Reducing Generational Copy Reserve Overhead with Fallback Compaction

Phil McGachey Antony L. Hosking

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907, USA
phil@cs.purdue.edu hosking@cs.purdue.edu

Abstract

As programming languages with managed runtimes become increasingly popular, it is essential that virtual machines are implemented efficiently. The performance of the memory management subsystem can be a defining factor in the performance of the virtual machine as a whole. We present a technique by which garbage collector performance can be improved.

We describe an algorithm that combines a standard generational copying collector with a mark and compact collector. We observe that, since most objects do not survive a garbage collection, it is not necessary to reserve space to copy them all. The result is a generational copying collector that operates with a smaller copy reserve overhead than traditional Appel-style collectors. We maintain correctness in the worst case through the use of mark and compact collection. When the reduced copy reserve overflows, a compacting phase ensures that all data are accommodated.

We have implemented this algorithm within the framework of Jikes RVM and MMTk. For most benchmarks examined, our experiments show that performance is comparable to or better than a standard generational copying collector.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—memory management (garbage collection)

General Terms Algorithms, Design, Performance, Experimentation

Keywords Java, garbage collection, generational collector, copying collector, mark and compact

1. Introduction

With the current popularity of VM-based languages such as Java or C# comes the need for high-performance virtual machines. A vital component of any such system is the garbage collector; its efficiency can determine the speed of the system as a whole. This paper presents a technique through which garbage collection performance can be improved.

The advantages and disadvantages of many popular uniprocessor garbage collection techniques have been widely discussed [1]. Generational copying collectors have been shown to improve cache performance through spatial locality, but suffer the space overhead of a copy reserve. Generational mark and sweep collectors are able to use all available space, but can lead to a fragmented heap. Compacting collectors provide both spatial locality and good heap utilization, but require multiple sweeps of the heap, leading to poor performance.

The algorithm outlined in this paper combines the positive features of generational copying and compacting collectors. It behaves as a generational copying collector, but uses a far smaller copy reserve. When that copy reserve is found to be insufficient, it falls back on a compacting collector to copy all remaining objects. Since the compacting collector is rarely used, the performance penalty related to compaction is seldom encountered. However, the reduced copy reserve allows garbage collections to occur less frequently, leading to less time spent in garbage collection, and higher overall throughput.

All garbage collection algorithms must account for the worst case in order to maintain correctness. The worst case behavior of a copying garbage collector occurs when all objects survive a collection. In this case, every object must be copied from the old space to the new, leading to long collection pauses. More significant, however, is the space required by those objects. Space must be reserved to receive these objects should the worst case arise.

In practice, however, the worst case virtually never occurs. Indeed, the survival rate of objects in the nursery of a generational collector is very low – we observe it to be frequently below 10%. As a result, 90% of the space allocated as a copy reserve is wasted. Clearly this is unsatisfactory; less space available to the allocator means that fewer objects can be allocated between collections. As a consequence, the space wasted by the copy reserve leads to more frequent collections, using processor time that could be assigned to the mutator. Additionally, these frequent collections give objects less time to die, meaning that short-lived objects are copied into the mature space. Not only does this add to the copying overhead, but it fills the mature space faster, leading to more frequent major collections.

The algorithm we propose eliminates the need for this wasted space. Since the vast majority of collections do not require the full copy reserve, we provide only what is needed in the common case. In doing so, we reduce the collection frequency, and improve performance.

However, the worst case behavior remains, and must be accounted for. In our system, we do not aim for high performance in the worst case - we simply require correctness. As such, when our estimate of the required copy reserve is too low, and the copied objects overflow, we switch to a compacting collector. This allows us to rearrange the objects in their current space, without allocating any further memory. We aim to set aside sufficient space for the copy reserve as to make expensive compacting collections rare. Indeed, in the vast majority of cases it is not necessary to fall back on the compacting collector at all, meaning that the only difference between our algorithm and a traditional generational copying collector is that of space utilization.

The remainder of this paper is structured as follows: Section 2 provides an overview of the work upon which we build. Section 3 details the design of our collection algorithm, while Section 4 discusses closely related work. Section 5 outlines some of the more interesting points of our implementation, for which experimental results are given in Section 6. Finally, Section 7 concludes.

2. Background

The algorithm presented in this paper combines a generational copying collector with a mark compact collector. This section outlines the background upon which this work builds.

2.1 Generational Collectors

Generational collectors are a class of garbage collector that exploit the age of objects to improve performance [2]. They are based on the *generational hypotheses*: the *weak generational hypothesis* states that most objects die young [3], while the *strong generational hypothesis* states that the older an object is, the less likely it is to die.

The heap managed by a generational collector is divided into two or more spaces, or generations [4]. The simplest generational collectors have a small nursery where objects are allocated, and a larger mature space where they are promoted after surviving a collection.

The nursery space is generally allocated to by a simple bump-pointer. Since all allocation is to the nursery, this simplifies the allocation process. Once an object has survived a nursery collection, it is promoted to the mature space. Collection within the mature space can be performed using a different algorithm from the nursery. For example, a generational mark sweep collector allocates objects in the nursery using a bump pointer, but when they are copied to the mature space uses a free list allocator. The mature space is then managed by a mark and sweep collector.

The main benefit of splitting the heap into generations is that it is no longer necessary to collect the entire heap in a single operation. By collecting only the nursery space, garbage collection pauses can be greatly decreased. However since the majority of objects in the nursery are garbage (due to the weak generational hypothesis), this minor collection also frees more space than a collection of an equivalent-sized region of a non-segregated heap.

2.2 Appel's Generational Copying Collector

Appel's generational copying collector uses a variable-sized nursery and a single mature generation [5]. The mature space is managed by a semi-space copying collector. New objects are allocated to the nursery using a bump-pointer allocator. When the nursery is full, a minor collection is triggered in which live objects are copied to the mature space. When the mature space fills, space is reclaimed by a major copying collection.

The size of the nursery varies depending on the occupancy of the mature space. Initially, when there are no objects in the mature space, the nursery takes up half of the heap. This is all the space available, after taking into account the copy reserve.

When it is detected that a minor collection will cause the nursery to shrink below a predetermined threshold, a major collection, which includes the mature space, is triggered. During a major collection, all reachable objects in either the nursery or mature from-space are copied into the mature to-space.

2.3 Copy Reserves

The major drawback to Appel's algorithm, as with most copying collectors, is the overhead of the copy reserve. In the worst case, it is possible for all objects to survive a garbage collection. In order to accommodate this eventuality, sufficient space must be set aside to copy all objects. This space is referred to as the copy reserve. In the case of a simple semi-space collector the copy reserve accounts for half the total heap size. In general, the size of the copy reserve must be equal to the size of the space being collected in case all objects survive.

The copy reserve reduces the effective size of the heap. As a result, the garbage collector must be triggered more frequently; since objects cannot be allocated to the copy reserve, fewer objects can be allocated before available memory is exhausted. A large copy reserve means that less space is available for allocation, and when the available memory is decreased, the collector must run more frequently.

2.4 Mark and Compact Collectors

Mark and compact collectors aim to gain the memory layout advantages of a copying collector while eliminating the need for a copy reserve. In a mark and compact collector, all live objects are marked by a tracing phase [6]. All marked objects are then relocated towards the "start" of the heap (where the start of the heap is defined as low memory addresses, and the end is high memory addresses). In this way, garbage objects are overwritten and live objects retained.

Since objects are not moved from one space to another, a copy reserve is not necessary. In the worst case, where all objects survive, a mark and compact collector simply does not perform any compaction. As a result, the whole heap can be used by the application without the overhead of the copy reserve. Also, while the copying overhead remains, it is likely to be present to a lesser degree. Long-lived objects will cluster towards the start of the space, having been moved there in previous collections. As a result, portions of the heap may not need to move in some collections.

While the design of a compacting collector would appear to be optimal in terms of space usage, the implementation offers some difficulties. Since objects move inside the heap, it is necessary to maintain forwarding pointers, as in a copying collector. However this proves to be more of a challenge in the compacting case.

