# AN IMPROVED GENERATIONAL COPYING GARBAGE COLLECTOR

A Thesis

Submitted to the Faculty

of

Purdue University

by

Philip McGachey

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science

December 2005

To my parents, for all their support and encouragement over the years.

# ACKNOWLEDGMENTS

Thanks go to my advisor, Tony Hosking, whose particular brand of cheerful advice and guidance helped immeasurably during the preparation of this thesis. Many thanks also to my committee, Jan Vitek and Suresh Jagannathan.

My gratitude goes also to Steve Blackburn, Daniel Frampton and Robin Garner for their work on MMTk, as well as their advice on the details of implementation. Their experience and eagerness to help greatly eased the implementation of this work.

Finally, special thanks to Adam Welc, upon whose original idea this work was based. His additional input both in discussions concerning the implementation details and in our late night “strategy meetings” has been invaluable.

TABLE OF CONTENTS

Page

Page

# LIST OF FIGURES

# ABSTRACT

McGachey, Philip. M.S., Purdue University, December, 2005. An Improved Generational Copying Garbage Collector. Major Professor: Antony Hosking.

Garbage collection frees the programmer from the responsibility of tracking dynamically allocated memory. As an increasingly popular element of modern programming languages, it is essential that garbage collection be performed efficiently. This thesis investigates a new method by which garbage collection can be performed.

The algorithm described combines a standard generational copying collector with a mark and compact collector. The result is a generational copying collector that operates with a smaller copying reserve overhead than traditional Appel-style collectors, while maintaining correctness in the worst case. When sufficient objects survive a collection, a compacting collection ensures that all data are accommodated.

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

# 1 INTRODUCTION

*The run time performance of a generational copying garbage collector can be improved by reducing the size of the copy reserve*.

The recent popularity of managed languages such as Java and C# have led to a great deal of research into the performance of runtime systems. A major component of any such environment is the garbage collector. This document presents a new technique through which garbage collection performance can be improved.

## 1.1 Garbage Collection

Garbage collection is the method by which dynamically allocated memory is automatically reclaimed. Programming languages with garbage collection free the programmer from the responsibility of tracking memory allocation, making code simpler to write. As well as reducing memory leaks, garbage collection eliminates whole classes of errors, such as dangling references or double releasing. Finally, garbage collection simplifies software engineering. In traditional languages such as C, it is necessary for programmers to negotiate which of them has the responsibility for freeing memory passed between modules. Since garbage collection has a global view of liveness this is no longer necessary.

## 1.2 A New Algorithm

This document presents a new garbage collection algorithm. It combines elements of a generational copying collector with a compacting collector. The resulting algorithm takes advantages of the positive features of a generational copying collector such as improved

partitioning objects by age, automatically compacting objects for improved spatial locality and fast allocation.

The new collector is able to reduce the space required for the copy reserve, alleviating one of the major drawbacks of the generational copying collector. The copy reserve in the new algorithm is set significantly smaller than the maximum required reserve. In the majority of cases, this presents no problem, since relatively few objects survive a garbage collection. In the rare cases where more objects survive than can fit in the copy reserve, correctness is maintained by the use of a compacting collector.

## 1.3 Summary Of Results

The collector was implemented and its performance examined. The results chapter of this document outlines the findings. In the majority of cases, performance of the new collector with suitable parameters was better than both the standard generational copying collector and the generational mark and sweep collector. When performance was degraded, it was often the result of an inappropriate copy reserve size. Only on one benchmark was performance seen to be uniformly poorer than the traditional algorithms.

## 1.4 Outline

The remainder of this thesis is structured as follows: Chapter 2 outlines the background upon which the new garbage collection algorithm has been built. Chapter 3 discusses the design of the collector, while Chapter 4 gives some details on the implementation. Chapter 5 describes some experiments run to measure the performance of the new collector. Chapter 6 concludes and suggests some possible future research directions resulting from this work.

# 2 BACKGROUND

## 2.1 Uniprocessor Tracing Garbage Collection

Automatic dynamic memory management, or *garbage collection* has been a topic of research for several decades [1]. The recent popularity of managed languages such as Java and C# has provoked new interest in devising efficient garbage collection algorithms.

While a great many techniques exist for garbage collection they all have the same high-level specification. Any memory location that can be reached transitively from a set of root pointers is considered to be live. Any object outside the live set is considered to be garbage, and can be reclaimed. The garbage collection algorithm is responsible for determining which objects are garbage, and for returning their storage space to the system to be reused by later allocations.

While many algorithms also exist for parallel garbage collection, this work centers around single-processor tracing techniques.

### 2.1.1 Motivation

Garbage collection offers several clear advantages to software engineers over manual dynamic memory management. By reclaiming memory automatically once it is no longer required, a garbage collected system is immune to whole classes of bugs that have previously caused havoc in large systems.

*Memory leaks* result when programmers allocate memory but forget to release it once it is no longer necessary. This memory then becomes unavailable to the system. Even if a series of memory leaks does not cause a program to crash due to memory exhaustion, they can have an adverse effect on performance. Chunks of allocated but unused memory cause fragmentation, which destroys spatial locality. This can result in poor cache performance

or an increase in paging. A garbage collected system avoids memory leaks by removing the responsibility of deallocating memory from the programmer. Once a memory location is determined to be unreachable, it is returned to the system.

Another common source of errors in manually managed memory is that of *dangling references*. If a programmer accidentally frees a region of memory prematurely, any subsequent attempt to access that memory location may produce an error. At best, this error may cause the system to fail due to a segmentation violation. A worse case would be if the memory location had been overwritten with other data. Reading or writing to this location will cause the program execution to be incorrect, but may not cause the program to terminate.

Dangling reference bugs can be difficult to detect, since memory allocation patterns may vary from one program execution to another. It may be the case that the memory location pointed to by a dangling reference is not overwritten immediately, causing some dereferences to return the intended result. Automatic memory management eliminates dangling pointer bugs by only deallocating memory once no live reference to it exists. Since it is guaranteed that any pointer usable by the programmer points to valid memory, dangling pointer dereferences become impossible.

Garbage collection is also of benefit when constructing large, modular systems. In manually memory-managed programs, it is necessary for programmers to determine who has the responsibility for deallocating a piece of memory. This may occur in a different module from that in which the memory was allocated. Confusion between programmers can lead to objects being released multiple times, or not at all. The use of garbage collection alleviates this problem; objects are freed according to a global view of the system. As such, no programmer is responsible for deallocation.

Additionally, the use of a garbage collector can simplify allocation. When memory is manually allocated and freed, *fragmentation* is a common problem. This arises when an object is deleted from the middle of an allocated region. Unless the allocator is able to place a new object of equal size in the gap, the space will be wasted. This problem has led to the use of *free list allocators* which interleave old and new objects to reduce fragmen-

tation [2]. While some garbage collected systems require the use of a free list allocator, many others perform compaction as part of their execution, eliminating the problem of fragmentation.

### 2.1.2 Tracing Collectors

Tracing garbage collectors determine which objects are reachable from the program *roots*. The roots of a program are defined as the programmer's entry points to the graph of dynamically allocated data. Generally, the roots will include pointers held in the stack or registers, as well as static or global variables. Tracing algorithms assume that all live objects can be reached through these roots, or through a chain of objects beginning at a root.

Reachability offers an approximation of liveness; objects that can be reached transitively from the roots are considered to be live, those which cannot be reached are garbage. It may be that an object determined by this method to be live is actually no longer required. For example, the program may have finished operating on a data structure, but a reference to it still exists through a pointer that has not yet been overwritten. While reachability is a conservative approximation of liveness, it guarantees that a live object is never considered to be garbage.

### 2.1.3 Mark and Sweep Collectors

A mark and sweep collector traces through the heap in order to determine which objects are no longer live. These objects are then collected, and the space they occupied can be reused.

During the tracing phase, every reachable object is marked. Generally this is done through setting a bit in the object header, although a separate bitmap may also be maintained. Once the tracing (or mark) phase has completed, all live objects have been marked, while all garbage objects have not. The second phase of the collection, the sweep, is then

performed. The heap is scanned linearly, and all unmarked objects encountered are reclaimed.

A mark and sweep collector allocates using a free list allocator. This is necessary since the heap is never compacted, and so will become fragmented over time. Whenever a memory request is made, the allocator locates space of an appropriate size into which the new object can be placed. This allocation method can be time-consuming and, in extreme cases, can fail if a large enough space cannot be found even though sufficient memory exists in the system. This problem can be alleviated through the use of segregated free lists where multiple free lists sorted by size are maintained, and objects are allocated to the smallest space available.

Additionally, free list allocators are unable to exploit spatial and temporal locality. Since objects created at the same time may well be accessed together, it would be preferable for them to be located close together. This way when one of them is loaded into the cache, the others may be brought in at the same time. However, since the free list allocator places objects wherever they fit it is unable to take advantage of this property.

### 2.1.4 Copying Collectors

Copying, or *scavenging* collectors separate the heap into *spaces*, and copy live objects between them. Once the live objects have been evacuated from a space, all objects remaining in that space are garbage, and can be reclaimed. The simplest copying collector uses two spaces, moving live object from one to the other when memory is exhausted; one of the most commonly used algorithms is due to Cheney [3]. The choice of the number of spaces in the heap influences collection frequency and memory utilization.

When a copying collector is used, memory in the heap can be allocated sequentially in memory. Each object allocated is placed directly after the previous object, as shown in Figure 2.1(a). This system has several advantages. First, the allocation sequence is very simple. In order to allocate space it is necessary only to increment a pointer. When

copying, objects are again allocated sequentially. As a result, the heap is automatically compacted on every collection.

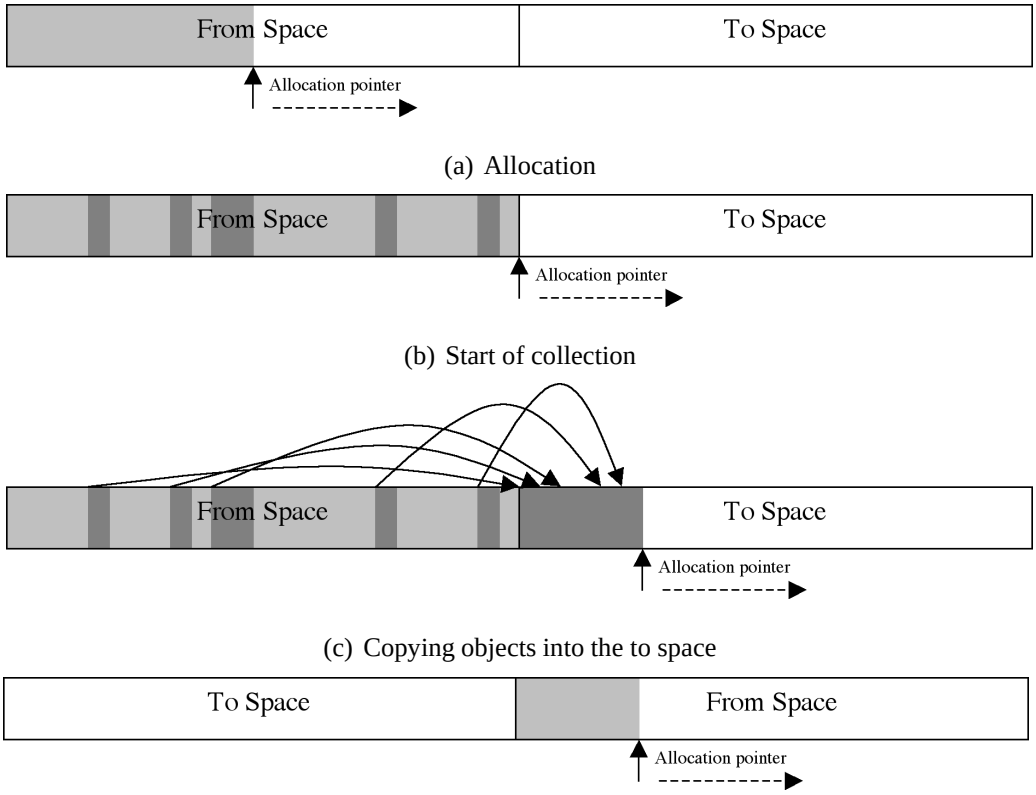



Figure 2.1. Semispace copying collector

When the first semispace is filled, as shown in Figure 2.1(b), a collection is triggered. Live objects are discovered by tracing, and are indicated in dark gray. The live objects are then copied from the *from space* into the *to space*, as in Figure 2.1(c). When all the live objects have been copied, the roles of the from space and to space are reversed. Figure 2.1(d) shows allocation continuing in the new from space.

