large number of consecutive copy passes, resulting in a longer pause. One solution for this problem is to use bidirectional remembered sets, which allow MCIC to collect windows in any order, so that it can collect windows that are mostly full along with windows that are mostly empty. Bidirectional remembered sets, however, have additional space overhead and increase write barrier cost.

We can also extend the incremental copying so that it does not perform old generation copying immediately after nursery collection. We call this the *split-phase* approach. In the split-phase approach, after we copy the first group of windows in the old generation, we allow the program to run until the nursery becomes *half* full. At this point, we copy the next old generation window group. We then allow the program to run until the nursery becomes (entirely) full, at which point we perform a nursery collection. This approach tends to spread out the copying pauses better.

There are some policy considerations in MCIC collection. One is when to trigger old generation marking. We should mark somewhat in advance of filling the old generation, to reduce the possibility of the collector's needing to collect several groups immediately after one nursery collection. However, if we mark too much in advance, then there are two negative effects: old objects have not had as much time to die, so we reclaim fewer dead ones; and meanwhile we have more objects promoted from the nursery, which we cannot reclaim until the *next* old generation collection. It is clear that in general MCIC will need more space than MC to perform reasonably, and will likely do more total work than MC. This is an expected tradeoff for incrementality in collection.

A second policy consideration is how often to collect old generation groups. The policy we gave above is to collect one group after each nursery collection (or, in split-phase, between each nursery collection and the next one). However, we could wait for more than one nursery collection, e.g., until nursery collection does not leave enough blocks for nursery allocation. We would then collect one group, hoping to free enough memory for nursery allocation to proceed. Doing group collection as rarely as possible defers reclaiming old generation memory, so we expect it will put more pressure on the nursery and cause more frequent nursery collections. It might also require an *earlier* marking trigger.

A third policy variation is to allow collection of more than one group at (or between) nursery collections. Since the goal is to bound maximum pause time, if the maximum nursery size is more than one window, say k windows, then, since a nursery collection could copy k windows of objects, we could allow MCIC to collect up to k groups (or, enough from-space windows to fill k to-space windows), if there is space available. Collecting groups sooner reclaims their free space sooner, allowing larger nurseries, etc. The point is that once we have marked, the only reason to wait to reclaim dead objects is to spread pauses out: even if more old generation objects die, we will not reclaim them until the *next* old generation marking and copying.

The MCIC collector has additional overhead compared with the MC collector. First, it performs a larger number of garbage collections. Second, it must use a more complex write barrier. MC needs to track pointers into only the nursery, whereas MCIC needs to track pointers into both the nursery and the uncollected portion of the old generation. This means that a simple directional write barrier cannot be used with the incremental copying collector (unless one runs the algorithm in a large address space, e.g., 64 bits, as has been proposed for OF). Remembered set sizes also tend to be larger for MCIC.

Incremental marking: MCIC performs non-incremental marking when it triggers old generation collection. We can bound the amount of marking work by replacing the non-incremental mark phase with incremental marking. Incremental marking can perform small amounts of marking work along with each collection, or even during allocation. In order to support incremental marking, the collector requires a more complex write barrier: it must log any pointer assignment whose target is an unmarked object in the region being marked. This will ensure that all reachable old generation objects are marked.

Replacing non-incremental marking with incremental marking will affect performance. Incremental marking must be started much sooner than a non-incremental mark phase, to ensure that marking completes before the collector runs out of space. As a consequence, the set of objects marked at the end of the mark phase will be an overestimate (larger, compared with MCIC) of the true set of reachable objects in the old generation. This in turn will increase the amount of copying work. The more complex write barrier will also reduce the performance of the collector.

Implementation: We have implemented a version of MCIC that performs incremental copying and uses unidirectional remembered sets. We describe some of the results from this collector and leave the implementation of MCI to future work.

4. RESULTS

We describe in this section our experimental setup, details of the collector implementation, the benchmarks we used, and experimental results comparing the collectors. We compare space overheads, copying costs, GC times, and total execution times for the VG, VG-MC, and VG-MCHW collectors. We also compare GC times and total execution times of VG-MC with a (non-generational) mark-sweep collector (MS) and a hybrid copying/mark-sweep generational collector that uses a variable size nursery (VG-MS). For the MC collectors, we use 20 windows in the old generation. For each benchmark, we ran the collectors in heaps ranging from the minimum space required to run the benchmark to 8 times the maximum live size.

4.1 Experimental Setup

We implemented our collector in Jikes RVM 2.2.2 [1, 2]. Jikes RVM does not have an interpreter: it compiles all bytecode to native code before execution. Jikes RVM has two compilers, a baseline compiler that essentially macro-expands each bytecode into non-optimized machine code, and an optimizing compiler. It also has an adaptive run-time system that first baseline compiles all methods and later optimizes methods that execute frequently. Methods can be optimized at three different levels depending on the execution frequency. However, the adaptive system does not produce reproducible results, since it uses timers and may optimize different methods in different runs.

We used a *pseudo-adaptive* configuration to run our experiments with reproducible results. We first ran each benchmark 7 times with the adaptive run-time system, logging the names of methods that were optimized and their optimization levels. We then determined the methods that were optimized in a majority of the runs, and the highest level to which each of these methods was optimized in a majority of runs. We ran our experiments with only these methods always optimized (to that optimization level) and all other methods always baseline compiled. The resulting system behavior is repeatable, and does very nearly the same total allocation as a typical adaptive system run; it differs from adaptive system behavior in that it tends to invoke the optimizing compiler before the application has built up its live data set, whereas adaptive runs tend to invoke the (memory-hungry) optimizing compiler in the thick of the application. Thus, adaptive system maximum live size tends to be bigger (and unpredictable). However, the pseudo-adaptive system, since its average and peak live sizes are closer to one another, tends to run closer to its peak. Thus, when scaled by live size, pseudo-adaptive is consistently closer to its peak memory usage more of the time.

Jikes RVM is itself written in Java, and some system classes can be compiled either at run time or at system build time. We compiled all the system classes at build time to avoid any non-application compilation at run time. The system classes are stored in a region called the *boot image*, that is separate from the program heap. We used the Java memory management toolkit (JMTk), standard with Jikes RVM 2.2.2, as the base collector framework. JMTk already supplied the generational collector and mark-sweep collectors. JMTk also provided us with most of the generic functionality required by a copying collector, so it aided rapid deployment of our two new collectors.

We ran our experiments on a Macintosh PowerPC with two 533 MHz G4 7410 processors (though the system uses only one of them), 32 KB on-chip L1 data and instruction caches, 1 MB unified L2 cache and 640 MB of memory, running PPC Linux 2.4.10. We performed our experiments with the machine in single user mode.

4.2 Implementation Details

We first describe the implementation of the non-incremental version of MC. The collector divides the entire usable virtual address space into a number of *regions*. The lowest region stores boot image objects, the next region stores immortal objects, and a third region stores large objects. All these regions come by default in the JMTk framework, for any collector. MC allocates objects larger than 8KB into the large object region, and this region is managed by the JMTk mark-sweep collector (the size threshold for allocation into this region is 8KB for all collectors that we compared MC against). JMTk rounds up the size of large objects to whole pages (4KB), and allocates and frees them in page-grained units.

The type information block (TIB) objects, which include the virtual method dispatch vectors, etc., and which are pointed to from the header of each object of their type, are allocated by JMTk into immortal space. This has a couple of benefits for the MC collector. First, the TIB objects do not have to be copied when performing collection. Second, the remembered set overhead is reduced significantly since we do not store pointers into immortal space in the remembered sets.