In a copying collector, the address to which an object has been moved can be stored in its old location. This way, any subsequent references to that object can be updated. This is possible in a copying collector because the old space is guaranteed not to be discarded before the collection completes. This is not the case in a compacting collector. It is possible for a forwarded object to be overwritten before all references to it have been updated.

Updating references in a compacting collector without having to maintain external data structures has been the focus of some research [7, 8]. Generally these algorithms require multiple additional passes over the heap, leading to longer pause times which make compacting collectors impractical.

3. Design

This section presents the design of the hybrid generational copying/compacting collector.

3.1 Algorithm Overview

The key observation underlining this work is that the full copy reserve in an Appel-style collector is very rarely required. Indeed, it has been observed in that the majority of cases, only a tiny percentage of the copy reserve is used [9]. We introduce a method of reducing this space overhead substantially.

Rather than allocating half of the available heap to the copy reserve, we instead set aside a smaller percentage. We refer to the copy reserve size as the percentage of the maximum copy reserve required, so a 10% copy reserve would actually occupy 5% of the total heap.

In the vast majority of cases, a well-chosen copy reserve size will accommodate all surviving objects. However, in all algorithms it is necessary to account for worst-case performance. In this instance, the worst case is where the survivors during a collection require more space than the allocated copy reserve provides. This is possible because the space from which objects are being copied can be larger than the copy reserve.

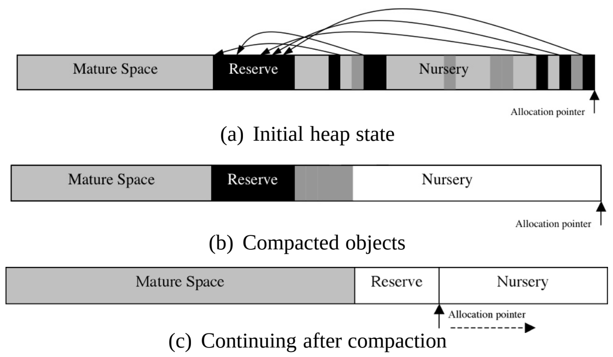
In this case, a compacting collection is triggered. Rather than copying survivors from the nursery to the mature space, or from the mature from-space to the mature to-space in the case of a major collection, objects are instead compacted, and then moved en-masse. This is performed without allocating further pages of memory. This fallback mechanism provides correctness in face of worst-case behavior.

3.2 Fallback Technique

Should the copy reserve prove to be insufficient during a collection, a compaction algorithm is activated. The collection can switch from copying to compacting without having to restart. This is because both copying and compacting collectors require a trace of the live objects; in the copying case this is combined with copying, while in the compacting case it is a separate phase.

3.2.1 Switching Collectors

When it is observed that no space remains in the copy reserve, the tracing routine switches from copying objects to simply marking them. This way, the trace can continue uninterrupted. When the trace is complete, an object can be in one of three states: unreachable (and hence garbage), copied or marked. This situation is illustrated in figure 1(a). Here the black objects have been evacuated into the copy reserve, filling it. The dark gray objects have subsequently been marked, and the light gray objects are unreachable.



(a) Initial heap state

(b) Compacted objects

(c) Continuing after compaction

Figure 1. Compacting collection

The compaction algorithm involves several linear sweeps of the heap, as described in Section 3.2.2. During the first of these scans, all references to copied objects are updated. This eliminates the need for copied objects to be retained for their forwarding data. As a result, the only important data remaining in the nursery is contained inside marked objects.

The remaining live objects in the nursery are then compacted to the front of the nursery, as shown in Figure 1(b). The white space in the nursery now contains no live objects, and so can be reclaimed. The copy reserve and compacted nursery objects then become part of the mature space, and the free space is divided between a new nursery and a new copy reserve, as shown in Figure 1(c).

The situation where a compaction is required in the mature space is analogous.

3.2.2 Compaction Algorithm

Since the compaction phase occurs only when no further copy reserve space remains, it is vital that no allocation occur until space can be made available. This means that the compaction algorithm itself must not perform any allocation. The compacting algorithm is designed to operate only over memory already assigned to the process, ensuring that the maximum heap size is never violated.

The key difficulty in designing a non-allocating compacting collector is that of forwarding pointers. In a copying collector, it is possible to overwrite copied objects with forwarding pointers, since the contents already exist in the copy reserve. However this is not possible with a compacting collector. If a forwarding pointer were to be installed in an object which was then itself later overwritten during compaction, subsequent reads of that pointer would be invalid. Forwarding pointers could be maintained in a separate data structure, but such a solution would require allocation. Alternatively, an additional header word could be added to each object to store forwarding pointers. However, since the compacting collector is used infrequently in this algorithm, it would be wasteful to add extra space overhead to every object.

Jonkers proposed a technique for managing forwarding pointers during a compacting collection [10]. This algorithm has the advantage over others in that it tracks forwarding pointers without allocating additional storage. It is performed through the technique of reference chaining.

The insight behind reference chaining is that in order to properly update pointers it is necessary to record either the object being moved or the references to that object. Traditional approaches track the former, and references are updated in a scan of the heap. The chaining algorithm instead associates references to an object with that object itself, and updates pointers as soon as the destination of the object is known. This solves the problem of managing forwarded pointers, and does not require allocation.

In order to identify objects with chained references, the use of one word per object is required by the algorithm. The location of this word is used to point to the head of the reference chain from that object. The contents of the word is moved to the referring object, and replaced upon completion of the algorithm, with the exception of one bit that indicates whether the current contents of the word refers to a reference chain. Thus, the only space overhead of the algorithm is that one bit.

In order to identify objects with chained references, the compaction algorithm requires two complete sweeps of the heap: one to update forward references, the other to update backward references, and to compact objects. While linear sweeping is more efficient than tracing (due to improved spatial locality), these additional operations make compacting collections far more expensive than standard copying collections. This is particularly true during minor collections, where a full trace of the heap is not normally required.

4. Related Work

There are several pieces of work that use a reduced copy reserve in order to improve generational copying collector performance. Additionally, some research has been performed in dynamically switching between multiple collection techniques in order to exploit the best properties of each.

4.1 Reduced Copy Reserve

The parallel copying collector implemented in Sun Microsystem's Hotspot virtual machine [11] makes use of a similar technique to minimize copy reserve overhead. However, Hotspot uses a fixed-size nursery, rather than an Appel-style variable-sized nursery. We believe that the variable-sized nursery reduces the space wasted in workloads with a low nursery survival rate.

Velasco *et al* make use of the same observation as this work [12]. They determine that the survival rate of collections is far below the space normally allocated to the copy reserve in an Appel style collector.

Their work differs from ours in several ways. They adopt a strategy of dynamically tuning the nursery copy reserve size, while we set the copy reserve as constant during execution. They suggest several simple heuristics to determine the optimal copy reserve size. In each case, the prior history of the survival rate is combined with a "security margin" at each collection to determine the space available for the next series of allocations. A danger of this approach is that it may lead to over-estimation of the copy reserve requirement, as a single unusually high survival rate can poison the heuristic.

Additionally, the work described by Velasco *et al* reduces only the nursery copy reserve. This misses the opportunity to utilize the memory used by the mature space. While the nursery space is larger shortly after a full collection, as the mature space fills up the amount of space required as a copy reserve similarly increases. Shortly before a full garbage collection, the mature space dominates the available memory in the heap, meaning that optimizing only the nursery will have little effect.