An additional benefit with a copying collector is the opportunity to optimize the placement of objects for cache performance. As a simple heuristic, objects that refer to one another may be referenced within a short period of time. When moving such objects, the collector may try to place them in the same cache block [4].

## Copying Overheads

While copying collectors have significant advantages due to their ability to lay out data, they come with some substantial overheads. The most obvious of these is the need to move objects. Copying data is an expensive operation and can dominate the pause time of the garbage collector. This is particularly true for simple copying collectors that move all live objects on each collection.

In addition, it is necessary to update all references to an object that has moved during a collection. Failure to do so leads to dangling references and incorrect program behavior. References are commonly updated through the use of *forwarding pointers*; when an object is copied to a different space its original location is overwritten with a pointer to its new location. This way any subsequent references discovered by the collector can be updated with the new address.

## Copy Reserve

The major drawback to copying collectors, however, is not the overheads of moving data or updating references. 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 objects. This space is referred to as the *copy reserve*. In the case of a simple two-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.

### 2.1.5 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. The tracing phase of the mark and compact collector is the same as in the mark and sweep case. Once it has completed, all live objects are marked and all garbage objects are not. In the second phase, however, live objects are compacted as opposed to dead objects collected. All marked objects are relocated towards the “front” of the heap (where the front of the heap is defined as low memory addresses, while the back is high memory addresses). In this way, garbage objects are overwritten and live objects retained.

Compacting collectors eliminate the fragmentation found in mark and sweep collectors. Since no gaps are left between live objects it is not necessary to use a free list allocator. A simple bump-pointer instead can be used, allocating the first new object directly after the final compacted object. Additionally, since objects are slid towards the front of the heap, allocation order is maintained. This offers better spatial locality than with a free list allocator.

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 of the copying collector 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, 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 is 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 a large external data structure has been the focus of some research. Generally these algorithms require multiple additional passes over the heap, leading to longer pause times which make compacting collectors impractical.

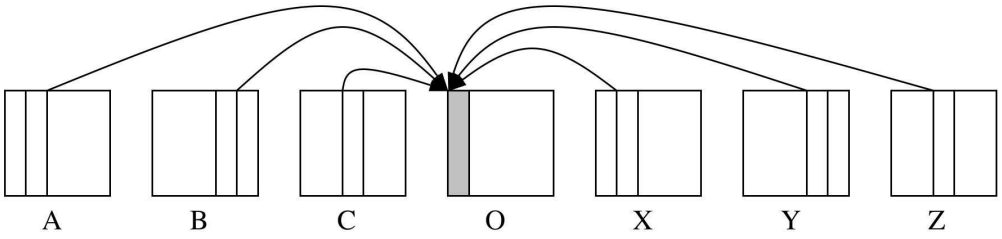
## 2.2 Reference Chaining

In [5], Jonkers proposed a technique for managing forwarding pointers during a compacting collection. This algorithm has the advantage over others in that it tracks forwarding pointers without allocating additional storage. It is performed through a technique of reference chaining.

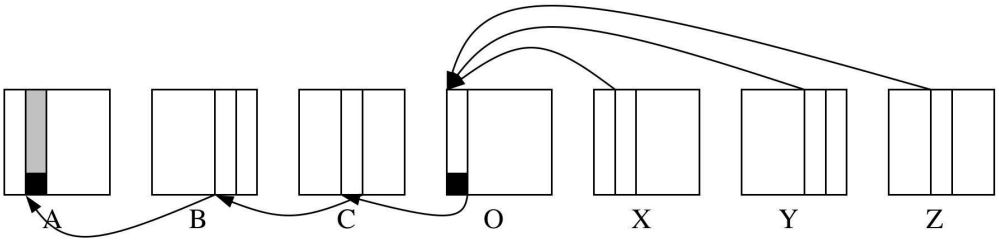
The insight behind reference chaining is that in order to properly update pointers it is necessary to record either the object being moved or the references to that object. Traditional approaches track the former, and references are updated in a scan of the heap. The chaining algorithm instead associates references to an object with that object itself, and updates pointers as soon as the destination of the object is known.

The algorithm requires that a word of the object be moved, and that one bit in that word be available for the algorithm's use. It involves two passes over the heap: one to update forward references, the other to move objects and update backward references. An illustration of the algorithm is shown in Figure 2.2.

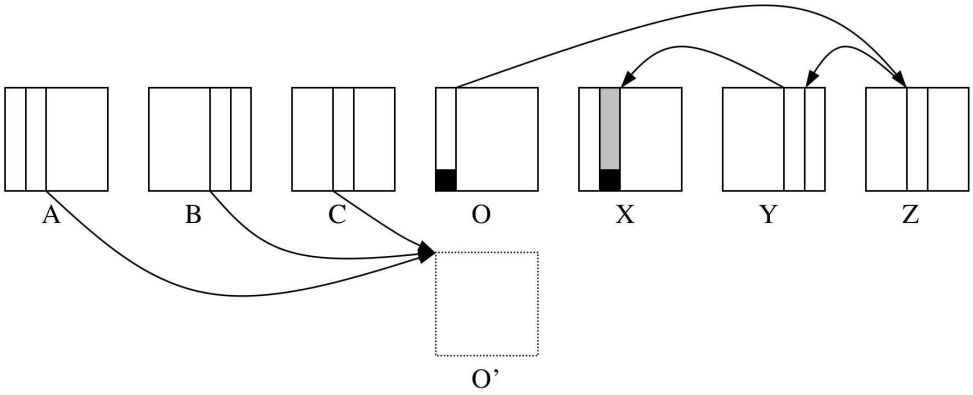
Object O is going to be moved as part of the compaction phase. Objects A, B and C hold references to object O. These are forward pointers, since objects A, B and C are at lower addresses in the heap than object O. Similarly, objects X, Y and Z hold backward references to O, since they are located at higher addresses. For this illustration, the space in which each object is allocated is irrelevant.



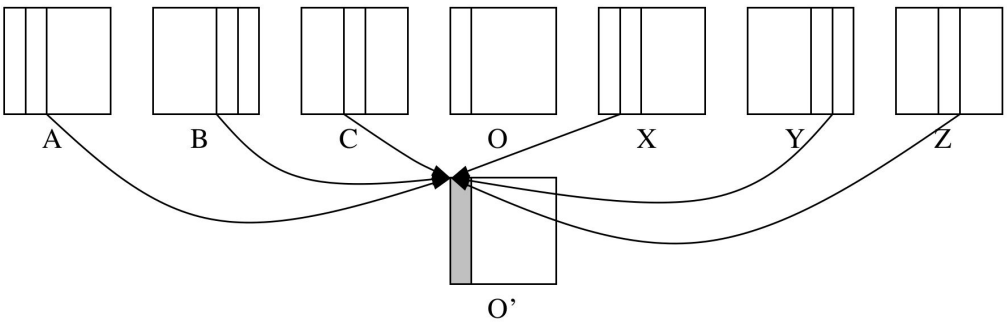

(a) Before chaining




(b) Forward references chained




(c) After first sweep




(d) After completion of chaining

Figure 2.2. Jonkers chaining algorithm

One field of object O, shown in gray in Figure 2.2(a), is available for use by the algorithm. This may be a word in the header, such as a hash code. One bit of the available word must be reserved for the chaining algorithm.

The heap is scanned from low addresses to high addresses. The first object encountered is object A. Upon tracing the references in A, it is discovered that one points to object O, which has been marked to move. Once object O has moved, it will be necessary for the reference in A to be updated. The available word of O is stored in the field of A that formerly contained a pointer to O. The available bit is set, indicated by a black square. A pointer to the field of A is stored in the location of the available word of O. The bottom bit of the pointer to the field of A is also set.

The next object encountered is object B. Upon discovering the reference to marked object O, the available word is checked. Since the bottom bit of the word is set, it can be determined that a chained reference to O already exists. The pointer to the field in A is copied to the field in B, with its bottom bit unset. The available word in O is set to point to the field of B, with its bottom bit set. The same process is applied to object C. The result of these operations is shown in Figure 2.2(b).

The scan now encounters object O. Since O is marked, the available word is checked. The set bottom bit indicates that references are chained. At this point it is known to which address object O will be moved, since all objects to be compacted before it have been scanned. The chain of references can then be traversed, with each field updated to point to $O'$, the location to which O will move. The end of the chain is reached when a field contains a word with its lowest bit set, which is the data from the available word of O. This data is replaced.

The scan continues, and encounters object X. X is chained to O in the same way as before, as are objects Y and Z. At the end of the first scan, the heap is as shown in Figure 2.2(c).

The second scan compacts the objects. Since at this point objects A, B and C no longer point to object O, the first object of interest is object O. Since it is marked, it is moved to

location $O'$. The chained references to objects X, Y and Z are updated in the same way as before. The final outcome is shown in Figure 2.2(d).

### 2.2.1 Generational Collectors

Generational collectors are a class of garbage collector that exploit the age of objects to improve performance. They are based on the *generational hypotheses*: the *weak generational hypothesis* states that most objects die young [6], while the *strong generational hypothesis* states that the older an object is, the less likely it is to die. Generational collectors have been shown to generally outperform their non-generational counterparts [7], and are today the most commonly used type of collector for the majority of systems.

#### Generational Collector Design

The heap managed by most Generational collectors is divided into two or more spaces, or generations. The simplest generational collectors have a small *nursery* where object 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 the majority of allocation in the collector 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.

Aside from the basic mark and sweep or copying collectors, elaborate mature space management strategies have been proposed [8] [9] [10]. These may use multiple older generations in order to better classify objects by age [11]. Others attempt to allocate connected objects together to improve cache performance.

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 minor generational hypothesis), this *minor* collection frees more space than a collection of an equivalent-sized region of a non-segregated heap.

When performing a minor collection it is desirable to scan as little of the heap as possible. Scanning the mature space would invalidate many of the advantages of collecting only the nursery. However, it is possible for objects within the mature space to refer to objects in the nursery. Without taking these references into account, live objects may be considered garbage, leading to dangling references.

To maintain correctness while eliminating the requirement to scan the whole heap, generational collectors maintain *remembered sets*. These are lists of the locations of pointers in the mature spaces to the nursery. If these sets are complete, it is not necessary to scan the mature space; the collector knows where any relevant pointers are located. Traversing the remembered sets generally involves inspecting far fewer locations than a scan of the heap would. While it is possible for references to point from the mature space to the nursery, the common case is for references to go from new objects to old, rather than the other way.

The data in the remembered sets can be kept accurate through virtual memory mechanisms (trapping page faults), or by use of a *write barrier*. This is a small piece of code that executes whenever a pointer to a location in the heap is overwritten. The barrier is inserted by the compiler, and is transparent to the programmer. A typical generational barrier will include a check to determine if the location being written to is inside the mature space, and if the destination of the pointer is in the nursery. If so, it will record the pointer location in the remembered set to be scanned by the garbage collector.

Write barriers can be made more efficient by using an inlined “fast path” for the common case where no remembered set entry is required [12]. When a pointer crosses the generational boundary, an out-of-line “slow path” is executed.

## 2.3 Appel's Generational Copying Collector

Appel's generational copying collector uses a variable-sized nursery and a single mature generation [13]. 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, similar to a semi-space collection.