MC creates two more regions, one for the old generation and one for the nursery. The old generation is divided into a number of fixed size *frames*. A frame is the largest contiguous chunk of memory into which allocation can be performed. The frame size in our implementation is 8MB. Each frame accommodates at most one old generation window. Old generation windows can, however, span multiple frames. The portion of the address space within a

Benchmark	Description	Maximum live size (MB)	Total Allocation (MB)
_202_jess	a Java expert system shell	4.0	291
_209_db	a small data management program	10.5	85
_213_javac	a Java compiler	11.0	285
_222_mpegaudio	an MPEG audio decoder	3.0	27
_227_mtrt	a dual-threaded ray tracer	10.0	145
_228_jack	a parser generator	4.5	329
pseudojbb	SPEC JBB200 with a fixed number of transactions	28.0	334
health	simulation of a health care system	67.5	552

Table 1: Description of the benchmarks used in the experiments

frame that is not occupied by a window is left unused. Since the frames are power-of-two aligned, only a single shift is required to find the frame number of a heap object during garbage collection.

MC uses a fast write barrier that records only pointer stores that cross the boundary separating the nursery region from the rest of the heap; this is the same barrier used by the generational copying collectors. The write barrier is partially inlined [6]: the code that tests for a store of an interesting pointer is inlined, and the code that inserts interesting slots into a remembered set is out of line. Each frame has an associated remembered set. The remembered set is implemented as a sequential store buffer [9]. Remembered sets are used to store the addresses of slots that reside in the heap and the boot image and that contain interesting pointers.

The MCHW implementation uses the same memory layout as MC, except for the layout of the old generation. The old generation for MCHW is not divided into power-of-two aligned frames. Instead, it is divided into page-size aligned windows when a full collection is performed. This ensures that objects are allocated contiguously in the old generation (which avoids an extra pass over the heap to calculate addresses). This condition does not necessarily hold true when frames are used since the window size and frame size are usually not equal.

We also implemented the MCIC collector. The write barrier for MCIC is slower than the one used for MC, since simple boundary crossing checks are not adequate. The full heap mark for MCIC is triggered sooner than it is for MC (when the heap is 85% full). This is because delaying the marking until the heap is almost full tends to cause multiple copying passes to be clustered together, causing long copying pauses in addition to the marking pause.

4.3 Benchmarks

We compare results from eight benchmarks, six from the SPEC JVM98 suite [15], plus pseudojbb, and health. pseudojbb is a modified version of the SPEC JBB2000 benchmark [16]. pseudojbb executes a fixed number of transactions (70000), which allows better comparison of the performance of the different collectors. health is an object oriented Java version of the Olden C benchmark [14]. We do not present results from two SPEC JVM98 benchmarks, _201_compress and _205_raytrace. _201_compress mostly allocates large objects, which are placed in large object space. Since the large object space for all the collectors is managed using mark-sweep, this benchmark is not interesting for our study. _227_mtrt is a dual threaded version of _205_raytrace. Since the results for the two benchmarks are very similar, we present results only for _227_mtrt. We ran all SPEC benchmarks using the default parameters, and ignoring explicit GC requests. For health, we set the number of levels to 8 and maximum time to 200.

Table 1 describes each of the benchmarks we used. We calculated the live size for a benchmark by finding the smallest heap in which it ran (rounding up to the next MB) with a VG collector that did *not* use a large object space (except for the thread stacks), and dividing the result by 2.

4.4 Space Accounting

In all the experiments we ran, the VG collector used the specified heap space to allocate heap objects and nursery remembered set entries. The nursery remembered set space was not accounted for separately. MC also used the heap space in the same manner. However, the old generation remembered sets were accounted for after each run completed, by adding the maximum space occupied by the remembered sets to the heap space. MCHW used the specified heap space to store heap objects (along with an extra header word), the nursery remembered set, and a bitmap.

4.5 Mark-Copy Space Overheads

Table 2 shows the remembered set space overhead for all eight benchmarks, for an MC collector that has 20 windows in the old generation. For each benchmark, the table shows the remembered set size for heap sizes ranging from 1.1 to 6 times the live size. The heap sizes do not include the remembered set size. They do however include a 7.5% copy reserve (we trigger a full collection when the free space drops down to 1.5 windows, so that the collector does not use nurseries smaller than 2.5% of the heap space). Each entry in the table represents the remembered set size as a percent of the heap size. An entry with a '–' indicates that MC did not perform a full collection for the heap size, and hence did not construct remembered sets.

For four of the eight benchmarks, the remembered set overhead for MC is always less than 5% of the maximum live size. For javac the overhead is about 11% of the live size, and for health the overhead is about 15%. The total minimum space overhead for MC varies between 12% and 25% of the maximum live size.

The header word overhead for MCHW is higher than the remembered set overhead for MC. Table 3 shows the overhead for the eight benchmarks. We calculate the overhead by dividing the header word size (4 bytes) by the average old generation object size (excluding the header word). This is because the header word is added only to old generation objects. The space overhead for MCHW is about 5–10% higher than the overhead for MC, still considerably lower than the generational copying overhead.

4.6 Copying Costs

We examine here the copying characteristics of VG-MC and VG. Although the performance of copying collectors is influenced by factors such as locality, the total amount of data copied is a good performance indicator and can explain differences in overall performance. For generational collectors, the amount of copying is strongly related to the number of full collections. We first look

Benchmark	Heap size relative to maximum live size						
	1.1	1.2	1.5	2	3	4	6
_202_jess	8.8%	8.3%	5.1%	1.4%	–	–	–
_209_db	1.6%	1.5%	1.0%	0.8%	0.3%	–	–
_213_javac	10.3%	9.2%	7.1%	4.8%	3.7%	2.8%	–
_222_mpegaudio	4.4%	3.7%	2.9%	2.0%	–	–	–
_227_mtrt	2.9%	2.1%	1.6%	–	–	–	–
_228_jack	5.1%	2.7%	2.6%	0.9%	0.6%	–	–
pseudojbb	1.7%	1.6%	1.2%	0.8%	–	–	–
health	14.2%	11.9%	–	–	–	–	–

Table 2: Remembered set size as a percent of the heap size, for MC using 20 windows in the old generation

Benchmark	Header Word overhead
_202_jess	9.0–10.7%
_209_db	13.5–13.7%
_213_javac	13.9–14.1%
_222_mpegaudio	9.7–11.0%
_227_mtrt	14.5–15.9%
_228_jack	11.3–11.6%
pseudojbb	5.2–6.6%
health	18.2–18.5%

Table 3: Average header word space overhead for MCHW

at the full collection behavior of VG-MC and VG for two benchmarks, and then see how this affects the amount of data copied.

Figure 7 shows the number of full heap collections performed by VG-MC and VG for javac and pseudojbb. The horizontal axis represents the total space used relative to the maximum live size, and is drawn to a logarithmic scale. The vertical axis represents the total number of full collections performed. For javac, VG-MC has a significant advantage until the heap has grown to 4.5 times the maximum live size. In space that is about 2.3 times the maximum live size, VG-MC performs as few as three full collections. VG performs approximately five times as many full collections at this point. The performance for pseudojbb is even better, with VG-MC performing only one full collection in space that is about two times the maximum live size. VG catches up with VG-MC only when the space grows to five times the maximum live size.