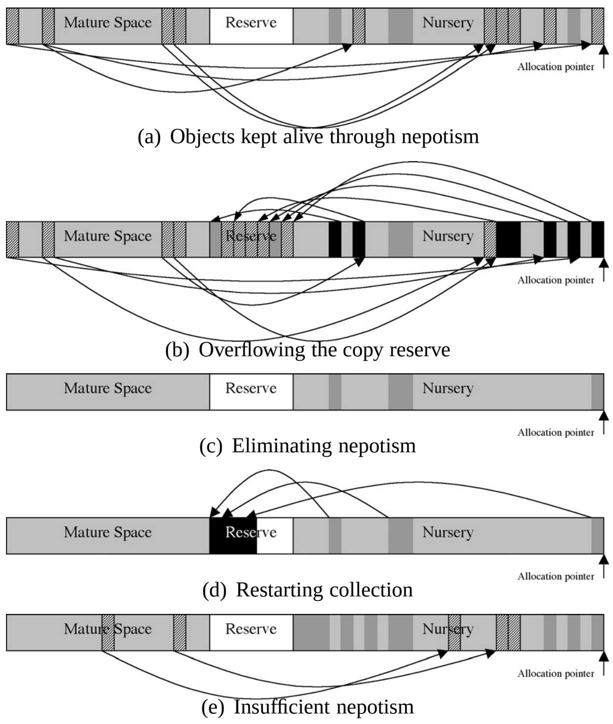
The most important difference, however, lies in the recovery strategy in the case of a missed prediction. While the design outlined in this paper performs a compacting collection, the earlier work instead relies on the principle of nepotism. This refers to garbage objects in the nursery being kept reachable by garbage objects in the mature space. In Figure 2(a), several objects (outlined in black) in the mature space are no longer live. However, since the mature space is not traced during a minor collection, they keep some objects in the nursery, which are not reachable from the program roots, alive through nepotism. Velasco *et al* rely on the fact that by performing a full collection they will not only free up memory from the mature space but also reduce the survival rate of the nursery by eliminating these redundant links.

Figure 2(b) shows this strategy. A minor collection has been performed, and the live objects in the nursery are found to require more space than the copy reserve can supply. In Figure 2(c), the unreachable objects in the mature space have been removed, and so the objects kept alive through nepotism are no longer seen as reachable. Figure 2(d) shows the successful completion of the collection, since without the additional objects retained through nepotism, all live objects fit in the copy reserve.

This argument is flawed, as can be seen in Figure 2(e). Even if the objects retained through nepotism were removed, there would still be insufficient space in the copy reserve. In the worst case, all objects could survive a collection. In order to maintain correctness, an algorithm must be able to perform a collection under these circumstances; it is for this reason that the 100% copy reserve was required in the original generational copying collector design. In the case where most or all objects survive, the algorithm outlined by Velasco *et al* would fail.

The solution proposed in this paper is able to handle the worst case safely. Should all objects survive, those that fit in the copy reserve will be moved. The remaining objects will be compacted inside the nursery, requiring no additional space.

The MC^2 collector produced by Sachindran *et al* [13] divides the mature space of a generational collector into frames, each of which can be collected separately. The result of this is that the copy reserve required by the collector is limited by the frame size; only one frame is needed, since in the worst case the data copied is a complete frame.



(a) Objects kept alive through nepotism
(b) Overflowing the copy reserve
(c) Eliminating nepotism
(d) Restarting collection
(e) Insufficient nepotism

Figure 2. Reducing survival rate through nepotism

The MC^2 collector is primarily aimed at memory-constrained situations, such as embedded processors, cell phones or PDAs. The work described in this paper does not necessarily aim to reduce the minimum heap size necessary to run programs (although such an improvement is often observed), rather to increase total throughput by making better use of the memory available. As will be shown in Section 6, the copying/compacting algorithm performs best with larger heap sizes, where the benefits of its space management make the most impact.

Additionally, MC^2 uses an incremental approach to the marking phase. This technique interleaves small amounts of marking during allocation, rather than performing a single monolithic marking trace over the heap. While this strategy reduces pause times (since less work is required during a collection), it increases the complexity of the allocation sequence, and requires a mechanism to fall back upon in case the incremental marking is unable to keep up with the allocation rate. Since the generational copying/compacting collector does not aim to make real-time guarantees, we chose to emphasize simplicity over pause times.

Indeed, the primary advantage of the copying/compacting collector over MC^2 is its simplicity. In the vast majority of cases, the collector behaves as a more efficient version of Appel's original generational copying collector. On the rare occasions that the copying collector runs out of space, it falls back on a mark and compact collector. This hybrid approach takes advantage of two well-understood algorithms, making implementation less complex.

4.2 Dynamic Collector Switching

A major part of our algorithm is the ability to switch seamlessly between generational copying and compacting collectors. Some previous work has used a similar technique in order to determine at run time the most appropriate collector algorithm. While the collector proposed in this work switches algorithm only as a fallback mechanism, it is clear that such a mechanism can be used in isolation to improve performance.

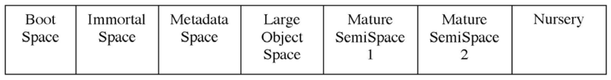


Figure 3. Generational copy/compact collector's heap layout

Printezis proposes an algorithm that switches between using a mark and sweep and a mark and compact collector to manage a generation in a generational collector [14]. He observes that each collector performs differently under different workloads; a mark and sweep collector offers faster old generation collection times, while the mark and compact collector performs nursery allocation more efficiently, and avoids the problem of fragmentation.

Based on this observation, the author describes a system which observes the current workload and *hot swaps* depending on which algorithm is more appropriate. If a program is observed to perform a large number of old-space collections it is more appropriate to use a mark and sweep approach. Conversely, if a workload performs frequent nursery collections with few survivors, a mark and compact collector will provide better performance.

Soman *et al* make use of a similar switching technique [15]. They observe that, since modern server environments may involve a number of different applications with different memory behaviors, a single collection algorithm is unlikely to perform best in all cases. Given this fact, they propose a system of application-specific GC that can use the most appropriate collector for the algorithm.

Soman *et al* differ from Printezis in several ways. They make use of Jikes RVM's many implemented collector algorithms, allowing them the flexibility to choose the most appropriate from a large pool. Additionally, their system uses annotations based on execution profiles in order to guide the selection of algorithm. As a result, their system does not incur a warm-up penalty while the appropriate collector is chosen.

5. Implementation Details

This section describes some of the interesting features of an implementation of the collector proposed in Section 3. The implementation was based on the Appel-style generational copying collector supplied with MMTk [16] running with Jikes RVM [17].

5.1 Heap Layout

Figure 3 shows the layout of the heap for our hybrid collector. It follows a similar design to that described by Appel, with additional spaces due to MMTk's implementation.

Spaces in MMTk have virtual address ranges fixed at build time. This allows references to space limits to be propagated as constants during compilation rather than being stored and referenced as variables at runtime. However, the full address range is rarely used for a given space. The size of a space is managed logically, by maintaining a record of the number of pages assigned to each space. This way the nursery and mature spaces can be resized dynamically according to the requirements of the algorithm.

The boot space is made up of the classes and data structures generated as part of the build process. This space is never garbage collected, and does not grow beyond the original size of the bootimage. By placing the boot space first in the virtual address space, offsets of code and data structures can be calculated at build time regardless of the layout of the rest of the heap.

The immortal space is also never garbage collected, and is used to store data that will exist for the lifetime of the virtual machine. This includes parts of the classloader and certain garbage collection data structures. Since objects in the immortal space never move, it is safe to use them during certain phases of garbage collection when data in the heap must be regarded as inconsistent.

The large object space is managed by Baker's treadmill collection algorithm [18]. This incremental, concurrent collector operates over a free-list allocated space. Objects larger than a set threshold are allocated in this space, with memory allocated in a page-sized granularity.

The mature space comprises two copying semi-spaces. Apart from brief periods during garbage collection, only one semi-space is active at any time. Allocation to the mature space is performed only during garbage collection, when objects are copied from the nursery or from one semi-space to the other. Allocation in the mature space is performed by a bump-pointer allocator.

The implementation of the mature space in MMTk differs from that described by Appel in that it does not remap the location of the old-space. In the algorithm outlined by Appel, upon evacuation of all nursery and mature space objects into the copy reserve, the memory containing live objects is remapped to the start of the heap. This is done to allow the nursery to be resized by simply sliding the location of the copy reserve space. Such a mechanism is not necessary in MMTk since the virtual address space allocated to the nursery is greater than the maximum size to which the nursery can grow.