### 2.3.1 Heap Layout

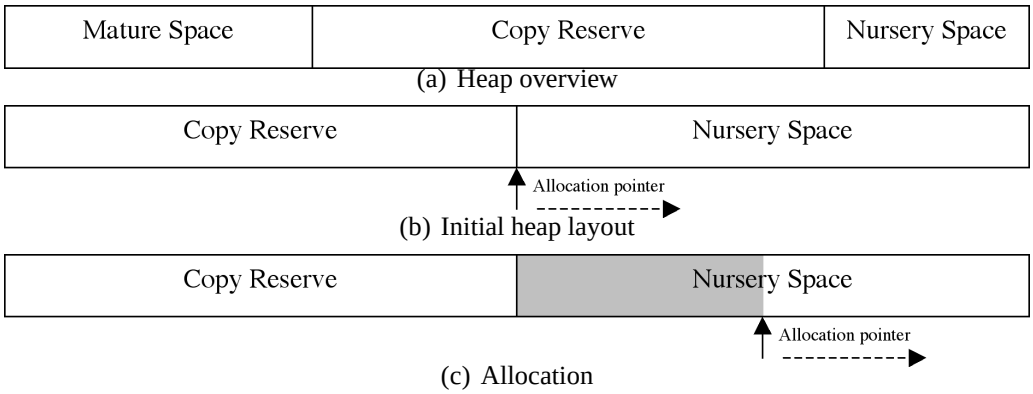



Figure 2.3. Heap layout in an Appel-style collector

The heap layout for Appel's collector is shown in fig 2.3(a). The copy reserve size is equal to the sum of the nursery size and the mature space size.

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. This is shown in Figure 2.3(b).

All allocation in the Appel collector is performed in the nursery. As shown in Figure 2.3(c), allocation is performed sequentially by a bump pointer.

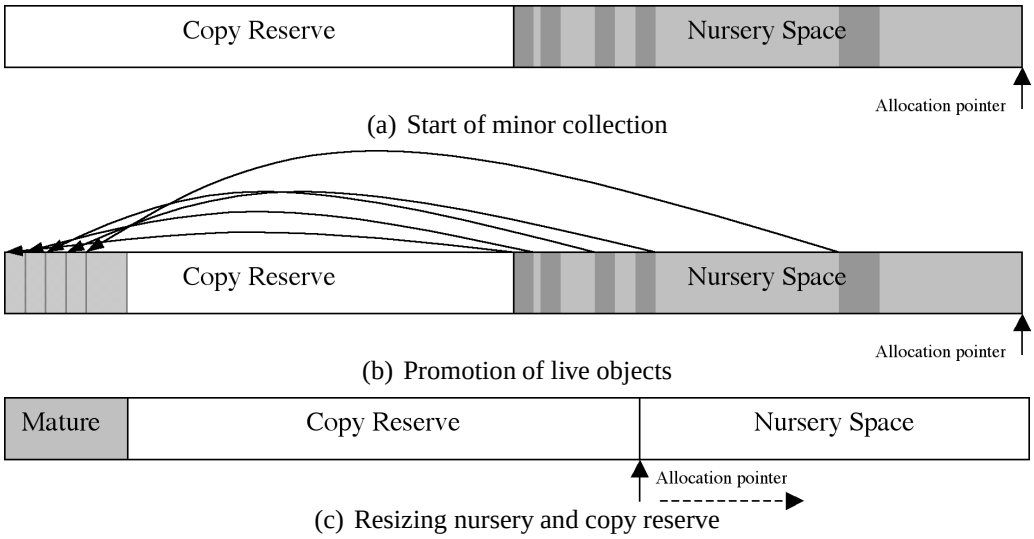



Figure 2.4. Minor collection in an Appel-style collector

### 2.3.2 Minor Collections

When the allocation pointer reaches the end of the nursery, a minor collection is required. In this process, all reachable objects (indicated in dark gray in Figure 2.4(a)) are located. They are then transferred into the copy reserve, stored sequentially, as in Figure 2.4(b). The space occupied by these objects becomes the mature space. The nursery is resized to accommodate the mature space, again leaving a copy reserve equal to the sum of the nursery and mature spaces. The result is shown in Figure 2.4(c).

### 2.3.3 Major Collections

When it is detected that a minor collection will cause the nursery to shrink below a predetermined threshold, a collection of the mature space is triggered. Figure 2.5(a) shows the heap layout on triggering of a major collection. As before, live objects are shown in dark gray. All live objects from both the mature space and nursery are copied into the reserve, as shown in Figure 2.5(b). The space occupied by these objects becomes the mature space and is moved to the front of the heap. The nursery is then resized to account for the new mature space size, as shown in Figure 2.5(c).

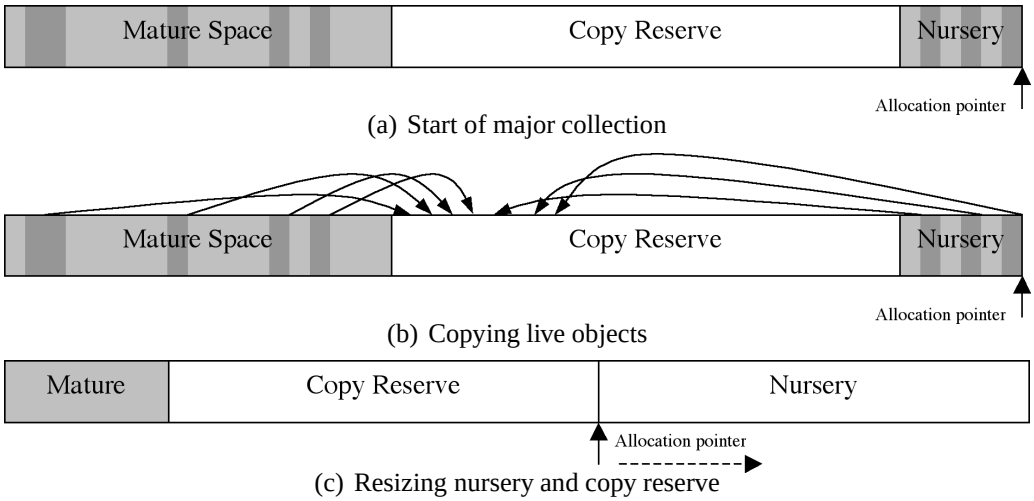



Figure 2.5. Major collection in an Appel-style collector

## 2.4 Jikes RVM and MMTk

Jikes Research Virtual Machine [14] is an open-source, high performance Java virtual machine. It was initially developed at IBM, before being released to the research community. Jikes RVM is written in Java.

The Memory Management Toolkit (MMTk) [15] is a portable memory management framework written in Java. MMTk handles all memory-related operations within Jikes RVM. It offers implementations of a series of algorithms, as well as commonly used components designed to facilitate the development of new algorithms. Since all collectors implemented in MMTk share a common underlying framework, performance comparisons can be made isolated from differences between virtual machine implementations.

### 2.4.1 Java Language Extensions

Since Jikes RVM and MMTk combine to form a complete virtual machine written in Java, there is a need for extensions to the Java language to support some essential functions. For example in MMTk it is necessary to access pointers, something which is not permitted in Java. To overcome these limitations, certain classes are defined that the

Jikes RVM compilers convert into native operations. Without these extensions, it would not be possible to write a garbage collector in Java.

### 2.4.2 Compilation Framework

Jikes RVM uses a variety of compilation strategies depending on the level of optimization required. The most aggressive of these is the adaptive framework.

The adaptive compiler determines through sampling which methods are "hot", that is which methods are called most often. Based on the result of this sampling, some methods are optimized further than others [16]. This means, however, that the performance of Jikes RVM at the highest optimization level is non-deterministic, since differences in sampling will cause different methods to be optimized.

### 2.4.3 MMTk Design

MMTk provides a framework of building blocks upon which memory management algorithms can be implemented. It also supplies implementations of common garbage collection algorithms, allowing comparisons to be made without necessitating re-implementation. Additionally, since all systems build upon a common framework any measured differences in performance can be attributed to variations in algorithm, rather than details of underlying implementation.

A garbage collector in MMTk is defined by an implementation of a *plan*. This class determines, among other things, the spaces in the heap, the allocators used for each, and the strategy by which they are garbage collected. Plans generally build upon a template which provides basic functionality for a class of collector, such as reference counting, tracing or generational.

The MMTk heap is divided up into various *spaces*, determined by the plan. All implementations share some common spaces: the *boot space* stores precompiled classes and data structures established when the virtual machine is built; the *immortal space* stores objects that are live throughout the execution of the system, and as such is never collected;

the *metadata space* holds temporary memory management-related objects, and is not used in determining liveness inside the heap; and the *large object space*, or LOS, is used to store objects larger than a certain threshold. The LOS is managed by a free-list allocator. The advantage of storing large objects separately from the rest of the heap is that it removes the need to move large objects when a copying collector is used.

Further spaces are defined inside the plan. MMTk provides implementations of commonly-used types of space such as a copying space, mark sweep space or reference counting space. Thus in order to implement a simple semi-space copying collector, it is necessary for the plan simply to create two copying spaces and determine the criteria under which objects are transferred from one to the other.

# 3 AN IMPROVED GENERATIONAL COPYING COLLECTOR

Generational collectors are today used in a wide range of settings, from production virtual machines to C/C++ compilers . They have been shown to offer significant performance advantages over their non-generational counterparts. In this section a design is presented for a new generational copying collection algorithm. This collector aims to improve the performance of a standard copying collector by reducing the overhead required for the copy reserve. By reducing the copy reserve overhead it aims to decrease the collection frequency, thus decreasing execution time. It combines a generational copying collector with a compacting collector.

## 3.1 Motivation

When selecting a generational garbage collection algorithm, certain limitations must be taken into account.

Copying collection algorithms offer improved spatial locality, since data is copied in traversal order. This can improve cache behavior and lead to reduced paging. Additionally, fragmentation is not a concern since the heap is compacted whenever objects are copied. However, copying collectors come with a significant space overhead. Since in the worst case all objects may have to be copied, a significant portion of the heap must be kept as a copy reserve. This increases the frequency of collections and limits the minimum heap size in which the collector can operate.

Mark and sweep algorithms do not have the limitation of requiring a copy reserve. As a result, a mark and sweep collector can run in a smaller heap than a copying collector. Additionally a mark and sweep collector will require fewer garbage collections when heap sizes are equal. However, mark and sweep collectors do not gain the spatial locality nor compaction benefits of a copying collector.

The collector presented in this work aims to address these limitations, providing the benefits of a copying collector while avoiding the overhead of the copy reserve. It is based on the generational copying collector first presented by Appel [13].

## 3.2 Algorithm Overview

The copy reserve in Appel's generational copying collector consumes half the heap but, as shown, is rarely fully used. We introduce a method of reducing this space overhead substantially.

We determine the copy reserve to be a percentage of the maximum possible reserve. It can be individually specified for the nursery and mature space to allow tailoring for individual workloads. For example if a benchmark is found to have an unusually high nursery survival rate, the nursery copy reserve can be increased to compensate.

In the vast majority of cases, a well-chosen set of copy reserve sizes 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 overflow the allocated copy reserve. In this case, a secondary compaction technique is used. Rather than copying survivors from the nursery to the mature space, or from the old mature space to the new mature space in 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.

## 3.3 Fallback Technique

Should the copy reserve prove to be insufficient during a collection, a compaction algorithm is activated. Since this compaction phase occurs only during a garbage collection, and when no further copy reserve space remains, it is vital that no allocation occur until space can be made available. The compacting algorithm is designed to operate only over memory already assigned to the process, ensuring that the maximum heap size is never violated.