VG-MC performs many fewer full collections because it defers full collection until the heap is almost full, while VG has to perform them when the heap is half full. The survival rate of objects in the old generation is much higher than nursery objects, since the old generation typically contains long lived objects. For example, for VG running javac in a heap that is 2.3 times the live size, 66% of old generation data survives on average, while only 21% of data in the nursery survives. By deferring full collections, VG-MC gives the old generation objects a longer time to die, thus significantly reducing the survival rate, which in turn reduces the total amount of copying work. Therefore, even though VG-MC potentially does more work at each full collection, the lower frequency allows it to do much less work overall.

Figure 8 shows the effect of the number of full collections on the amount of copying performed by the collectors. The graphs show the relative *mark/cons* ratio, the ratio of total bytes marked and copied ("marked") to total bytes allocated ("cons'ed"), for VG-MC and VG, for javac and pseudojbb. For javac, VG copies approximately two times more data than VG-MC in space that is about 2.3 times the maximum live size. At four times the live size, VG copies 30% more data than VG-MC. In heaps that are six times the live size or larger, VG-MC does not perform full collections and the copying cost for VG is 20-25% higher. For pseudojbb, the copying cost for VG is twice as much as that for VG-MC, in space that is about 2.5 times the maximum live size. In heaps between 2.5–4.5 times the live size, the cost is at least 50% higher. Eventually, at six times the live size, the collectors perform roughly equal amounts of copying. We look at the effect this reduced copying has on overall performance of the collectors in the next section.

4.7 Mark-Copy vs. Copying Collection

Figure 9 shows GC times for VG-MC, VG-MCHW, and VG, relative to the best GC time for the eight benchmarks. Figure 10 shows the total execution times for the collectors relative to the best total execution time. Although we measured the performance of BG-MC, we omit the results in the graphs in order to make them legible. We briefly discuss the results for BG-MC at the end of the section. In the graphs, the vertical axis represents relative times, and is drawn to a logarithmic scale. The horizontal axis represents total space relative to the maximum live size, and is also drawn to a logarithmic scale, so as to show greater detail for smaller heap sizes.

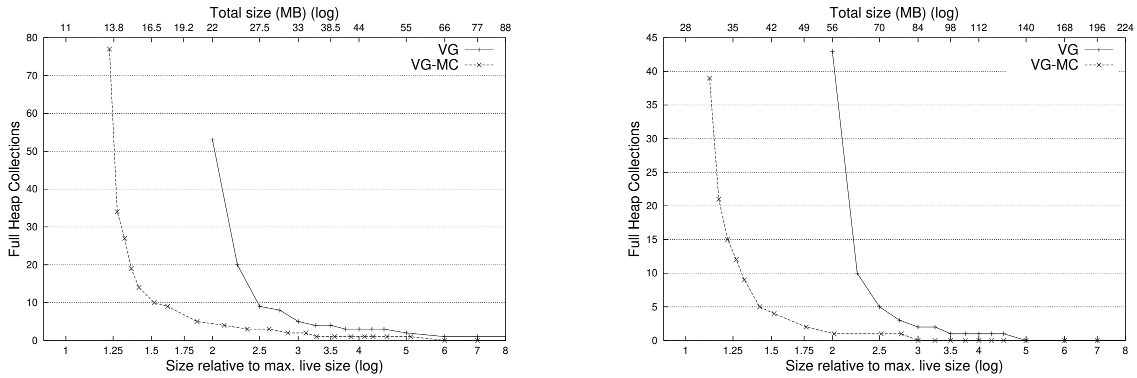
For all eight benchmarks, the MC collectors clearly run in much smaller space than VG. Since VG uses a large object space (which is managed by a mark-sweep collector), it can run in heaps slightly smaller than twice the maximum live size (it does not have to reserve any space for copying large objects). For all of the benchmarks, the VG collector does much worse than the MC collectors when the space available is around twice the maximum live size.

javac: When the space available is about 2.3 times the maximum live size, the GC time for VG is twice as high as that for VG-MC. In space that is about four times the maximum live size, the GC time for VG is 1.5 times higher. When the total space grows to about five times the live size, the GC time for VG is 15% higher than that for VG-MC. The GC time for VG-MC drops sharply at this point because it does not perform any full collections. The GC time for VG is 50–60% higher than that for VG-MC in space that is six times the maximum live size or larger.

The shapes of the curves for the relative execution times are similar. In space that is about 2.3 times the maximum live size, total execution time with VG is 40% higher than with VG-MC. VG consistently performs within 10% of VG-MC only in heaps larger than 4.5 times the live size. At eight times the live size, the total execution time for VG is 5% worse than that for VG-MC. The GC and total execution time curves for VG-MC and VG are very similar in shape to the curves for the copying costs, which shows that VG-MC obtains the performance improvement by reducing copying cost significantly.

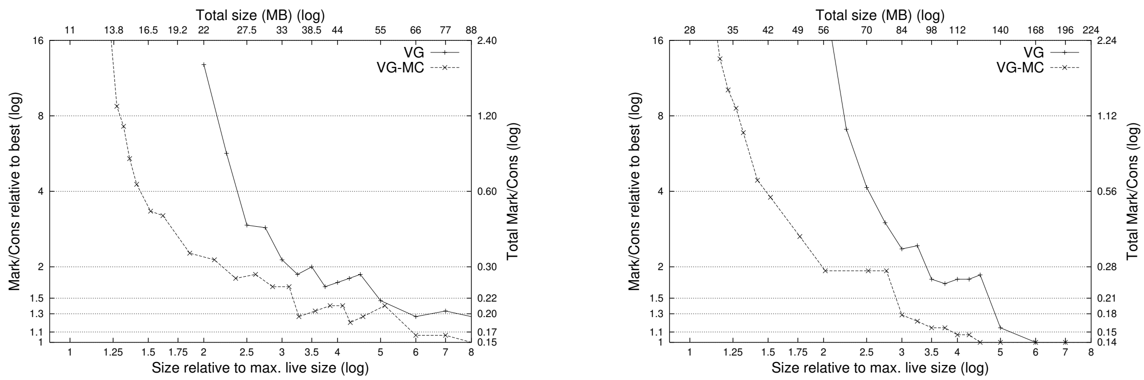
VG-MCHW, which uses an extra word per object in the old generation, has a slightly higher space overhead than VG-MC. In tight heaps, it is significantly slower than VG-MC. When the space available is larger than twice the live size, it performs at most 7% worse than VG-MC, and is typically within a few percent of VG-MC. It outperforms VG at most heap sizes.

db: This benchmark builds a large linked list, which it repeatedly traverses during execution. This leads to locality issues. VG-MC performs better than VG until the space grows to about three times the maximum live size. The performance of all collectors degrades in large heaps (possibly due to locality effects on TLB misses). VG-MCHW is significantly slower than VG-MC in small heaps and about 5–8% worse than VG and VG-MC in heaps that are 2.25



(a) javac (b) pseudojbb

Figure 7: Number of full heap collections performed for javac **and** pseudojbb



(a) javac (b) pseudojbb

Figure 8: Relative Mark/Cons ratio for javac **and** pseudojbb

times the live size or larger. This is possibly because of the effect the header word has on the cache.

jess, mpegaudio, mtrt, and jack: The results for these benchmarks are similar. VG starts off with total execution times about 25–50% worse than VG-MC. Its performance is typically within 10% of VG-MC only in heaps that are 2.5–3 times the live size or larger, and it performs almost as well as VG-MC at eight times the maximum live size. VG-MCHW generally performs within a few percent of VG-MC in heaps larger than twice the live size, and outperforms VG in heaps that are 3–4 times the live size or smaller.

pseudojbb: The total execution time for VG at 2.25 times the live size is 30% worse than that for VG-MC. VG comes within 10% of VG-MC in space that is 2.75 times the live size. VG performs almost as well as VG-MC in space that is five times the live size or larger. VG-MCHW is 5–20% slower than VG-MC in tight heaps. In heaps larger than twice the live size, its performance is about the same as VG-MC.

health: Here, the curve for VG-MC is noisy in the smaller heap sizes. This is because, for some of the small heap sizes, the remembered set overhead for VG-MC is very large (up to 14MB). This overhead causes configurations that use less heap space to use up a large amount of total space.