The nursery is the space in which all new allocation occurs. It is a copy space, and is allocated to by a bump pointer. The nursery is placed at the end of the heap in order to simplify write barrier code. To maintain remembered sets of all references into the nursery it is necessary to determine on every pointer store whether the target is in the nursery and the source outside. By placing the nursery at the end of the heap, only a single comparison is necessary for each of these tests.

5.2 Mark Stage

At the start of a collection, a count is made of the number of copy reserve pages remaining. This number is calculated as a percentage of the copy reserve required by Appel's collector. As pages are allocated to the mature space for surviving objects, this count is decremented. When it reaches zero the copy reserve is full, and a compacting collection must be triggered.

Upon the triggering of a compacting collection, the behavior of object tracing changes. Previously, when an object was found to be reachable, it was copied to the mature space and replaced with a pointer to its new location. Under the compacting collector, objects are instead marked and left in place. Once the tracing phase is complete, an object in a compacting space can be in one of three states: forwarded, marked or garbage.

During the mark stage of a major collection, a total of the sizes of live mature space objects to be compacted is maintained. This indicates the amount of space required by compacted mature objects, and is used to determine the final placement of compacted nursery objects.

5.3 Sweeping

Compaction follows Jonkers' algorithm described in Section 3.2.2. The algorithm requires the temporary use of one word per object, and must be able to overwrite one bit. In this implementation, the word per object used by the algorithm is a pointer inside the header to per-class metadata. Since this is an aligned pointer, its low-order bit is always zero, and can thus be overwritten.

Some optimizations to the algorithm are made possible by implementation in MMTk. The chaining algorithm calls for two complete sweeps over the heap: one to update forward references, the other to compact and update backward references. However, in the generational system it is not necessary for these sweeps to cover the entire heap.

On a minor collection, there is no need to sweep the entire heap for references to the nursery. This information has already been logged in the remembered sets maintained by the write barriers. As a result, the first sweep of the heap requires processing the remembered sets and sweeping the nursery. This is a major improvement over the expense of sweeping every space in the system.

The initial sweep in a major collection, however, requires that the majority of the heap be swept. There exists no equivalent to the remembered sets for the mature space, meaning that pointers may exist anywhere in the heap. As a small optimization, it is not necessary to sweep the metadata space. This is because the metadata space is only used by the garbage collector, and is not used to determine liveness in other parts of the heap.

A difficulty arises in sweeping the heap during major collections. The version of MMTk upon which the implementation is based did not support linear sweeping through memory. All collection algorithms were implemented using tracing. As a result, the MMTk implementation had different assumptions from those required for a sweeping collector.

During a sweep, all objects encountered are scanned and their pointers processed. In order for this to function correctly, all pointers must be valid. Whether they point to live or dead objects, all pointers should refer to an object header. Pointers that do not resolve to an object header are dangling references, and can lead to errors.

Sweeping an immortal space is susceptible to this kind of error. By definition, an immortal space is never garbage collected. However, occasionally objects in an immortal space become unreachable from the program roots. In a tracing collector is is necessary only that all objects reachable from the program roots have valid references. Dangling pointers in unreachable objects will never be seen, since such an object will not be visited by a trace. However, a sweeping collector will encounter such references.

To work around this problem, two arrays are maintained to record liveness of objects in the immortal and boot spaces. These arrays maintain a bit per addressable word in these spaces. Upon marking an object, the bit corresponding to that object's header is set. Sweeping is performed by traversing these arrays to determine liveness, rather than by by sweeping the spaces themselves. This solution is not ideal, since it wastes space and causes a small time overhead in the marking phase. However in the absence of garbage collection in the immortal and boot spaces, it is necessary for correctness.

5.4 Block Copy

The compacting phase of the algorithm requires that objects be moved en-masse from one memory space to another. Once all objects are compacted within their space, they must be block copied to the appropriate location at the end of the mature space. In the case of minor collections, this requires only remapping the nursery objects. In major collections, both the nursery and old mature spaces must be remapped. As the block copy mechanism is used during a compacting collection, it has the additional requirement is that no allocation occur during the remapping. Doing so would allocate pages beyond the system's budget.

Block copying is performed using the mmap system call. By default, MMTk obtains virtual address pages from the operating system's scratch file. Modifications to this system were made to instead allocate pages from a named heap file. By maintaining information on the offset in the heap file mapped to a given virtual page it is possible to remap the data, effectively modifying the virtual address associated with it. In this way, data can be quickly remapped from one virtual address to another.

This implementation adds some run-time overhead, due to the need for additional system calls. The existing MMTk collectors do not incur the overhead of space management, because they do not unmap pages. When a virtual address range is first allocated, MMTk maps scratch memory to that range. However, when objects are evacuated from that address range, MMTk simply stops using the memory, rather than unmapping it. This saves the overhead of remapping when the same address range is needed again. As a result, the standard MMTk collectors will often have more memory mapped than the maximum heap size would allow.

This is not possible when remapping is required, as in the case of the generational copying/compacting collector. In this case it is possible that a new physical address range will be mapped to a given virtual address range. It is important to ensure that no two physical addresses are mapped to the same virtual address. To avoid this, memory is unmapped whenever it is not in use.

6. Experimental Results

This section presents the results of running the new garbage collection algorithm with a series of benchmarks. The algorithm was implemented by modifying the GenCopy collector distributed with Jikes RVM version 2.3.4. Benchmarks were run using a machine with an Intel Pentium 4 processor running at 2.26GHz and with 512 Mb of RAM. The Operating System was Mandrake Linux 9.2, using kernel version 2.4.22-10mdk.

6.1 Methodology

Timing and garbage collection information for each of the selected benchmarks was gathered. Both generational copying and generational mark sweep collectors were analyzed for comparison with the new generational copying/compacting collector.

The methodology used to obtain the numbers was the same for each collector. First, a VM was built configured with the appropriate collector. Next, each benchmark was run 11 times within a single VM invocation. The first run served as a warm-up, compiling all necessary methods using Jikes RVM's optimizing compiler (at opt-level 2). This eliminated the overhead of compilation from subsequent runs. The results for the compilation run were discarded. The remaining ten runs were used as timing runs. A full-heap garbage collection was performed before each timing run.

6.2 Measurements

The total elapsed time (both mutator and GC) was measured for each benchmark. Additionally, a count was maintained of both major and minor collections. In the case of the GenCC algorithm, the number of nursery and mature compactions was also noted.

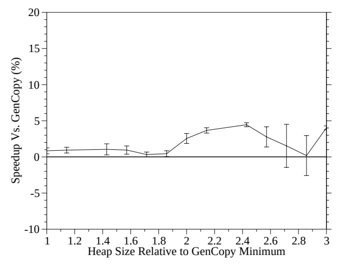
In the remainder of this section, the traditional generational copying collector will be referred to as *GenCopy*, the generational mark sweep collector as *GenMS* and the new generational copying/compacting collector as *GenCC*.

6.3 Benchmarks

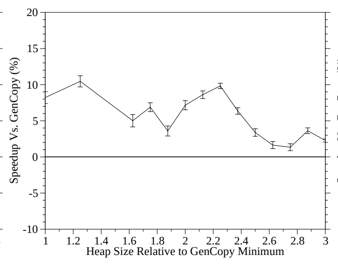
Benchmarks from the SPECjvm98 suite [19] were used to examine the performance of the generational copying/compacting algorithm. All benchmarks were run in the default configuration (input size 100). Figure 4 shows a comparison between the total runtimes of GenCopy and GenCC for each benchmark.

Of the benchmarks, _202_jess, _213_javac and _228_jack exhibit the most interesting garbage collection behavior, and will be examined in Section 6.4. Of the remaining benchmarks, it can be seen that _222_mpegaudio shows only a minor improvement, _201_compress and _227_mtrt show larger improvements, while _209_db shows a performance decrease.