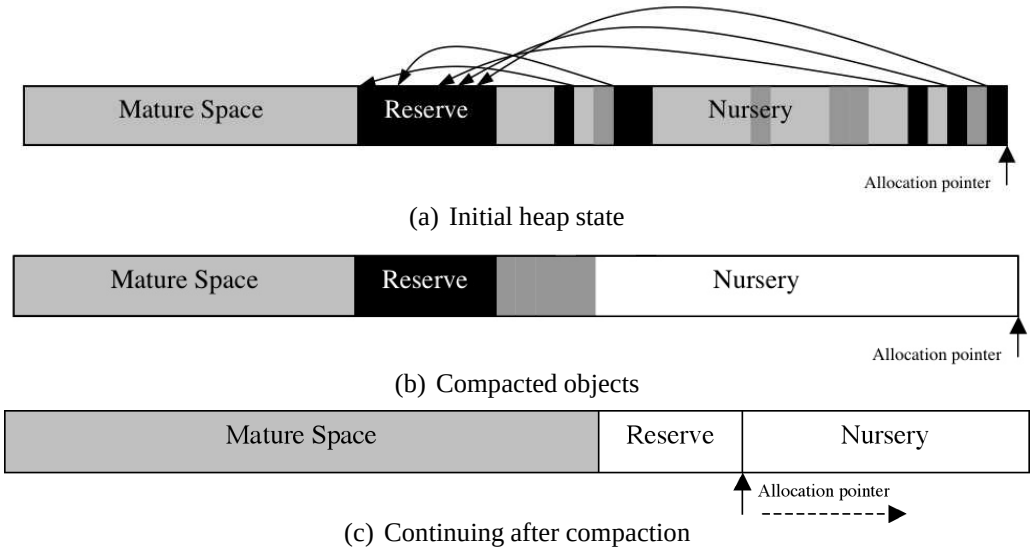



Figure 3.1. Compacting collection

The state of the heap upon entering a compaction phase is shown in Figure 3.1(a). The black objects in the nursery have been forwarded to the reserve, which is now full. The remaining dark gray objects in the nursery are live, but have no place to be copied.

The remaining live objects in the nursery are then compacted to the front of the nursery, as shown in Figure 3.1(b). The white space in the nursery is now free. 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 3.1(c).

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

## 3.4 Related Work

Velasco *et al* make use of the same observation as in this work [17]. They determine that the survival rate of collections is far below the space normally allocated as copy reserve in an Appel style collector.

Their work differs from the present work in several ways. They adopt a strategy of dynamically tuning the nursery copy reserve size, while this work sets mature and nursery copy reserves constant during execution. They suggest several simple heuristics to deter-

mine 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 chapter performs a compacting collection, the earlier work instead relies on the principal of *nepotism*. This refers to garbage objects in the nursery being kept reachable by garbage objects in the mature space. In Figure 3.2(a), several objects 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. The authors 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 3.2(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 3.2(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 3.2(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 3.2(e). Even if the objects retained through nepotism were removed, there would still be insufficient space in the copy reserve.

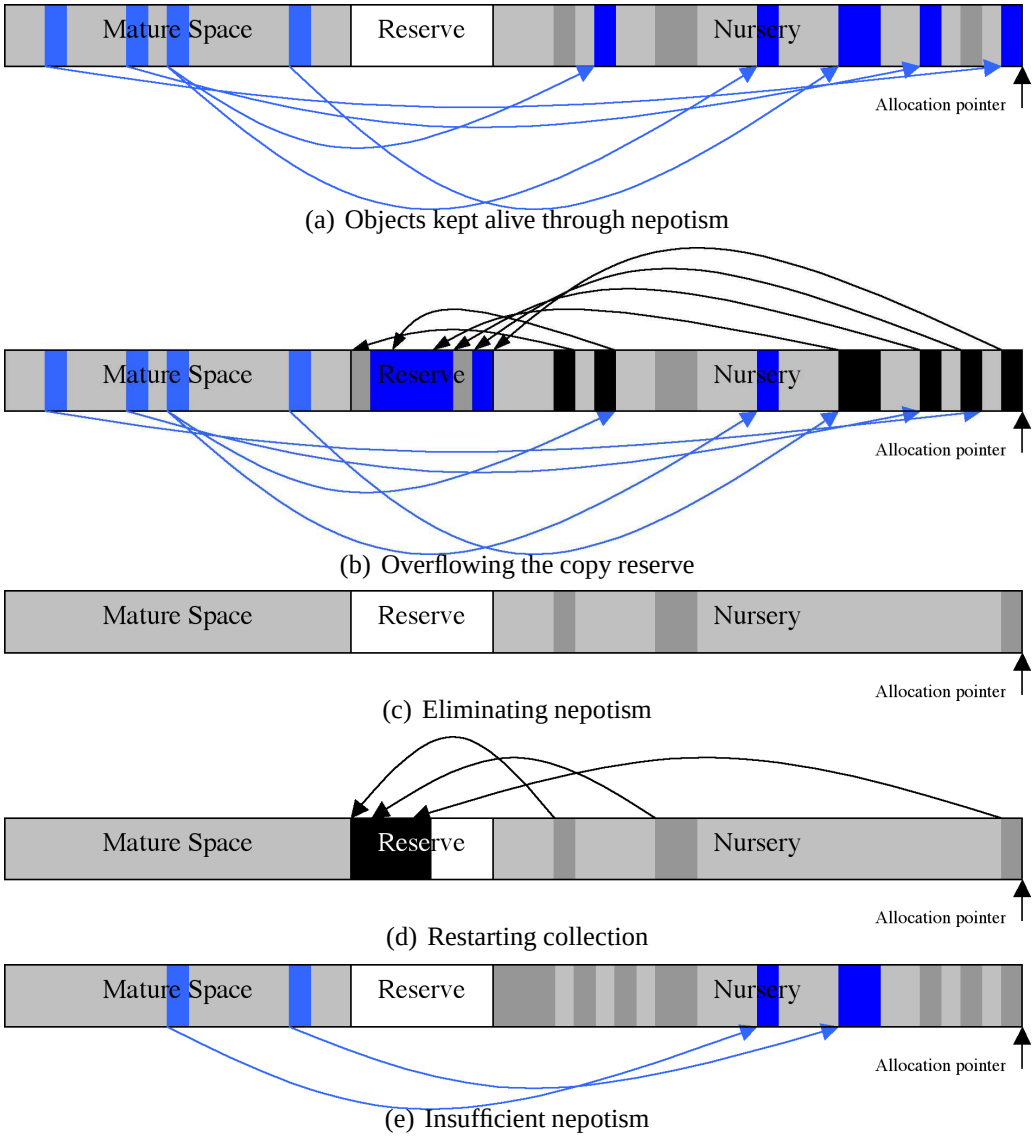



Figure 3.2. Reducing survival rate through nepotism

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 chapter 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. This is described in Section 3.3.

# 4 IMPLEMENTATION

This chapter describes an implementation of the collector proposed in Section 3. The implementation was based on the Appel-style generational copying collector supplied with MMTk [15] running with Jikes RVM [18]. It makes use of Jonkers' reference chaining algorithm outlined in Section 2.2. Both MMTk and Jikes were modified during the implementation.

## 4.1 Heap Layout

Figure 4.1 shows the layout of the heap for this collector. It follows a similar design to that seen in Figure 2.3(a), with additional spaces due to MMTk 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 space 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.

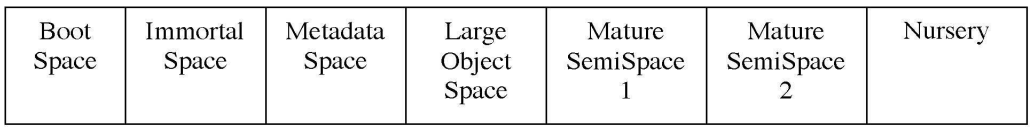



Figure 4.1. Generational copy/compact collector's heap layout

### 4.1.1 Boot and Immortal Spaces

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.

### 4.1.2 Large Object Space

The large object space is managed by Baker's treadmill collection algorithm [19]. 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.

### 4.1.3 Mature Space

The mature space comprises two copy spaces. Apart from brief periods during garbage collection, only one semispace 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 semispace 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 oldspace. In the algorithm outlined in Section 2.3, upon evacuation of all nursery and mature space objects into the copy re-

serve, the memory containing live objects is remapped to the start of the heap. This is done in order that the nursery can 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.

### 4.1.4 Nursery Space

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.

## 4.2 Triggering Compaction

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.

## 4.3 Mark Stage

Upon the triggering of a compacting collection, the behavior of object tracing changes. Previously, an object, upon being found to be reachable, 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 object 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. Section 4.5 details the need for this count.

## 4.4 Scanning

The compaction follows Jonkers' algorithm described in Section 2.2. 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 scan 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 scanning the nursery. This is a major improvement over the expense of scanning every space in the system.

The initial sweep in a major collection, however, requires that the majority of the heap be scanned. 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 in 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 go to an object header. Pointers that do not resolve to an object header are dangling references, and can lead to errors as described in Section 2.1.1.

Since the MMTk collectors do not make use of scanning, they do not have this requirement. 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.

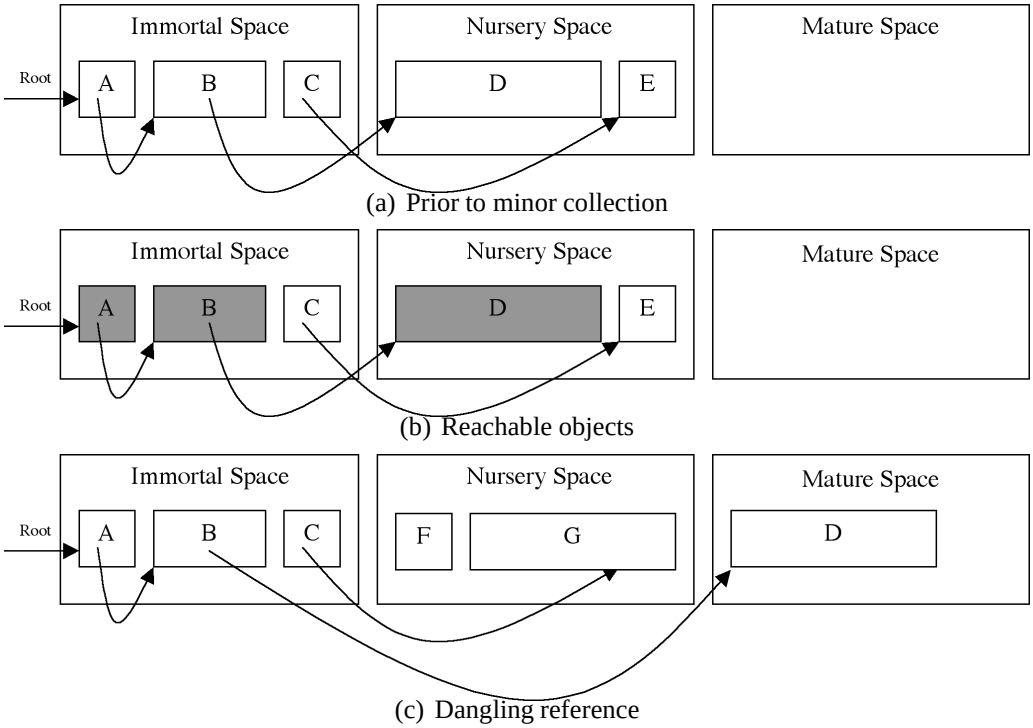



Figure 4.2. Creation of dangling reference

Within the immortal and boot spaces of MMTk it is possible for dangling references to be created. Figure 4.2(a) shows part of an object graph that spans the immortal, nursery and mature spaces. In the mature space, Objects A and B are reachable from a root, while object C is not. Figure 4.2(b) shows a minor collection; objects A, B and D are reachable, while objects C and E are not. However, since the immortal space is not collected, objects

A, B and C do not move. Object D is promoted to the mature space, and B's pointer updated. Object E is reclaimed as garbage.