The execution time of VG converges much more quickly for health than for the other benchmarks. At about 2.25 times the live size, VG performs 5% better than VG-MC, and in heaps larger than three times the live size, VG performs as well as VG-MC. However, since the live size is quite large, the actual additional memory required by VG to perform as well as VG-MC is approximately 67MB. VG-MCHW performs about 3–5% worse than VG-MC and VG in heaps that are 2.5 times the live size or larger, again possibly due to locality effects caused by the additional word per object.

Summary: VG-MC and VG-MCHW are capable of running in much smaller heaps than VG. The minimum space overhead for VG-MC is between 1.12–1.25 times the maximum live size, and 5–10% higher for VG-MCHW. The space advantage is clear in the plots for pseudojbb and health. The minimum space required to run pseudojbb using VG is 25MB higher than the space required by MC. The additional space required by VG for health is 50MB.

VG-MC consistently performs better than VG in small and moderate size heaps. For VG, garbage collection is up to eight times slower in small heaps and is 1.2–2 times slower in moderate size heaps. Overall execution time is significantly higher in small heaps, and is typically 5–10% higher in moderate size heaps. For five of the eight benchmarks, VG-MC performs within 6% of the best execution time, in space that is slightly larger than twice the maximum live size. VG-MC achieves the improved performance by utilizing

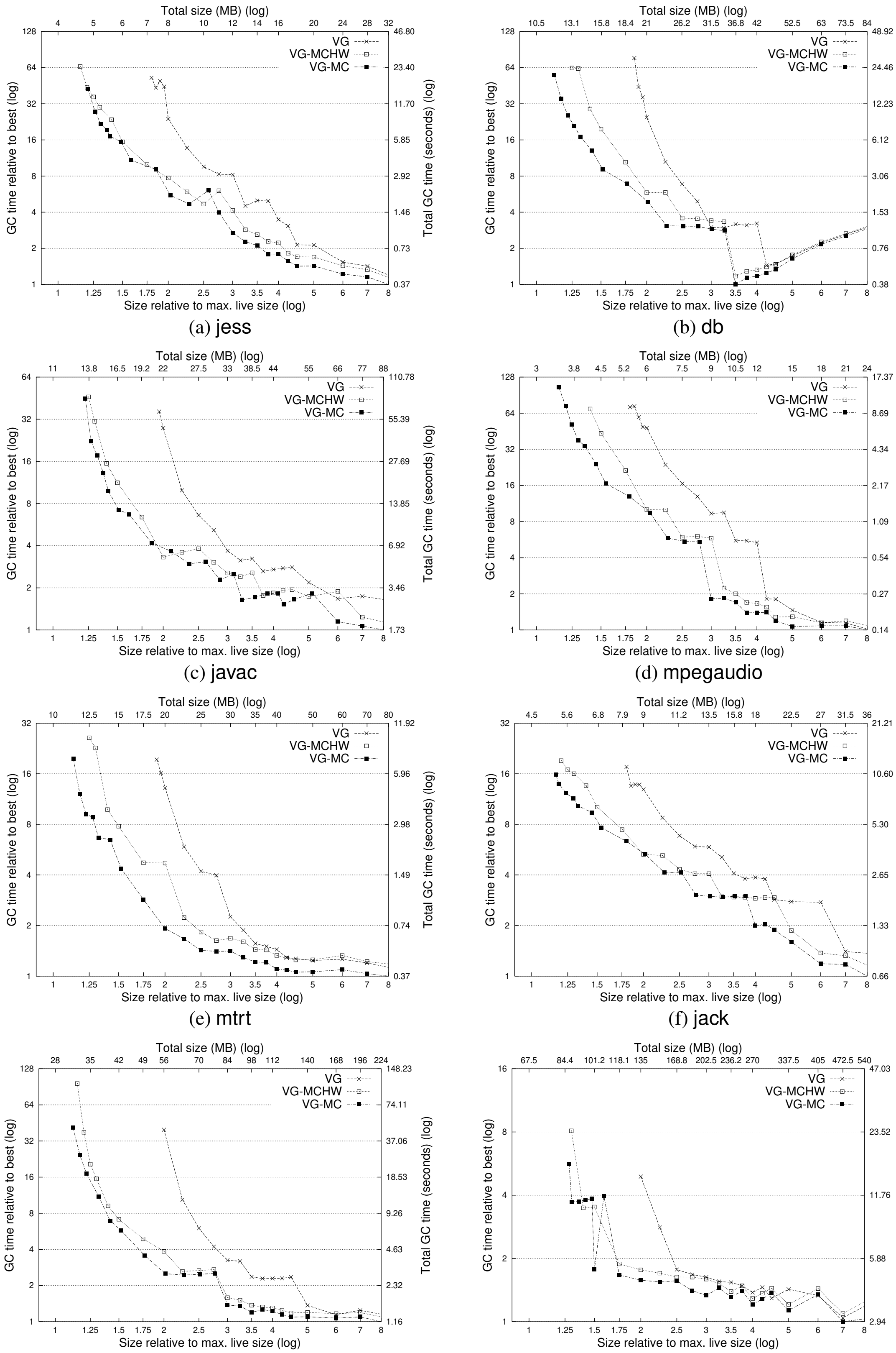
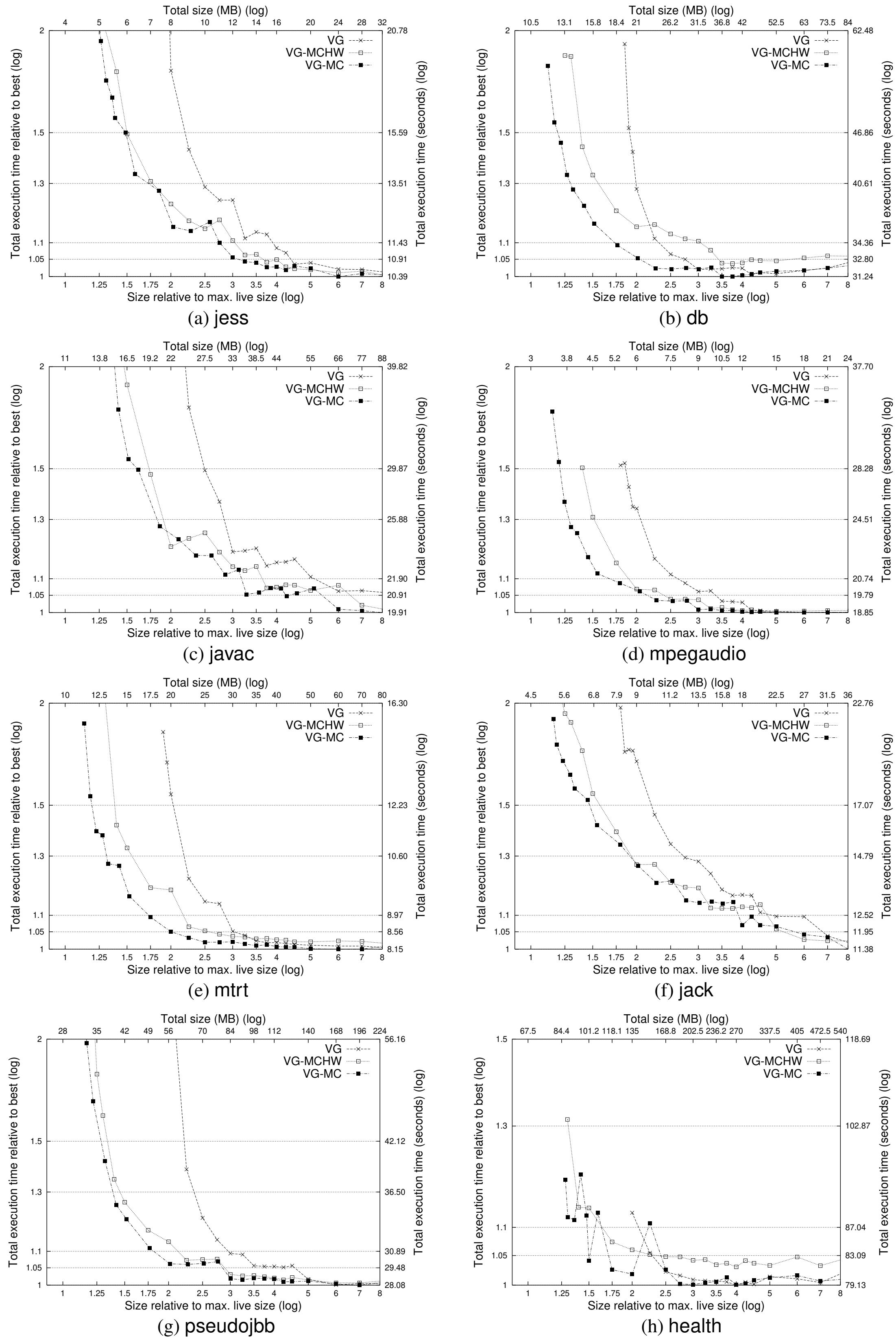