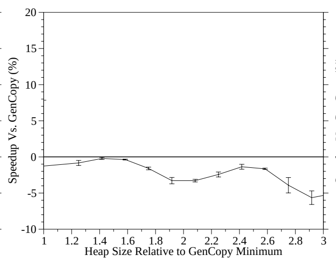
It would be expected that _222_mpegaudio would see little improvement through an improved garbage collection algorithm. Since the amount of garbage collection activity is very low in this benchmark, any improvement in the collector will offer little speedup overall. _227_mtrt and _201_compress demonstrate some



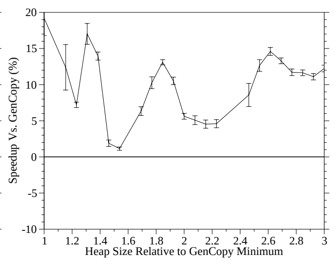
(a) _201_compress



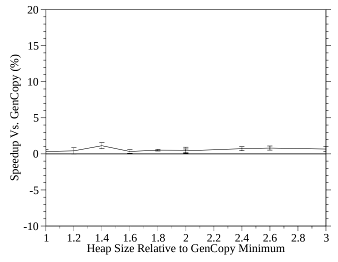
(b) _202_jess



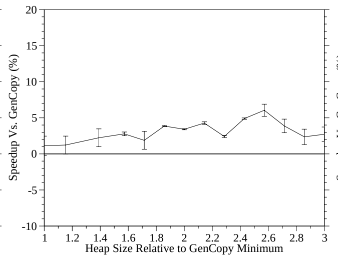
(c) _209_db



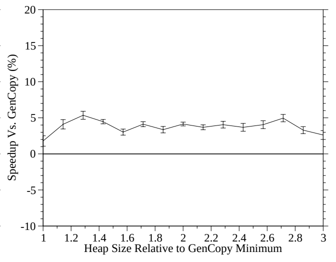
(d) _213_javac



(e) _222_mpegaudio

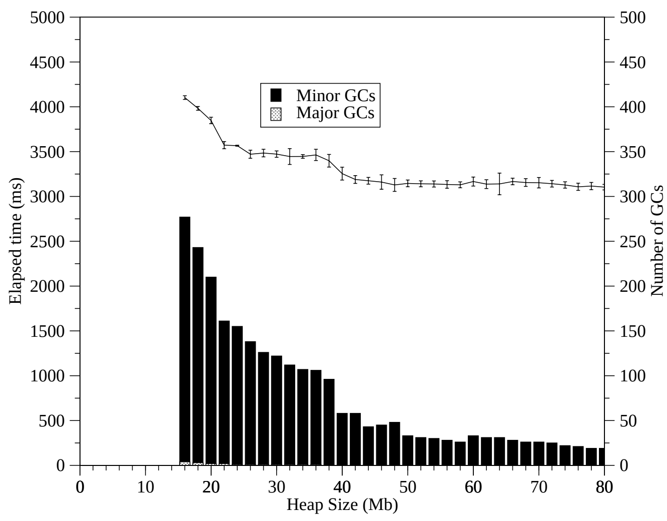


(f) _227_mtrt

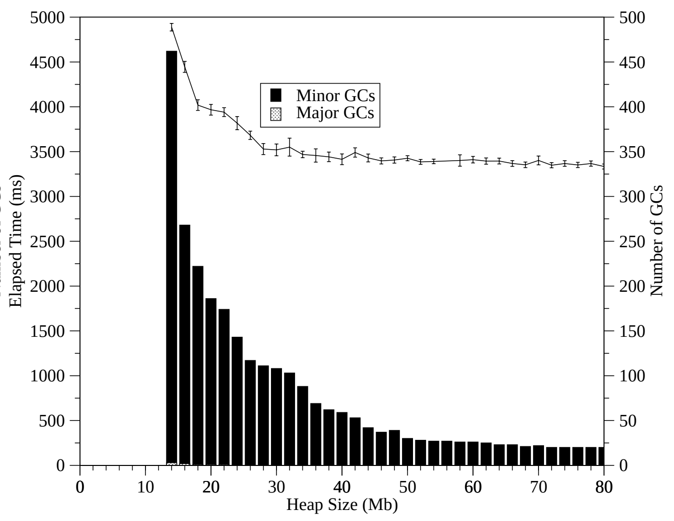


(g))_228_jack

Figure 4. All SPECjvm98 benchmarks, 10% copy reserve



(a) GenCopy



(b) GenMS

Figure 5. _202_jess when run with traditional collectors

improvement. Both of these benchmarks have relatively low nursery survival rates. This means that the new algorithm allows more objects to become unreachable between collections, and so triggers fewer major collections. This is particularly apparent at larger heap sizes.

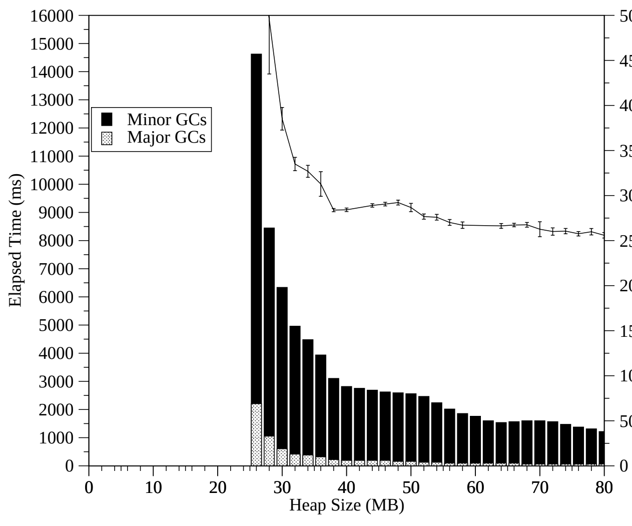
The remaining benchmark, _209_db, shows a minor decrease in performance. The reason for this is that it has frequent major collections, indicating a high nursery survival rate. This in turn causes the compacting collector to be called more often, leading to a performance degradation. It is worth noting, however, that the performance drop is less than 5%, while the performance gained by the other benchmarks is substantially higher.

6.4 Results

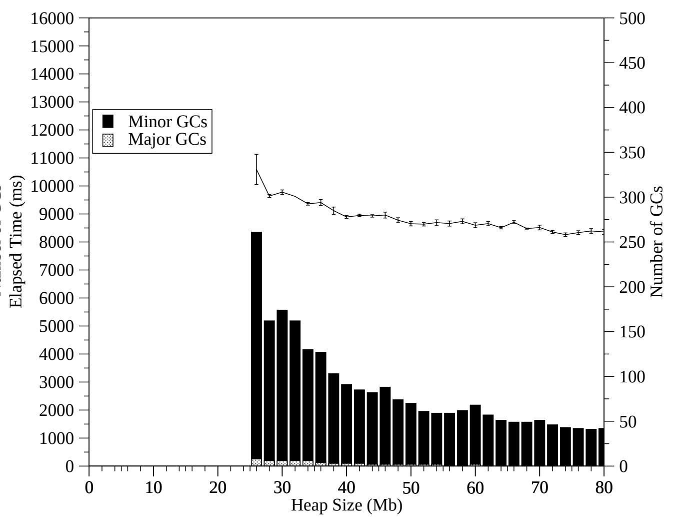
For the remainder of this section, we will look in detail at _202_jess, _213_javac and _228_jack. Figures 5 to 10 show the results.

Each graph shows the elapsed time for a benchmark run using a given configuration on the left y-axis, plotted as a line with 90% confidence intervals, and show the number of collections on the right y-axis, plotted as stacked bars. Using Figures 5 and 8 as examples, Figure 5(a) and Figure 5(b) show the performance over various heap sizes of _202_jess using the GenCopy and GenMS collectors. Figures 8(a) to 8(c) show the results over a range of heap sizes when a copy reserve of 5%, 10% and 80% is used. The remaining graphs (Figure 8(d), 8(e) and 8(f)) show a varying copy reserve percentage for a fixed heap size.