Figure 4.2(c) shows the state of the heap at the start of the next minor collection. New data has been allocated to the nursery. If this next collection uses the standard tracing mechanism, the traversal will occur as before. However, should the collection use compaction, a sweep of the heap will be required. When the immortal space is swept, objects A and B will be encountered and their pointers processed. However, when object C is reached an error will occur. Its pointer is no longer valid since the object it referred to (object E) no longer exists. Worse, it now points to the middle of object G, meaning that random data will be interpreted as an object header.

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.

The second sweep, used to compact objects and update backward pointing references, need only be performed in spaces being compacted. This means that in a minor collection, only a sweep of the nursery is required. In a major collection both the mature space and nursery must be swept. This is because by this point in the algorithm all pointers have either been chained or updated. Only the objects to be compacted must be scanned in order to enchain any remaining pointers and to perform the compaction.

## 4.5 Compaction

The addresses to which objects will be relocated must be calculated during both the first and second scans in the compaction algorithm. This involves maintaining pointers for the nursery and, if required, the mature space which are incremented upon scanning an

object which will be compacted. Care must be taken to ensure that alignment constraints are maintained, and that compacted objects do not overwrite embedded metadata.

The updated values of references must be adjusted to account for the fact that compacted data is remapped in the final stage of the collection. Thus an object compacted to the front of the nursery space will eventually be relocated to the end of the mature space. The final address can be easily calculated using the known virtual address ranges of the spaces, and the current location of the mature space bump pointer.

This calculation is complicated slightly during major collections. This is because two compacted regions, the nursery and old mature space, require relocation. It is necessary to know the size of one of the relocated spaces in order to calculate the address to which the second space will be remapped. The information for this calculation is gathered during the marking phase, where the total size of marked mature objects is maintained. From this it is possible to predict the address range required by the compacted mature space, and by extension the location to which the nursery objects will be remapped.

The compaction is performed in the second scan of the heap. Objects are compacted within their own spaces. The location to which objects are compacted is determined in the same way that the pointer was maintained in the previous step, without the adjustment for remapping to the mature space. This way predicted addresses correspond to actual addresses.

## 4.6 Block Copy

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. An additional requirement is that no allocation occur during the remapping, since this 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.

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 two physical addresses will not be mapped to a single virtual address. To avoid this, memory is unmapped whenever it is not in use. While this adds some overhead to benchmarks elapsed time, it is a more honest approach, since the amount of memory mapped is never more than the maximum heap size.

# 5 EXPERIMENTS

This chapter presents the results of running the new garbage collection algorithm with a series of benchmarks.

## 5.1 Platform

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.

## 5.2 Benchmarks

The performance of the new garbage collector algorithm was tested using several benchmarks from the SPECjvm98 suite [20].

Of the full SPECjvm98 suite, several benchmarks were considered to be uninteresting from the perspective of garbage collection. For example, _222_mpegaudio requires virtually no garbage collection activity, meaning that differences between algorithms would be insignificant. Several other benchmarks, such as _227_mtrt and _209_db have very short object lifetimes, meaning that only nursery collections are required. The benchmarks outlined below have sufficient allocation and garbage collection requirements to make them interesting candidates for this performance analysis.

## 5.3 Metrics

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.

### 5.3.1 Methodology

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.

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*.

### 5.3.2 Traditional Collectors

Data are reported for each interesting benchmark when run using generational copying and generational mark and sweep collectors. These graphs show a line plotting the mean elapsed time of the benchmark, and a set of bars showing the number of major and minor garbage collections. The error bars on the elapsed time graph represent confidence intervals for a 90% confidence value.

Study of the elapsed time data for GenCopy and GenMS collectors shows that the times of each collector/benchmark pair tends to stabilize as the heap size grows. The point of inflection in the generational copying graph at which this begins to occur was chosen as the heap size for comparison to the new algorithm. The reasoning was that the point of

inflection was the optimal heap size for the generational copying collector. Comparisons at this heap size would therefore not unduly disadvantage the existing collectors.

### 5.3.3 Variable Copy Reserve

Results for GenCC are given for a varying copy reserve size. Here, the heap size remains constant, having been arrived at as indicated above. Since the copy reserve is split into mature and nursery components, presentation of this data necessitates several graphs. Each represents data collected for a fixed heap size, a fixed nursery copy reserve and a variable mature copy reserve.

On each graph the elapsed time of GenCC is shown as a continuous black line. Each measurement on the line is accompanied by a set of error bars which represent a 90% confidence interval. Also shown by red and green continuous lines are the elapsed times for the traditional GenCopy and GenMS collectors respectively. These lines represent the times for the traditional collectors at this heap size; they are straight because these collectors do not allow variable copy reserves. The dashed lines accompanying each of these lines represent a 90% confidence interval.

The bars at the bottom of each graph show the number of garbage collections performed by the copying/compacting algorithm. The x-axis for this data is the same as for the elapsed time, so the bars directly relate to the data points in the line above. The bars categorize collections into minor and major, as well as compacting and normal.

Graphs are not presented for all nursery copy reserve sizes on all benchmarks. Generally the extreme values in the nursery copy reserve scale show very poor performance since these copy reserves are unreasonable. Efforts have been made to ensure that the graphs shown are interesting yet representative.

### 5.3.4 Variable Heap Size

Additional graphs show the performance of all three collectors over a range of heap sizes. In the variable heap size graphs, nursery and mature copy reserves remain constant.

Each is fixed at 20% which the variable copy reserve graphs show to be a reasonable value. It is low enough that the space benefits of reducing the copy reserve can be seen, while high enough that compaction is not required. These copy reserve sizes would therefore be likely to be used in any real-world implementation of the generational copying/compacting collector.

The variable heap size graphs show the elapsed times for all three collectors. The x-axis represents the heap size. The black line represents the new collector, while the red and green lines represent the generational copying and generational mark and sweep collectors respectively. The error bars represent a 90% confidence interval.

The bar chart below the elapsed time plots shows the number of garbage collections performed by the new algorithm, broken down as in the previous variable copy reserve graphs.

## 5.4 Results

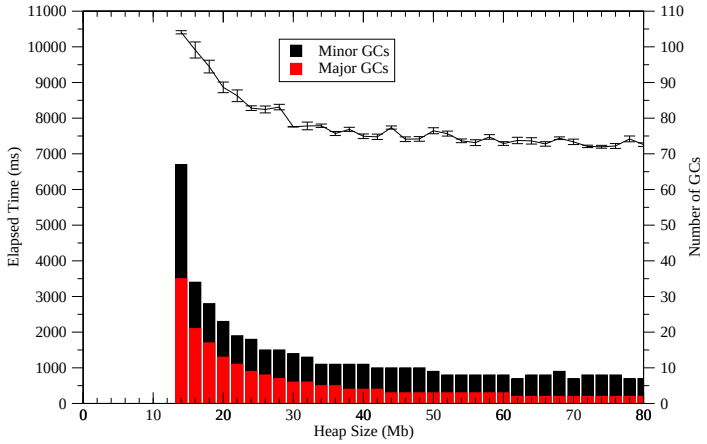
### 5.4.1 _201_compress

_201_compress performs a modified LZW compression algorithm over a fixed input. The algorithm scans through the data set in order to identify common strings, and replaces them with codes. It then performs the decompression process.

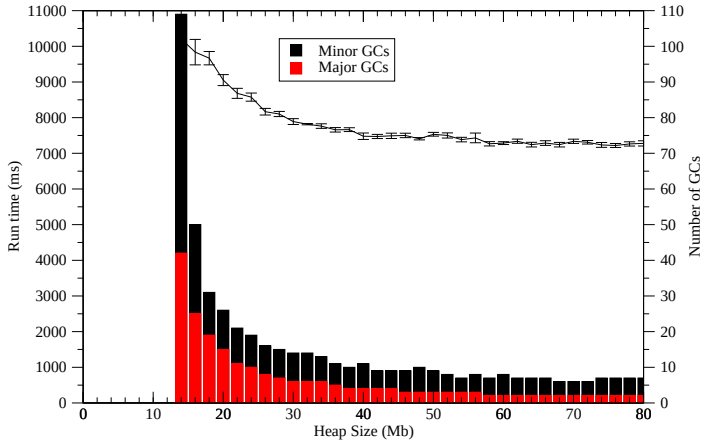
#### Traditional Collectors

Of the benchmarks presented, compress has the highest fraction of major collections. This indicates that it has a low mature mortality rate. The slow rate at which mature objects are collected forces more frequent major collections.

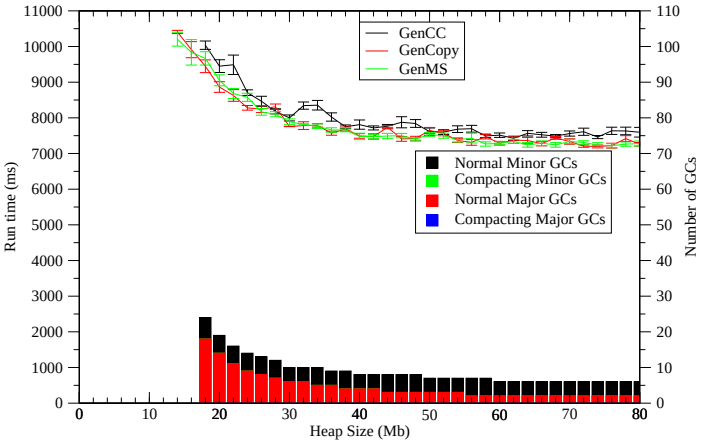
Figure 5.1(a) shows the elapsed time and number of collections required for compress when using GenCopy. As can be seen, at small heap sizes almost half of the collections are major. This ratio drops to a quarter at larger heap sizes. A similar trend can be seen in Figure 5.1(b), where the benchmark is run using GenMS.