Figure 9: GC times for VG-MC, VG-MCHW, and VG relative to the best GC time



(a) jess (b) db

(c) javac (d) mpegaudio

(e) mtrt (f) jack

(g) pseudojbb (h) health

Figure 10: Total execution times for VG-MC, VG-MCHW, and VG relative to the best total execution time

the heap space better and performing less copying. Full collections for VG-MC typically take 20% more time (per byte copied) than full collections for VG. However, VG-MC performs full collections much less frequently, thus reducing the copying and GC overheads considerably in small and moderate size heaps.

The performance of VG-MCHW is usually significantly worse than VG-MC in tight heaps. In heaps larger than twice the maximum live size, it typically performs less than 10% worse, and is usually within a few percent of VG-MC. It does not outperform VG-MC for a couple of reasons. First, the remembered set overhead for the benchmarks we experimented with is smaller than the extra word overhead for VG-MCHW. Second, VG-MCHW suffers from some locality effects because of the extra word added to each object in the old generation. VG-MCHW typically outperforms VG in small and moderate size heaps. However, VG performs better than VG-MCHW for db and health because of locality effects.

We also measured the performance of BG-MC, using a nursery that occupied at most 20% of the heap space. However, we did not present the results in this section. We found that BG-MC performs quite well when compared with VG-MC. For all benchmarks, execution times with BG-MC were less than 10% worse than with VG-MC at all heap sizes, and usually within a few percent of VG-MC while providing a bound on the nursery pause times.

4.8 Mark-Copy vs. Mark-Sweep

We now relate MC to mark-sweep collection. We compare the performance of a (non-generational) mark-sweep collector (MS) and a hybrid copying/mark-sweep generational collector (VG-MS) with VG-MC. MS collectors can run in much smaller heaps than standard copying collectors. In the best case, to be able to run a program, MS requires space that is slightly more than the maximum live size for a program. However, MS usually suffers from some degree of fragmentation, and the amount of additional space required depends on the degree of fragmentation. MC, on the other hand, does not suffer from fragmentation. Like other copying collectors, it avoids fragmentation by regularly copying and compacting live data. However, MC does require space for copying and for remembered sets.

The MS collector we use for the comparison comes with the Jikes RVM 2.2.2 release. The collector uses segregated free lists to manage the heap memory, with 51 different size classes (a separate size class for every 4 bytes for the range $[8,63]$, every 8 bytes for the range $[64,127]$, every 16 bytes for range $[128,255]$, every 32 bytes for the range $[256,511]$, every 256 bytes for the range $[512,2047]$, and every 1024 bytes for the range $[2048,8192]$). The Jikes RVM MS collector divides the entire heap into a pool of blocks, each of which can be assigned to a free list for any of the size classes. An object is allocated in the free list for the smallest size class that can accommodate it. After garbage collection, if a block becomes empty, it is returned to the free block pool. All objects larger than 8KB are handled by a large object allocator, which rounds up object sizes to whole pages(4KB).

The VG-MS collector we use is a two generation collector that uses a variable size nursery. Nursery survivors are copied into the old generation, which is managed using the MS collector described above. VG-MS has a couple of advantages compared with MS. First, it exploits the generational hypothesis, filtering out short lived objects and performing much less allocation into MS space, because of which it triggers many fewer full collections than MS. Second, it uses bump pointer allocation in the nursery, which reduces allocation cost significantly.

Figure 11 shows GC times relative to the best GC time for VG-MC, VG-MS, and MS for the eight benchmarks. Figure 12 shows total execution times relative to the best execution time. The horizontal axis in all graphs represents total space and is drawn to a logarithmic scale. The vertical axis on the graphs represents relative times and is also drawn to a logarithmic scale. We now describe the performance of the three collectors for each of the eight benchmarks.

javac: In heap sizes between 1.4–3 times the live size, VG-MS performs slightly worse than VG-MC at a few points due to slightly higher mutator (program, as opposed to collector) times. VG-MS performs 2–5% slower than VG-MC in heaps larger than 3.25 times the live size, due to slightly higher GC times and mutator times. Mutator times for VG-MS are higher possibly because VG-MC compacts the old generation data and hence achieves somewhat better locality (both collectors use an identical write barrier). MS is significantly slower in heaps smaller than three times the live size, and it eventually performs as well as VG-MS.

db: In heaps larger than 1.5 times the live size, VG-MC is 5–10% faster than VG-MS. This happens even though the GC times for VG-MC and VG-MS are approximately the same, which suggests that VG-MC might be compacting and ordering objects in a manner that improves locality significantly, thus lowering mutator time. MS has the lowest GC time in heaps larger than five times the live size, but it performs 5–10% worse than VG-MC.