Figures 5(a) and 5(b) show that the performance of _202_jess is better using GenCopy than it is using GenMS. Additionally,

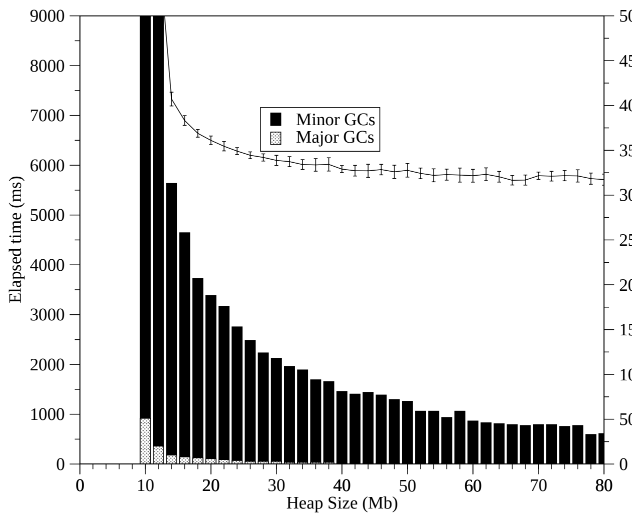


(a) GenCopy

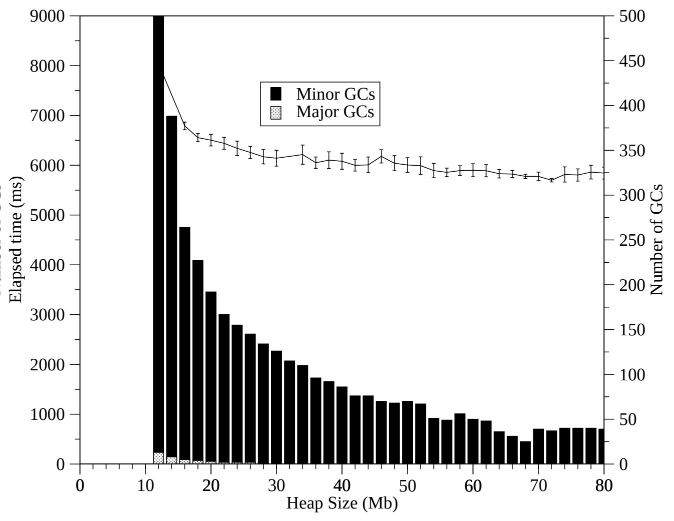


(b) GenMS

Figure 6. _213_javac when run with traditional collectors



(a) GenCopy



(b) GenMS

Figure 7. _228_jack when run with traditional collectors

neither collector performs many major collections, indicating that the survival rate in the nursery is low. GenCopy benefits from this, since it uses a variable-sized nursery, while GenMS uses a fixed size. As a result, the nursery is able to take up more of the heap under GenCopy, leading to fewer collections.

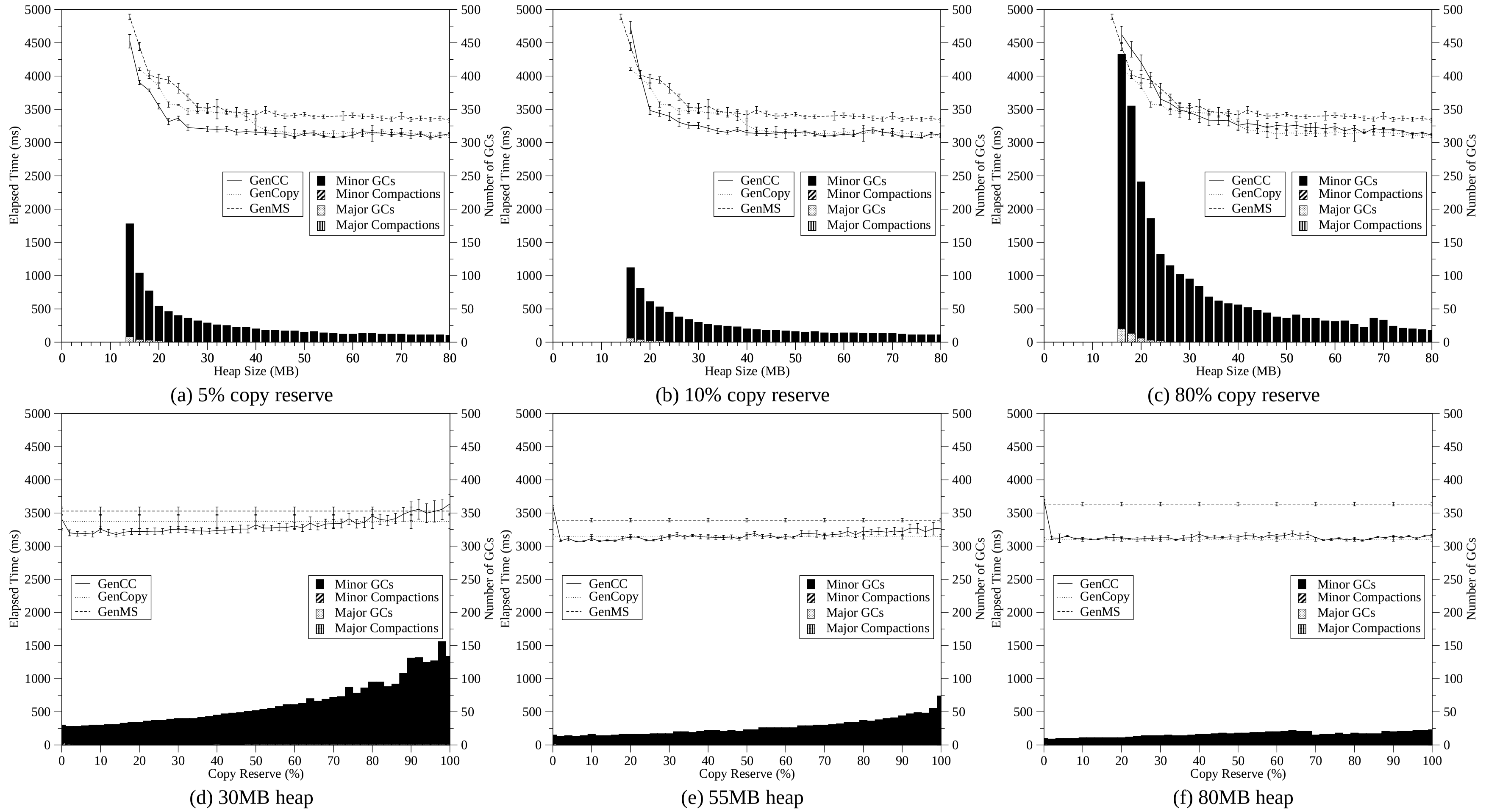
It can be seen from Figure 8 that the performance of GenCC is almost always better than GenMS. This is to be expected, since GenCC takes advantage of the low nursery survival rate, and is able to perform far fewer collections. Additionally, GenCC's performance is almost always comparable to or better than GenCopy. The improvement is most noticeable at small copy reserve sizes.

Very little compaction is performed when running _202_jess. since the nursery survival rate is low. The compacting collector is only initiated when the copy reserve is set to its minimum. Since the minimum copy reserve consists of eight pages used to copy VM and GC data, Figures 8(d) to 8(e) show that, in most cases, the survivors of a nursery collection fit entirely inside that minimum space.

Figures 6(a) and 6(b) show that there are significantly more major collections in _213_javac than in _202_jess. This indicates that the nursery survival rate is higher. Of the benchmarks presented, _213_javac exhibits the effects of the generational hypotheses most clearly and is, as a result, the most interesting of the SPECjvm98 benchmarks from a garbage collection perspective.

It can be seen in Figure 9(a) that when the _213_javac benchmark is run using a small copy reserve, several compacting collections are triggered. However, despite the additional overhead of these collections, GenCC outperforms GenCopy. This is particularly noticeable at larger heap sizes, where more space is available for the collector to manage. Additionally, in the majority of cases, GenCC outperforms GenMS.