(a) GenCopy




(b) GenMS




(c) GenCC

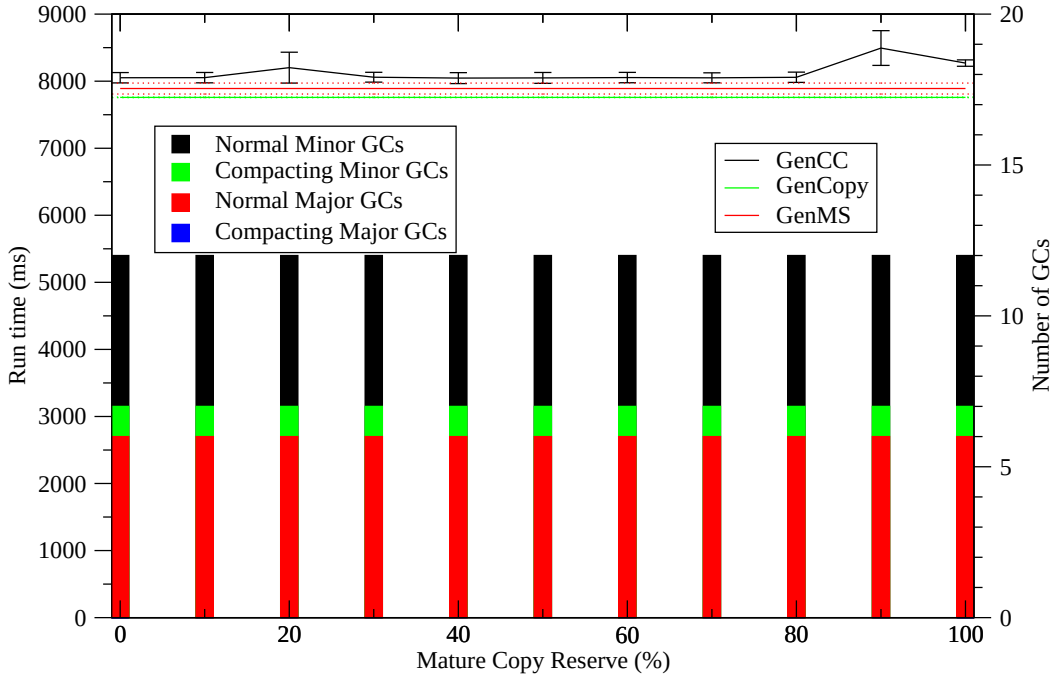
Figure 5.1. _201_compress

Variable Copy Reserve

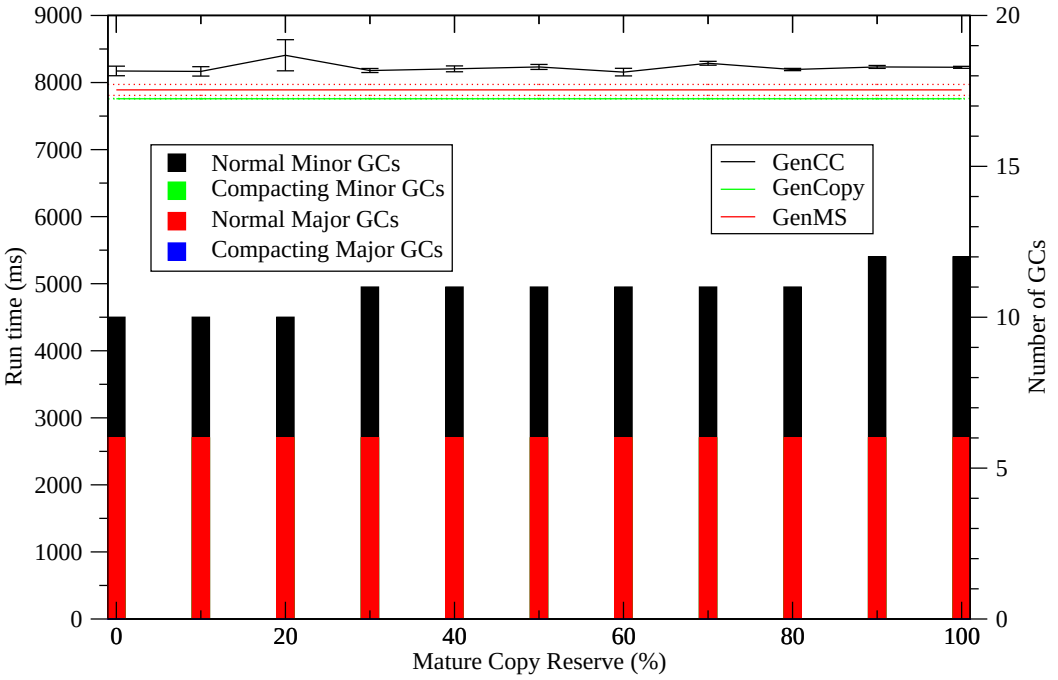
Figure 5.2(b) shows the elapsed time and number of garbage collections required for the benchmark when a 20% nursery copy reserve is used. As can be seen, the performance of GenCC is poorer than that of the standard generational copying and generational mark and sweep collectors. The reason for this is the higher garbage collection survival rate, as described above. The advantage of the copying/compacting algorithm lies in its ability to manage the available space better than a standard copying collector. By doing so it reduces the number of collections needed. However, if a benchmark's live set makes frequent collections unavoidable, the benefit of the copying/compacting scheme is lost. All that remains in this case is the additional overhead imposed by the algorithm.

When the nursery copy reserve percentage is varied, similar performance lags are seen. However, Figure 5.2(a) shows an unusual result. While the elapsed time performance of the copying/compacting collector is still worse than the traditional collectors, the difference is less than when the 20% nursery copy reserve is used. It can be seen, additionally, that several nursery compacting collections are performed. Normally, this would be expected to decrease performance. However, in this case the space saved by the elimination of the nursery copy reserve leads to fewer major collections. This trade-off improves performance over situations where the compacting collector is not triggered.

While it might be expected that setting the nursery copy reserve to 0% would lead to every minor collection requiring compaction, it can be seen that this is not the case. The reason for this is that a small minimum copy reserve of eight pages was allocatated as part of the implementation. This copy reserve guarantees that essential VM data structures and GC metadata are copied before any compaction is performed. In cases where the copy reserve is set to 0% yet a compaction is not required, it is simply that all live nursery objects fit into these eight pages.




(a) 0% nursery copy reserve




(b) 20% nursery copy reserve

Figure 5.2. _201_compress with 30Mb heap

Variable Heap

The GenCC algorithm performs particularly poorly at the 30Mb heap size. However, in general the characteristics of _201_compress disadvantage the copying/compacting collector, meaning that the traditional collectors outperform it.

Figure 5.1(c) shows the elapsed time and number of garbage collectors for _201_compress when the heap size is varied. The nursery and mature space copy reserves for GenCC were fixed at 20%. As can be seen, the new garbage collection algorithm does not perform as well as the other two for this benchmark. This is due to the object lifetime behavior described above.

It is worth noting that, on average, the new algorithm does not perform substantially worse than the traditional algorithms. There are several heap sizes around 30Mb, however, where performance drops significantly. The graphs in this section represent data gathered at a heap size of 30Mb.

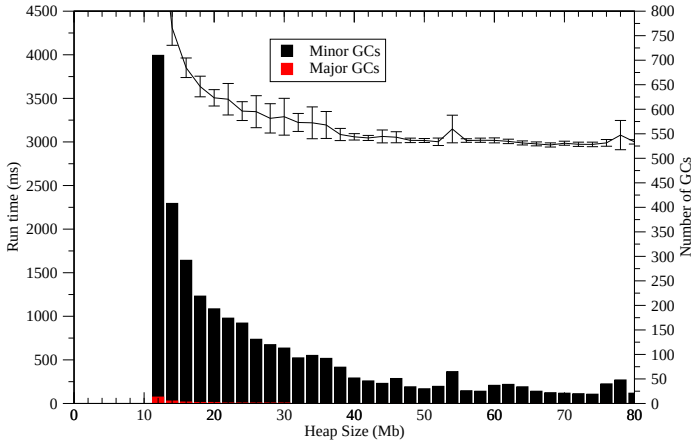
### 5.4.2 _202_jess

_202_jess is the Java Expert System Shell – an expert system shell solver. It iteratively applies a growing set of rules to a problem statement until it reaches a solution.

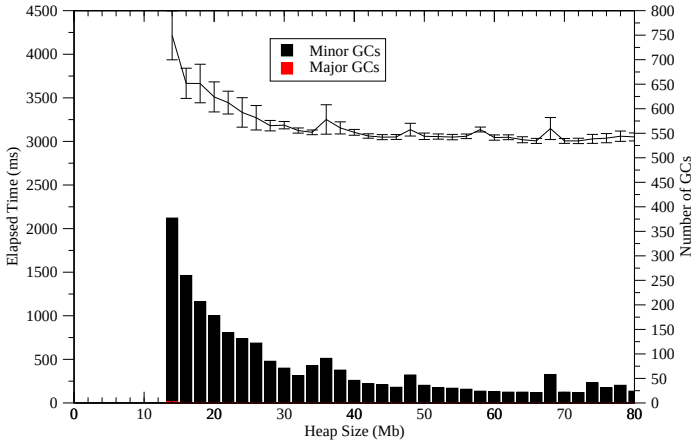
Traditional Collectors

Figure 5.3(a) shows the elapsed time and number of garbage collections for jess when run using GenCopy. As can be seen, there are very few full collections. For heap sizes greater than 30Mb, no full collections are required. Figure 5.3(b) shows the same result. Using GenMS, no full-heap collections are required once the heap size grows beyond 20Mb.

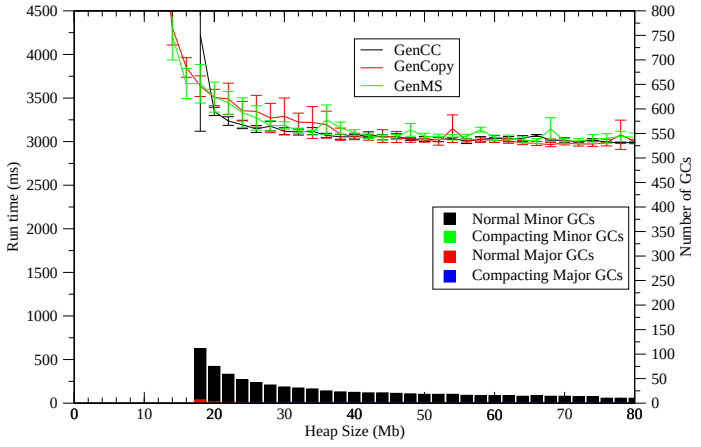
This indicates that very few objects are promoted to the mature space. This is due to very short life spans of objects; the vast majority are garbage before the first minor collection. This property would suggest that the copying/compacting algorithm should




(a) GenCopy




(b) GenMS




(c) GenCC

Figure 5.3. _202_jess

perform well with the jess benchmark. Since the space required for the copy reserve is smaller, the nursery can grow to be larger. This gives the already short-lived objects more time in which to become unreachable, and as a result will allow more garbage to be collected in each minor collection.

### Variable Copy Reserve

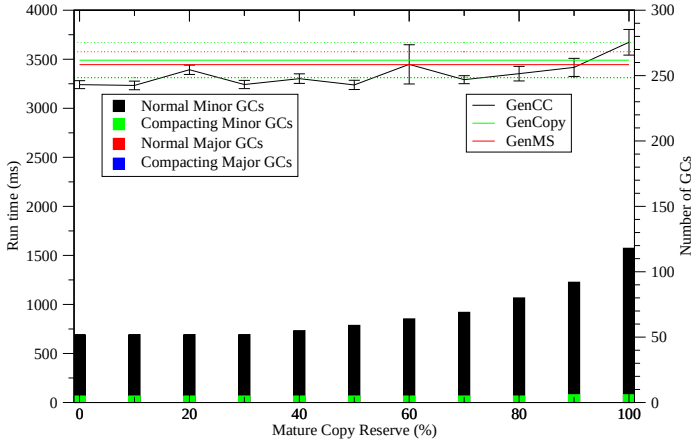
As expected, the high infant mortality rate of _202_jess leads to performance improvements in the new garbage collection algorithm over the traditional methods. This is most noticeable at small copy reserve sizes, since they allow the nursery to grow to the largest size.

Figure 5.4(a) shows the performance of GenCC when the minimum nursery copy reserve is used. This consists of eight pages, used to copy VM and GC related objects. It is noticeable that these few pages are often sufficient to perform a minor collection. This can be seen by the fact that not all minor collections require compaction.

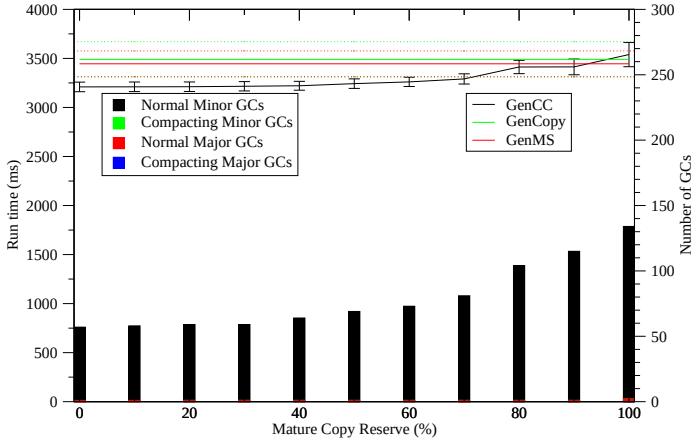
For small mature copy reserve sizes (up to 70%), GenCC outperforms both traditional collectors. It is important to note that this is despite the fact that many of the minor collections are compactions, meaning that they are less efficient than the normal copying nursery collection.

Once the nursery copy reserve is increased to 20%, as shown in Figure 5.4(b) there are no more compacting collections. As before, when the copy reserve for the mature space is also small, the performance gains achieved by the new algorithm are substantial. Only when the mature copy reserve grows beyond 70% does the performance degrade to that of GenMS. This is because the overhead of the copying/compacting algorithm is greater than that of the other collectors.