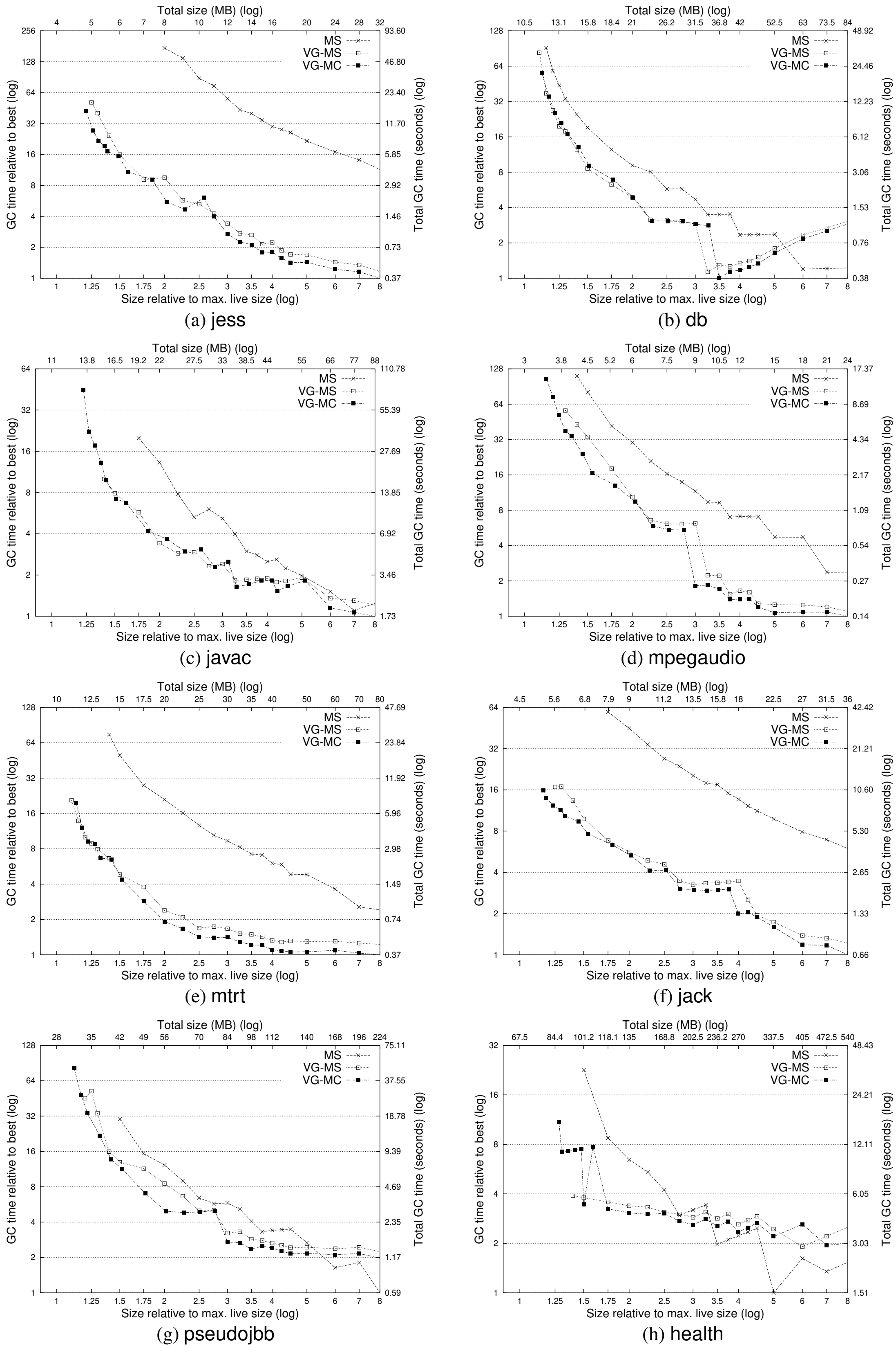
jess, jack, mpegaudio: Total execution times for VG-MS are significantly higher than that for VG-MC in heaps smaller than 1.5 times the live size, and about 1–3% higher than that for VG-MC in heaps larger than 2.5 times the maximum live size. For jess and jack, the performance of MS is significantly worse than the generational collectors at all heap sizes. For mpegaudio, the performance of MS is consistently within 5% of the generational collectors only in heaps larger than 3.5 times the live size.

mtrt: In heaps larger than 1.5 times the live size, VG-MS is 5% slower than VG-MC due to higher GC times and mutator times. MS performs significantly worse than both generational collectors at all heap sizes.

pseudojbb: VG-MS performs 3–5% worse than VG-MC at most heap sizes, again due to a combination of higher GC and mutator times. The performance of MS is worse than the generational collectors at all heap sizes. Interestingly, MS has the lowest GC time in heaps larger than five times the live size. However, it is 6–8% slower than VG-MC due to higher mutator times.

health: VG-MC suffers from large remembered set overheads in small heaps making the curve noisy. It performs 5–10% worse than VG-MS in some small heaps. However, in heaps larger than 2.5 times the live size, VG-MC performs 5% better than VG-MS. MS has the lowest GC time of all three collectors in heaps larger than 4.5 times the live size, but, it has the highest execution times.

Summary: VG-MC can run in space that is comparable to the minimum required by VG-MS. MS usually has a higher fragmentation overhead than VG-MS and thus requires somewhat larger heaps to be able to run successfully. Both VG-MC and VG-MS run significantly faster than MS for most benchmarks in small and moderate size heaps, affirming the generational hypothesis. The GC times for VG-MC are slightly lower or about the same as those for VG-MS. However, for javac, mtrt, pseudojbb, and health the total execution times for VG-MS are up to 5% higher than those for VG-MC. For db, execution times are 5–10% higher. Since both collectors use an identical write barrier, our guess is that this performance improvement happens because data in the old generation is compacted by VG-MC, because of which it achieves better cache locality and/or TLB performance. For health, VG-MC suffers from somewhat large remembered sets, which makes its performance worse than that of VG-MS in small heaps.



(a) jess (b) db (c) javac (d) mpegaudio (e) mtrt (f) jack (g) pseudojbb (h) health