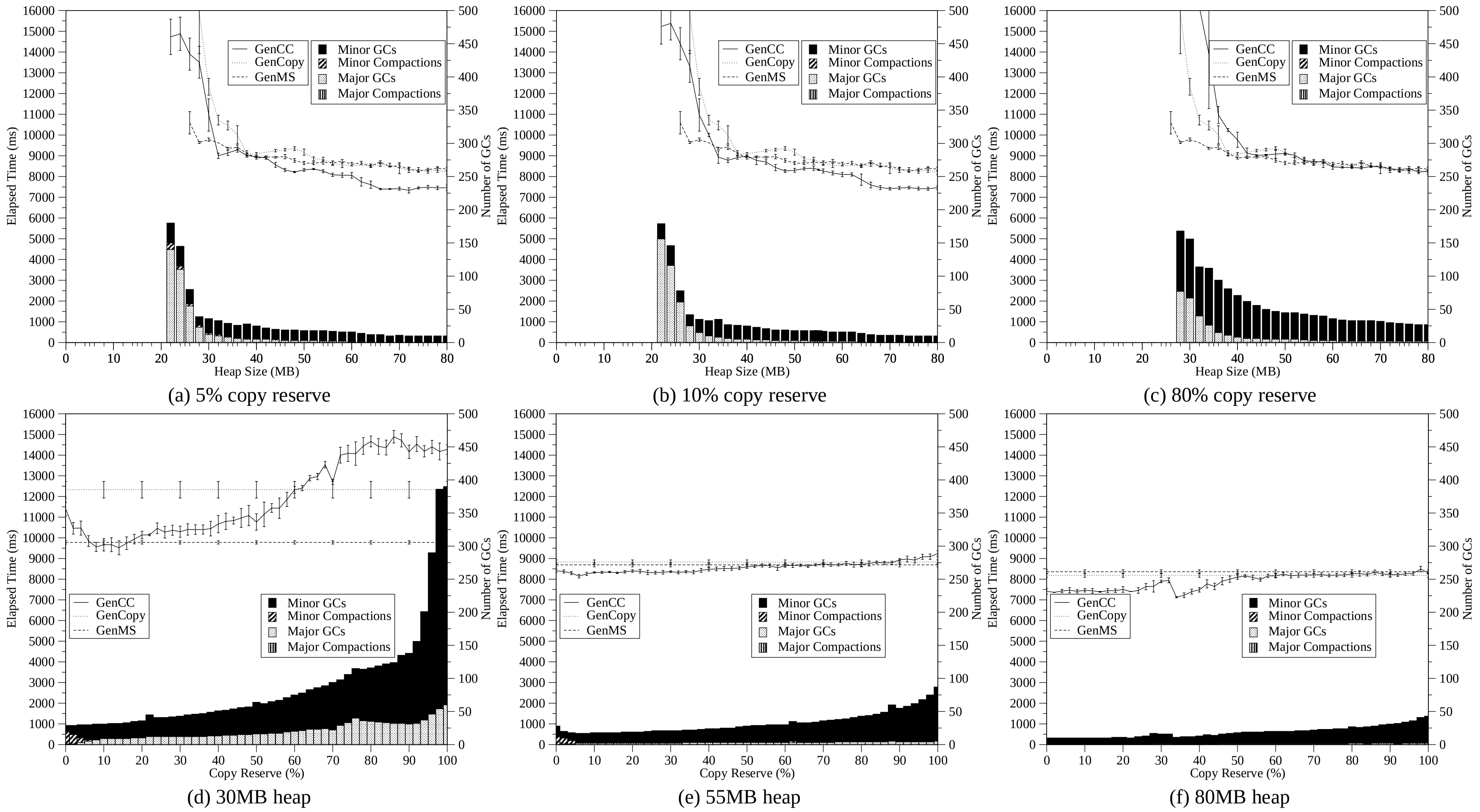
The performance advantage becomes smaller once the copy reserve percentage is increased. Figure 9(f) shows an 80% copy re-



(a) 5% copy reserve (b) 10% copy reserve (c) 80% copy reserve

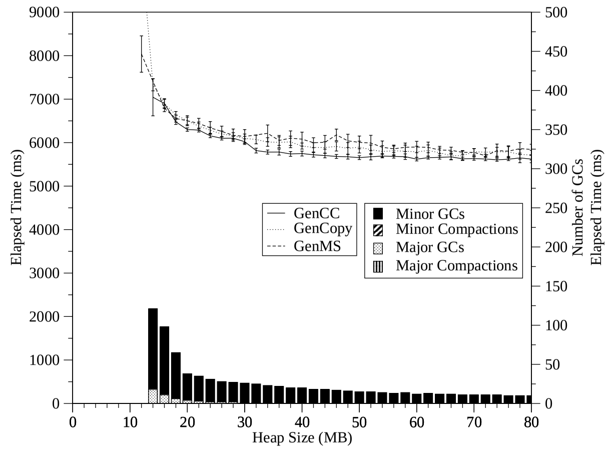
(d) 30MB heap (e) 55MB heap (f) 80MB heap

Figure 8. _202_jess

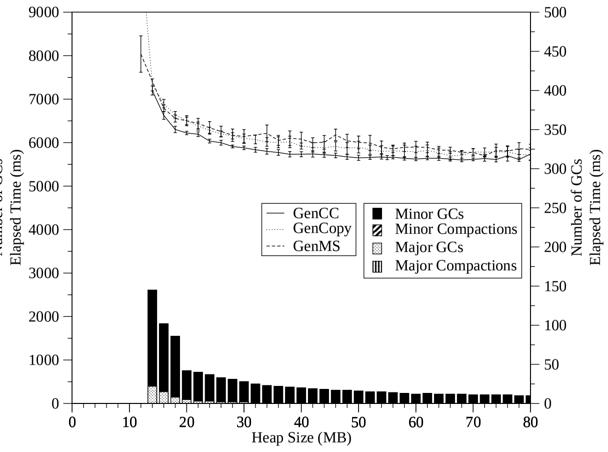


(a) 5% copy reserve
(b) 10% copy reserve
(c) 80% copy reserve
(d) 30MB heap
(e) 55MB heap
(f) 80MB heap