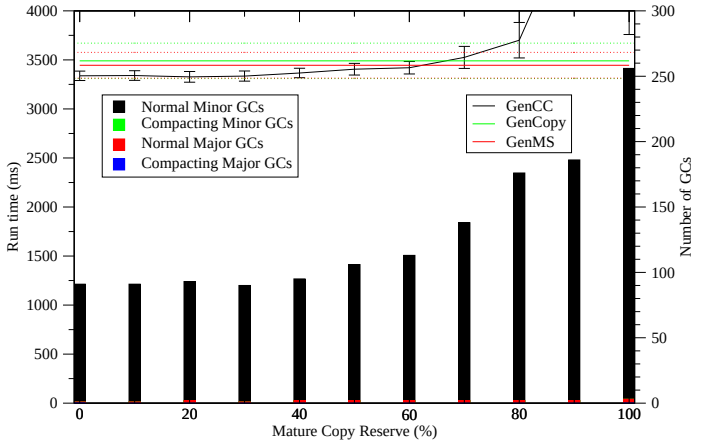
As the size of the nursery copy reserve is increased, the advantage gained by using GenCC diminishes. Once the nursery copy reserve is set at 80%, as in Figure 5.4(c), the performance advantage at small mature copy reserve sizes becomes statistically insignif-




(a) 0% nursery copy reserve




(b) 20% nursery copy reserve




(c) 80% nursery copy reserve

Figure 5.4. _202_jess with 22Mb heap

icant. When both the nursery and mature space copy reserves are large, the performance gains from effective use of space are outweighed by the overhead of the new algorithm.

#### Variable Heap Size

Figure 5.3(c) shows a comparison between the three collectors. It can be seen that when the heap size is large there is little difference between the algorithms' elapsed times. This is not surprising, since there are very few garbage collections performed, so the differences in collection times are swamped by the elapsed time of the rest of the benchmark.

At smaller heap sizes (up to 30%) the garbage collector runs frequently enough to differentiate between the algorithms. Here it can be seen that the generational copying/compacting algorithm outperforms both traditional algorithms.

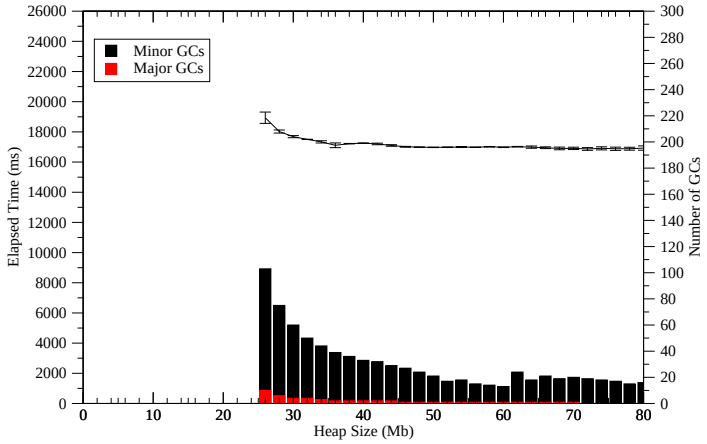
### 5.4.3 _209_db

The _209_db benchmark reads an input file and creates a database in memory. It then performs various actions on this database, including insertions, deletions, sorts and searches.

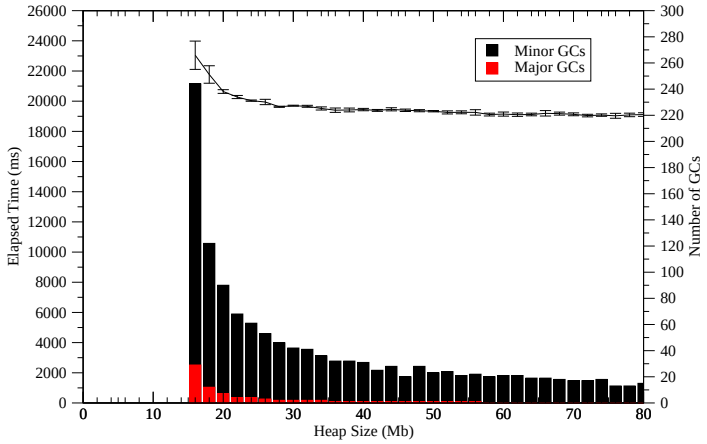
#### Traditional Collectors

Figure 5.5(a) shows the performance of _209_db when run with GenCopy. There are few major collections once the heap size grows beyond 34Mb. Similarly, in Figure 5.5(b), it can be seen that there are no major collections once the heap size grows beyond 20Mb. This indicates that much of the data in the benchmark is short-lived.

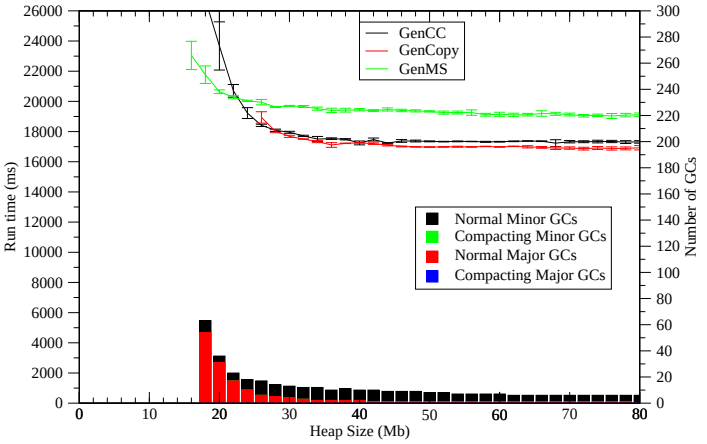
It is interesting to note that GenCopy runs significantly faster for this benchmark than GenMS. This may be because of spatial locality effects in the old space. The database structure is likely to be allocated within a relatively small part of the memory space. Since the copying collector automatically groups objects by connectivity, the structure is likely to fit inside the cache.




(a) GenCopy




(b) GenMS




(c) GenCC

Figure 5.5. _209_db

### Variable Copy Reserve

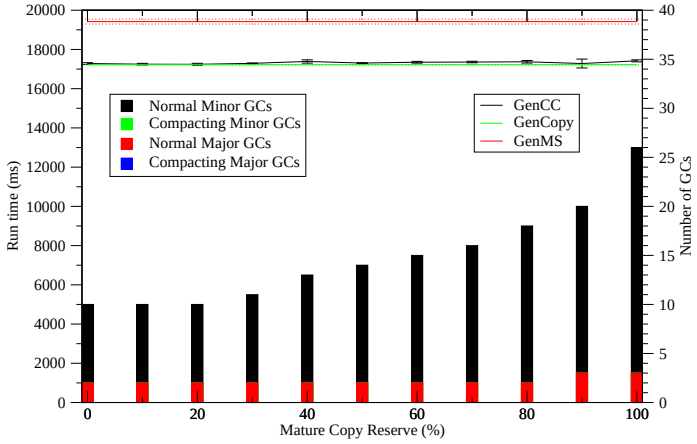
As seen above, GenCopy greatly outperforms the GenMS for the _209_db benchmark. As such, it would be expected that GenCC, which shares many of the characteristics of the generational copying collector, will do likewise. This supposition is supported by Figure 5.6(a). Here, the benchmark is run using a 38Mb heap and a 20% nursery copy reserve. It can be seen that the performance of the new algorithm is almost identical to the traditional generational copying collector.

When the size of the nursery copy reserve is increased to 60%, as in Figure 5.6(a), there is still very little difference between the algorithms. A slight performance decrease is seen in Figure 5.6(c), where the nursery copy reserve is equal to the size of the nursery.

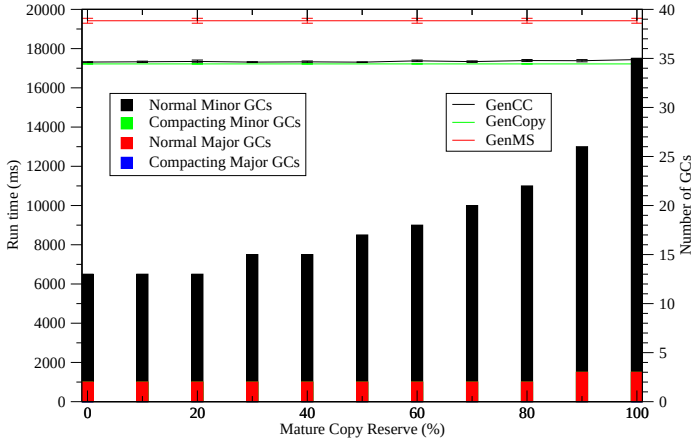
The minor differences observed when varying the copy reserve for this benchmark indicate that allocation and garbage collection do not take up a significant portion of the program's elapsed time. This is supported by the slower performance of the generational mark and sweep collector. The mark and sweep collector does not lay out related data in contiguous memory locations. As a result, it is likely that the performance decrease caused by the loss of spatial locality would outweigh any differences caused by the allocation and garbage collection times.

### Variable Heap Size

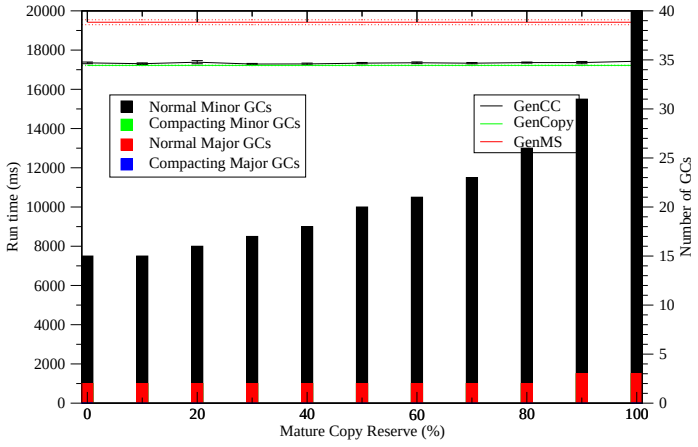
Figure 5.5(c) shows the performance of the three collectors when the heap size is varied. As might be expected, the performance of GenCC is very similar to GenCopy. However, as the heap size grows large, the new algorithm runs slightly slower than GenCopy. This is because the allocation rate is low enough that the overheads of GenCC slightly outweigh the benefits.




(a) 20% nursery copy reserve




(b) 60% nursery copy reserve




(c) 100% nursery copy reserve

Figure 5.6. _209_db with 38Mb heap

### 5.4.4 _213_javac

The _213_javac benchmark runs the Java compiler included with Sun Microsystem's JDK 1.0.2. It performs compilation to bytecode on a series of input files several times.

Of the benchmarks presented, _213_javac exhibits the full effect of the generational hypothesis most clearly. It has a fairly high infant mortality rate, and those objects that do survive tend to live for some time afterward. Additionally, the allocation rate is higher than most of the other benchmarks. It is, from a garbage collection point of view, the most interesting of the benchmarks.

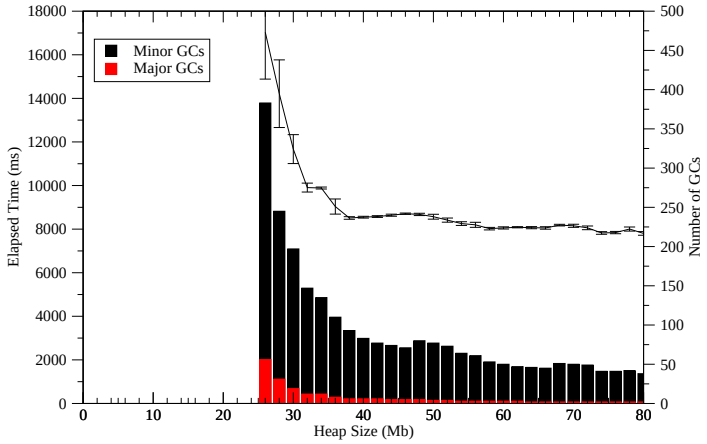
#### Traditional Collectors

Figure 5.7(a) shows the performance of _213_javac when run with GenCopy. There are frequent major collections at lower heap sizes. Beyond 54Mb, the number of major collections drops to three, and then to two beyond 64Mb. This represents substantially more major collections than most other benchmarks. In Figure 5.7(b) there are fewer major collections. This is because GenMS allows more of the mature space to be utilized.