Figure 11: GC times for VG-MC, VG-MS, and MS relative to the best GC time

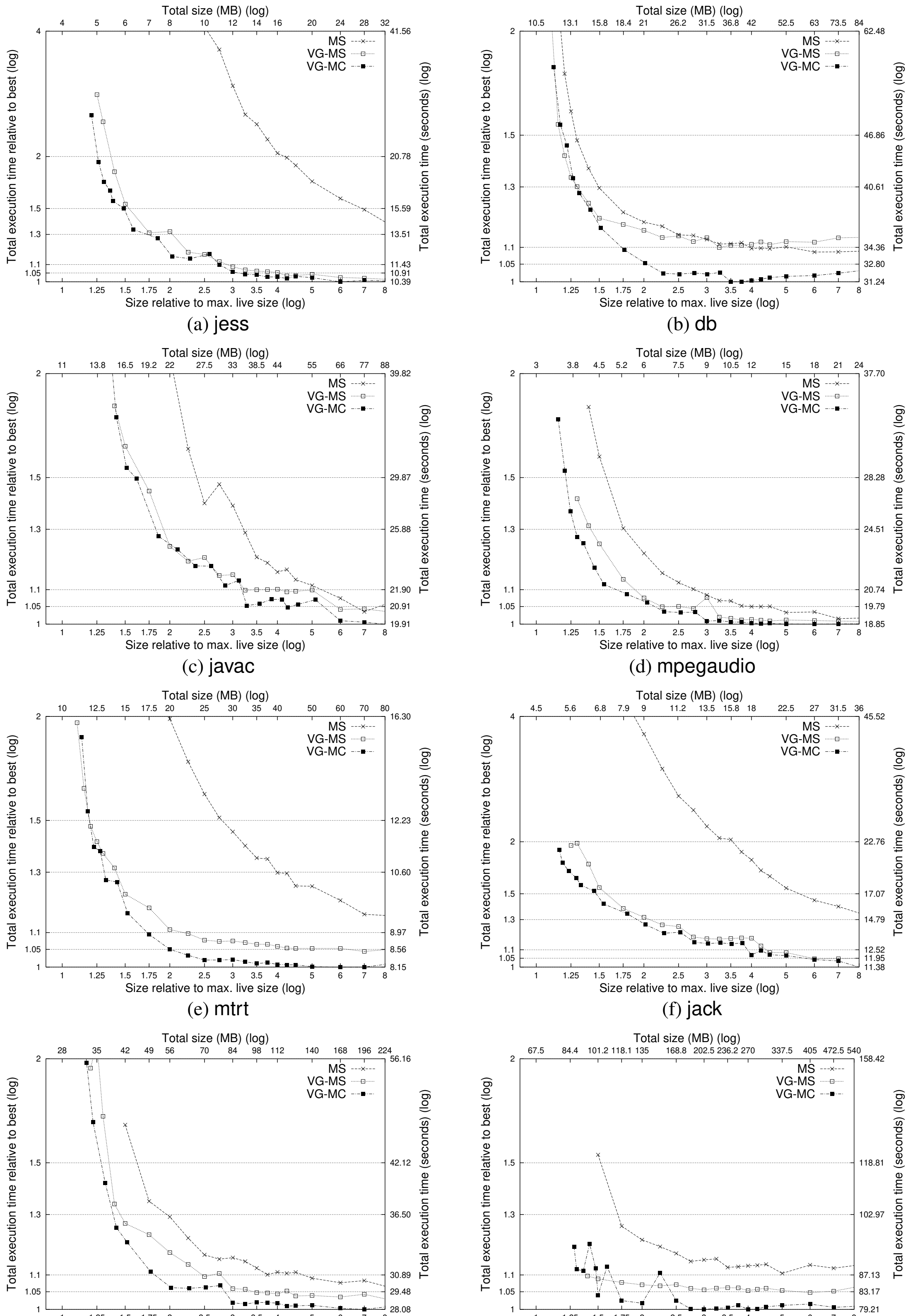
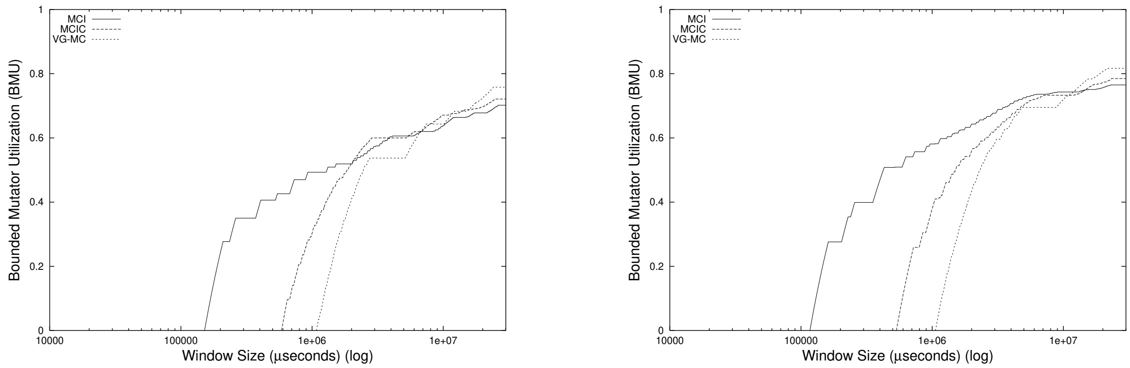


Figure 12: Total execution times for VG-MC, VG-MS, and MS relative to the best total execution time



(a) javac 22MB (b) javac 33MB

Figure 13: BMU curves for javac

5. PAUSE TIME

We now look at the pause time characteristics of MC. We consider more than just the maximum pause times that occurred, since these do not indicate how the collection pauses are distributed over the running of the program. For example, a collector might cause a series of short pauses whose effect is similar to a long pause, which cannot be detected by looking only at the maximum pause time of the collector. Instead, we look at the *mutator utilization* curves for the collectors, following the methodology of Cheng and Blelloch [7]. They define the *utilization* for any time window to be the fraction of the time that the mutator (the program, as opposed to the collector) executes within the window. The minimum utilization across all windows of the same size is called the *minimum mutator utilization* (MMU) for that window size. An interesting property of this definition is that a larger window can actually have *lower* utilization than a smaller one. To avoid this, we extend the definition of MMU to what we call the *bounded minimum mutator utilization* (BMU). The BMU for a given window size is the minimum mutator utilization for all windows of that size *or greater*.

Figure 13 shows BMU curves for javac for heap sizes equal to two and three times the maximum live size (22MB, 33MB). The x-intercept of the curves indicates the maximum pause time, and the asymptotic y-value indicates the fraction of the total time used for mutator execution (average utilization). The three curves in each graph are for VG-MC, MCIC, and a speculative fully incremental MC collector (MCI). Since it is difficult to factor out the write barrier cost, the VG-MC curve actually represents utilization inclusive of the write barrier. The real mutator utilization will be a little lower. For MCIC, however, we factor in the additional cost of the complex write barrier by apportioning the additional mutator time (difference between mutator time for MCIC and VG-MC) to the GC time. This does assume that the write barrier overheads are spread uniformly over the execution.

To predict the performance of MCI, we take each mark pause for MCIC and distribute it equally among the copy pauses prior to the mark. We charge the mark an additional 20% before we distribute it. A real collector with an incremental mark would probably do worse. This is because the amount of live data found by a collector with an incremental mark is usually larger than the amount found by a collector with a non-incremental mark. However, the curve does give an idea of the expected collector performance.

VG-MC has the highest y-intercept value, since it has the best overall mutator utilization, but it also has the largest x-intercept value since it performs non-incremental marking and copying. The slope of the curve is also steep, showing that for windows not much larger than the maximum pause time, the utilization is high. The x-intercept for MCIC is much smaller, because of the incrementality of copying, but this incrementality also lowers the overall throughput (asymptotic y-value). MCIC is about 5% slower than VG-MC. The speculative MCI collector has a maximum pause time of around 150ms. It allows a minimum utilization of about 25% for a window slightly larger than the maximum pause time. The overall utilization (asymptotic y-value) for the MCI collector is only slightly lower than that for MCIC. In practice, we expect the difference to be somewhat larger. These results show that the MC collector can be made completely incremental while still obtaining good throughput. With more tuning we expect to get lower pauses than the values we show here.

6. OTHER RELATED WORK

In the introduction we discussed copying collectors most directly related to our work. We now consider other similar copying techniques.

Lang and Dupont [11] describe a collector that performs incremental compaction in a heap managed by a mark-sweep collector. Ben-Yitzhak et al. [4] propose a collector similar to Lang and Dupont's, but that is also parallel. Lang and Dupont divide the heap into equal size segments. At each collection, the collector marks the entire heap, and then compacts one segment. Thus the collector is primarily mark-sweep, but the compaction helps in overcoming fragmentation. MC is primarily a copying collector, and its incremental copying (whether batched or distributed over time) reduces copy reserve overhead. For a heap consisting of n segments, the Lang and Dupont collector performs n full heap marks in order to compact each segment once. MC occasionally performs *one* full heap mark, but then compacts the entire heap with no further marking. Further, MC can compact multiple segments in one pass, thus minimizing the number of passes required to compact the heap. MC does have a remembered set overhead that the Lang and Dupont collector does not have.