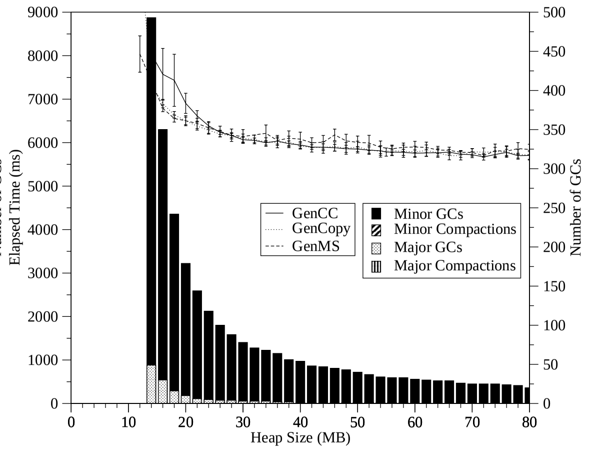
Figure 9. _202_javac



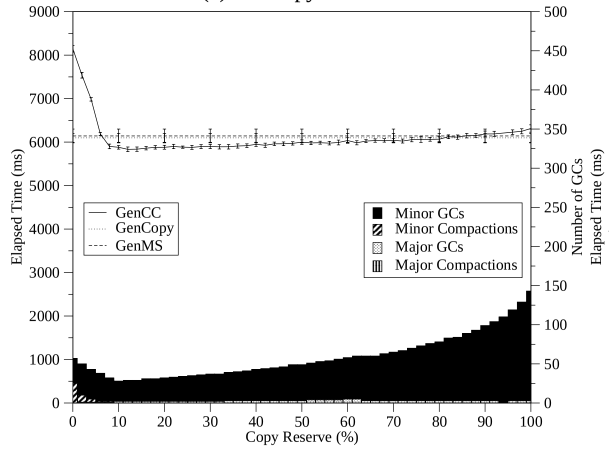
(a) 5% copy reserve



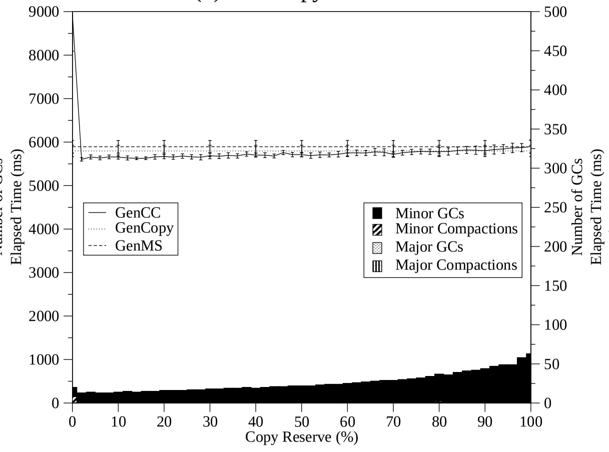
(b) 10% copy reserve



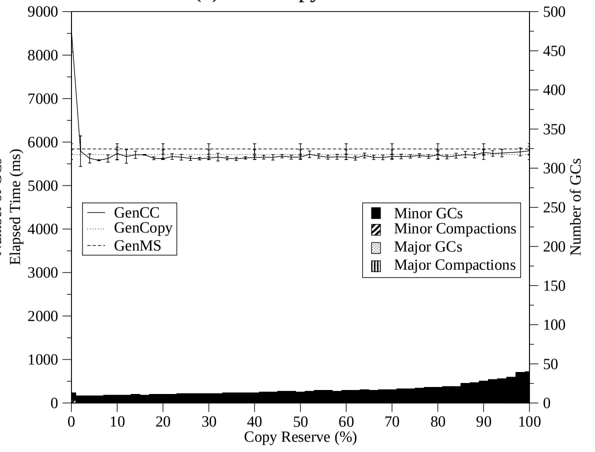
(c) 80% copy reserve



(d) 30MB heap



(e) 55MB heap



(f) 80MB heap

Figure 10. _202_jack

serve, where it can be seen that the performance of GenCC degrades to similar to or worse than GenCopy. This is expected since, while theoretically the GenCC algorithm with 100% copy reserve should be identical to GenCopy, the implementation adds some overheads that lead to performance degradation. For example, the implementation of the block copying mechanism relies on maintaining additional state regarding mapped pages, and can require additional system calls to manage the memory space correctly. Additionally, support for the reference chaining technique requires that a check be made on some memory accesses during GC. For large heaps, this is a minor effect, as can be seen in Figures 9(e) and 9(f). However, when a smaller heap is used in Figure 9(d) these overheads can lead to significant performance loss at higher copy reserve sizes. It is expected that a production version of the GenCC algorithm would generally be used with the copy reserve set at a suitable value (determined experimentally or dynamically). As a result, these overheads would not be an issue.

Figures 7 and 10 show performance data for _228_jack. It can be seen in Figures 7(a) and 7(b) that the traditional collectors perform few major collections. As in the case of _202_jess, this indicates that the nursery survival rate is low.

Figure 10(a) shows that, even with a copy reserve of 5%, the compacting collector is not triggered. This is a result of the low nursery survival rate. The GenCC collector here outperforms both the GenCopy and GenMS collectors. As the copy reserve increases, the performance of GenCC converges with that of the traditional collectors.

Figures 10(d), 10(e) and 10(f) indicate that compaction occurs only when the copy reserve is at its minimum. If these extreme cases are discounted, it can be seen that the performance of GenCC is comparable to or better than both GenCopy and GenMS for all three heap sizes.

7. Conclusions

We have presented a generational copying algorithm that makes use of a reduced copy reserve to improve performance. On the rare occasion when the reduced copy reserve is insufficient, our collector falls back on a compaction technique. This ensures correctness in the worst case.

We have provided details of our implementation, and shown an evaluation of the collector. We found that the choice of copy reserve size is critical to the performance of the algorithm. If this reserve is set too high, the benefits of improved space utilization are lost. If, on the other hand, the copy reserve is too small, overall performance is hurt by the frequent use of the compacting collector.

Our experiments show that, when a reasonably-sized copy reserve is used, our collector outperforms both the traditional generational copying and generational mark sweep collectors in most cases. Using a 10% copy reserve, we observe an average speedup of 4% over a standard generational copying collector, with a maximum speedup of almost 20%.

Acknowledgments

This work is supported by the National Science Foundation under grants Nos. CCR-0085792, CNS-0509377, CCF-0540866, and CNS-0551658, and by IBM and Microsoft. Any opinions, findings and conclusions expressed herein are the authors and do not necessarily reflect those of the sponsors.

References

[1] JONES R., LINS. R. Garbage Collection: Algorithms for Automatic Dynamic Memory Management. Wiley, 1996

[2] LIEBERMAN, H., HEWITT, C. A Real-Time Garbage Collector Based on the Lifetimes of Objects. In *Commun. ACM* 26, 6 (Jun. 1983), 419-429.

[3] UNGAR, D. Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm. In *Proceedings of the ACM Symposium on Practical Software Development Environments*, pp 157–167, 1984.

[4] BLACKBURN, S. M., JONES, R., MCKINLEY, K. S., AND MOSS, J. B.. Beltway: Getting Around Garbage Collection Gridlock. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation* (Berlin, Germany, June 17 - 19, 2002). PLDI '02. ACM Press, New York, NY, 153-164.

[5] APPEL, A. W. Simple Generational Garbage Collection and Fast Allocation. In *Software – Practice And Experience*, 19(2):171–183, February 1989.

[6] COHEN, J., NICOLAU, A. Comparison of Compacting Algorithms for Garbage Collection. In *ACM Trans. Program. Lang. Syst.*, 5, 4 (Oct. 1983), 532-553.

[7] HADDON, B. K., WAITE, W. M. A Compaction Procedure for Variable Length Storage Elements. In *The Computer Journal*, 10:162–165, August 1967

[8] FISHER, D. A. Bounded Workspace Garbage Collection in an Address Order Preserving List Processing Environment. In *Communications of the ACM*, 18(5):251–252, May 1975.

[9] WILSON, P. R. Uniprocessor Garbage Collection Techniques. Technical Report, University of Texas, January 1995

[10] JONKERS, H. B. M. A Fast Garbage Collection Algorithm. In *Information Processing Letters*, 9(1):25–30, July 1979.

[11] The Java Hotspot Virtual Machine v1.4.1. White Paper. http://java.sun.com/products/hotspot/index.html

[12] VELASCO, J. M., ORTIZ, A., OLCOZ, K. AND TIRADO, F. Adaptive Tuning of Reserved Space in an Appel Collector. In *Proceedings of the European Conference on Object Oriented Programming* (Oslo, Norway). vol 3086 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004.

[13] SACHINDRAN, N., MOSS, J. E. B. AND BERGER, E. D. MC^2: High-Performance Garbage Collection for Memory-Constrained Environments. In *Proceedings of the 19th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Vancouver, BC, Canada). pp 81–98. ACM Press.

[14] PRINTEZIS, T. Hot-Swapping Between a Mark&Sweep and a Mark&Compact Garbage Collector in a Generational Environment. In *Java Virtual Machine Research and Technology Symposium*, USENIX 2001

[15] SOMAN, S., KRINTZ, C., BACON, D. F. Dynamic Selection of Application-Specific Garbage Collectors. In *Proceedings of the 4th International Symposium on Memory Management* (Vancouver, BC, Canada, October 24 - 25, 2004). ISMM '04. ACM Press, New York, NY, 49-60.

[16] BLACKBURN, S. M., CHENG, P. AND MCKINLEY, K. S. Oil and Water? High Performance Garbage Collection in Java with MMTk. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering* (Washington, DC, USA). pp 137–146. IEEE Computer Society.

[17] ALPERN, B., ATTANASIO, C. R., BARTON, J. J., COCCHI, A., HUMMEL, S. F, LIEBER, D., NGO, T. MERGEN, M., SHEPHERD, J. C. AND SMITH, S. Implementing Jalapeño in Java. In *Proceedings of the ACM Conference on Object Oriented Programming Systems, Languages and Applications* (Denver, CO, USA). pp 314–324. ACM Press.

[18] BAKER, H. G. The Treadmill: Real-Time Garbage Collection Without Motion Sickness. In *ACM SIGPLAN Notices*, 27(3):66-70, March 1992.

[19] SPEC. SPECjvm98 Benchmarks, 1998. http://www.spec.org/osg/jvm98.