The MCI collector is similar to the Mature Object Space (MOS, or Train) collector [8]. MOS divides the old generation into equal sized windows called *cars*. In MOS, cars bound the amount of copying performed in each collection. The difference between MOS and MCI is in the manner in which they provide completeness. For MCI, the full heap mark ensures completeness. MOS groups cars into logical units called *trains*, and performs copying in a manner that moves any large cycle of garbage into a single train, which can then be reclaimed in its entirety. Our experiments with MOS (not detailed here) indicate that it tends to copy a large amount of unreachable data before the data is moved to a separate train and reclaimed. We expect that the full heap mark performed by MCI operates with significantly reduced overhead. Detailed comparison of MCI with MOS is beyond the scope of this paper.

One might also compare MC with mark-sweep-compact (MSC), using compaction of the entire old generation. This differs from the Lang and Dupont collector in that it compacts the entire heap, and differs from MC in that it uses sliding compaction rather than copying. MSC has a space advantage over MC, namely the one window copy reserve space plus the old generation remembered sets of MC. However, MSC is more difficult to implement, and inherently makes more passes over the compacted objects. Thus, we expect MSC to run substantially slower than MC, even given its (modest) space advantage.

7. CONCLUSIONS

We have presented a new style of copying collector, MC, that significantly reduces space requirements compared to standard copying collectors. We showed that MC collectors can run in space that is as small as 1.12–1.25 times the maximum live size of a program, while standard copying collectors usually require two times the maximum live size. This increases the range of applications that can use copying garbage collection. We have shown that MC collectors outperform generational copying collectors in equal sized heaps; they do so by significantly reducing the amount of copying. Overall performance with MC is significantly better in tight heaps, and is typically around 5–10% better in moderate size heaps. We compared MC against a mark-sweep (MS) collector and a hybrid copying/mark-sweep generational collector (VG-MS); we showed that MC can run in heaps that are comparable in size to the minimum for MS and VG-MS. For most benchmarks, MC is significantly faster than MS in small and moderate size heaps. When compared with VG-MS, MC improves total execution time by about 5% for some benchmarks. The improvements are caused both by lower GC times and by better locality. We also described partially and fully incremental versions of MC, which better bound pause time and offer improved mutator utilization for shorter time scales.

8. ACKNOWLEDGMENTS

We thank Rick Hudson for the initial idea that sparked the work for this paper. We also thank IBM Research for providing us with Jikes RVM, and Steve Blackburn and Perry Cheng for writing the JMTk garbage collection toolkit. Thanks to Kathryn McKinley and Emery Berger for helpful discussions and comments, and to Chris Hoffmann for providing us with the pseudo-adaptive system in Jikes RVM. We thank the anonymous reviewers for their many suggestions for improving the paper.

9. REFERENCES

[1] Bowen Alpern, C. R. Attanasio, Anthony Cocchi, Derek Lieber, Stephen Smith, Ton Ngo, John J. Barton, Susan Flynn Hummel, Janice C. Sheperd, and Mark Mergen. Implementing Jalapeño in Java. In OOPSLA '99 [13], pages 314–324.

[2] Bowen Alpern, Dick Attanasio, John J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, Anthony Cocchi, Stephen J. Fink, David Grove, Michael Hind, Susan Flynn Hummel, D. Lieber, V. Litvinov, Mark Mergen, Ton Ngo, J. R. Russell, Vivek Sarkar, Manuel J. Serrano, Janice Shepherd, S. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM System Journal*, 39(1), February 2000.

[3] Andrew W. Appel. Simple generational garbage collection and fast allocation. *Software Practice and Experience*, 19(2):171–183, 1989.

[4] Ori Ben-Yitzhak, Irit Goft, Elliot Kolodner, Kean Kuiper, and Victor Leikehman. An algorithm for parallel incremental compaction. In ISMM '02 [10], pages 100–105.

[5] Stephen M. Blackburn, Richard Jones, Kathryn S. McKinley, and J. Eliot B. Moss. Beltway: Getting around garbage collection gridlock. In *Proceedings of SIGPLAN 2002 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, pages 153–164, Berlin, June 2002. ACM Press.

[6] Stephen M. Blackburn and Kathryn S. McKinley. In or out? Putting write barriers in their place. In ISMM '02 [10], pages 175–184.

[7] Perry Cheng and Guy Blelloch. A parallel, real-time garbage collector. In *Proceedings of SIGPLAN 2001 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, pages 125–136, Snowbird, Utah, June 2001. ACM Press.

[8] Richard L. Hudson and J. Eliot B. Moss. Incremental garbage collection for mature objects. In Yves Bekkers and Jacques Cohen, editors, *Proceedings of International Workshop on Memory Management*, volume 637 of *Lecture Notes in Computer Science*, pages 388–403, University of Massachusetts, USA, 16–18 September 1992. Springer-Verlag.

[9] Richard L. Hudson, J. Eliot B. Moss, Amer Diwan, and Christopher F. Weight. A language-independent garbage collector toolkit. Technical Report COINS 91-47, University of Massachusetts at Amherst, Department of Computer and Information Science, September 1991.

[10] *ISMM'02 Proceedings of the Third International Symposium on Memory Management*, ACM SIGPLAN Notices, Berlin, June 2002. ACM Press.

[11] Bernard Lang and Francis Dupont. Incremental incrementally compacting garbage collection. In *SIGPLAN'87 Symposium on Interpreters and Interpretive Techniques*, volume 22(7) of *ACM SIGPLAN Notices*, pages 253–263. ACM Press, 1987.

[12] Henry Lieberman and Carl E. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, 1983. Also report TM–184, Laboratory for Computer Science, MIT, Cambridge, MA, July 1980 and AI Lab Memo 569, 1981.

[13] *OOPSLA'99 ACM Conference on Object-Oriented Systems, Languages, and Applications*, volume 34(10) of *ACM SIGPLAN Notices*, Denver, CO, October 1999. ACM Press.

[14] Anne Rogers, Martin C. Carlisle, John H. Reppy, and Laurie J. Hendren. Supporting dynamic data structures on distributed-memory machines. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 17(2):233–263, March 1995.

[15] Standard Performance Evaluation Corporation. *SPECjvm98 Documentation*, release 1.03 edition, March 1999.

[16] Standard Performance Evaluation Corporation. *SPECjbb2000 (Java Business Benchmark) Documentation*, release 1.01 edition, 2001.

[17] Darko Stefanović. *Properties of Age-Based Automatic Memory Reclamation Algorithms*. Ph.d. thesis, University of Massachusetts, Department of Computer Science, Amherst, MA, 1999.

[18] Darko Stefanović, Matthew Hertz, Stephen M. Blackburn, Kathryn S. McKinley, and J. Eliot B. Moss. Older-first garbage collection in practice: Evaluation in a Java virtual machine. In *Proceedings of the ACM SIGPLAN Workshop on Memory System Performance (MSP 2002)*, Berlin, Germany, June 2002.

[19] Darko Stefanović, Kathryn S. McKinley, and J. Eliot B. Moss. Age-based garbage collection. In OOPSLA '99 [13], pages 370–381.

[20] David M. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices*, 19(5):157–167, April 1984. Also published as ACM Software Engineering Notes 9, 3 (May 1984) — Proceedings of the ACM/SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, 157–167, April 1984.