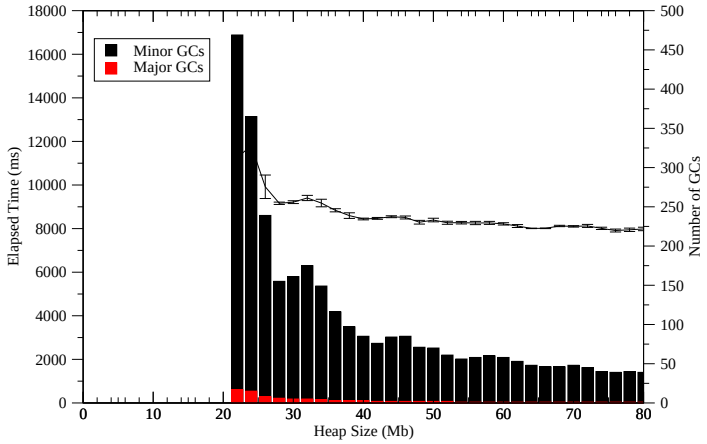
#### Variable Copy Reserve

Since _213_javac contains interesting behavior from a garbage collection standpoint, two sets of results are presented here. The first indicates the performance of the new collector when the heap is relatively small (36Mb), while the second shows performance for a large heap (74Mb).

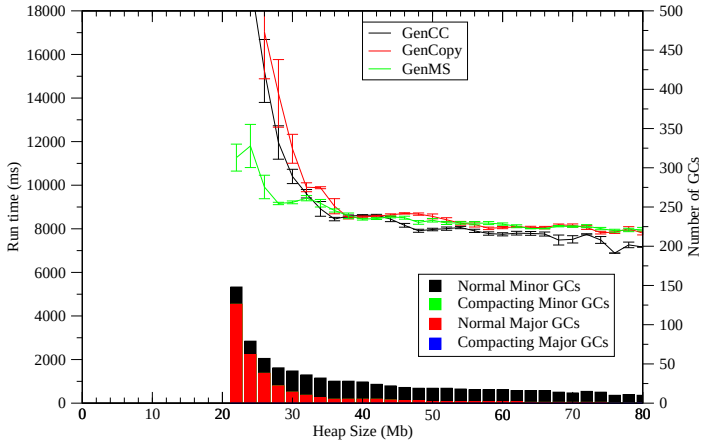
Figure 5.8(a) shows the performance of the generational copying/compacting collector when a minimum copy reserve is used. As might be expected, around half of the collections are compacting collections, and as a result the performance suffers. This is an indication of why it is not advisable to use the copying/compacting collector when frequent compactions are likely to be necessary.




(a) GenCopy

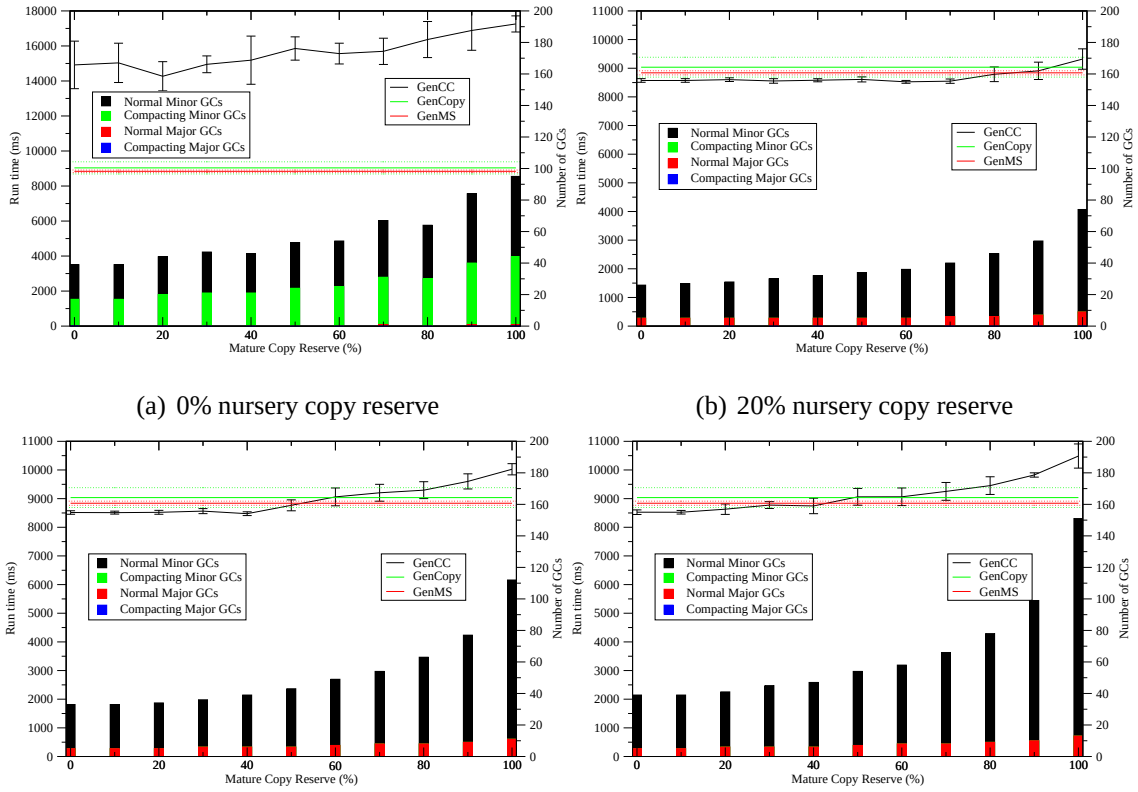



(b) GenMS




(c) GenCC

Figure 5.7. _213_javac




(a) 0% nursery copy reserve

(b) 20% nursery copy reserve

(c) 60% nursery copy reserve

(d) 100% nursery copy reserve

Figure 5.8. _213_javac with 36Mb heap

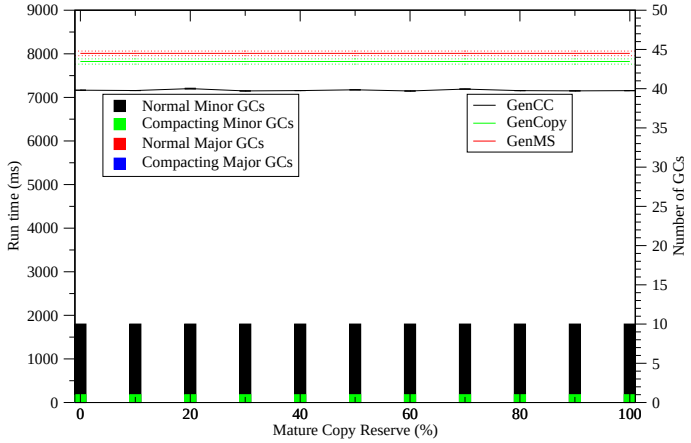
When a reasonable nursery copy reserve is used, as in Figure 5.8(b), it can be seen that the copying/compacting collector outperforms both traditional collectors. For all mature copy reserve sizes below 80% there is a clear advantage to the new algorithm. As in the other benchmarks, once the mature copy reserve becomes too large, the benefits of the new algorithm are outweighed by the performance overheads. It is also useful to note that, even with a copy reserve as low as 20%, no compacting collections are required.

A larger nursery copy reserve is used in Figure 5.8(c). Here it can be seen that until the mature copy reserve reaches 50% there is a clear performance advantage to GenCopy. This shows that, even for comparatively large copy reserves, the copying/compacting algorithm benefits from its more effective space management.

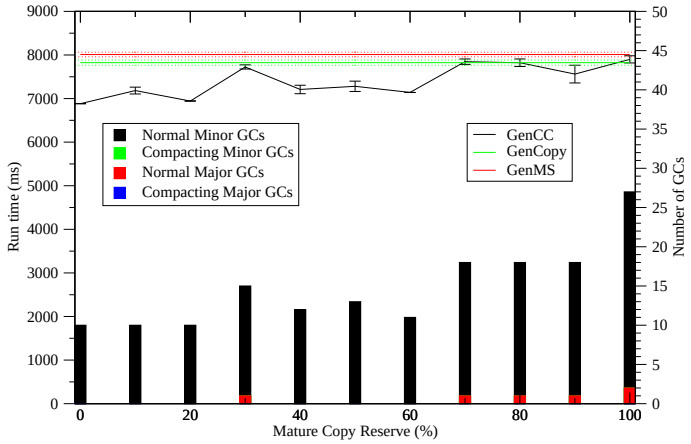
Finally, Figure 5.8(d) shows that, for small mature copy reserves, a performance advantage can still be gained when the nursery copy reserve is at 100%. However, it can also be seen that when both the nursery and mature copy reserves are high (*i.e.*, the collector is behaving as a traditional generational copying collector), the performance is significantly worse than the standard collectors. This is to be expected, since the new algorithm comes with overheads that are not encountered by traditional collectors.

Figure 5.9(a) shows a significant performance gain when the new algorithm is used with a minimal nursery copy reserve. Since the heap is large, there are no major collections required. The entire working set can be collected using minor collections over the expanded nursery. While some of these minor collections involve compaction, the time saved by avoiding major collections more than compensates for the longer minor collections. Since there are no major collections, the mature space never grows. This makes varying the mature copy reserve meaningless, since it is expressed as a percentage of zero.

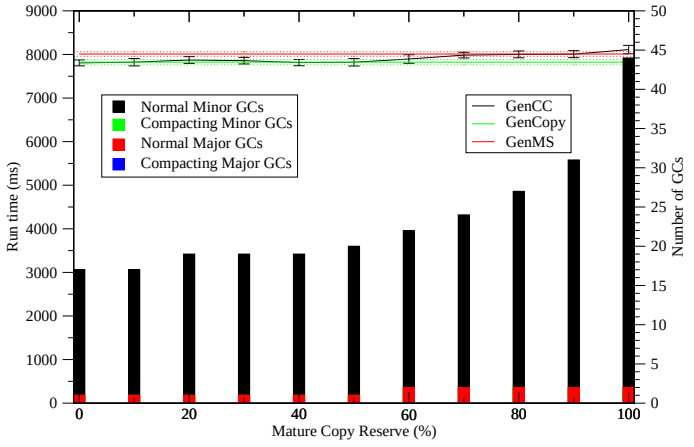
Figure 5.9(b) shows that with a larger nursery copy reserve of 20% GenCC still outperforms the traditional collectors. Since the number of collections is still relatively small, the triggering of an extra full collection can severely affect the elapsed time of the benchmark. This is seen when the mature copy reserve is set at 30%. Here the heap layout causes an additional full GC to be triggered, leading to a substantially slower overall time.




(a) 0% nursery copy reserve




(b) 20% nursery copy reserve




(c) 80% nursery copy reserve

Figure 5.9. _213_javac with 74Mb heap

It is clear from this that the elapsed time of the benchmark is dominated by the mutator rather than the garbage collector. In other benchmarks the inclusion of an additional collection has not caused such a major change in performance. This would be expected, since the size of the heap allows the mutator to perform a great deal of allocation without interruption by the garbage collector.

Finally, Figure 5.9(c) shows that, for large heap sizes, the performance of GenCC is comparable to that of the traditional collectors even with large copy reserve sizes. Since the elapsed time is dominated by the mutator rather than the collector, the cost of the copying/compacting collector is amortized over longer mutator periods, meaning that it has a smaller effect on the total time.

#### Variable Heap Size

Figure 5.7(c) shows that, for the majority of heap sizes, the new generational copying/compacting algorithm outperforms both the GenCopy and the GenMS collectors. This is particularly noticeable at larger heap sizes where, as the heap size increases, so does the advantage to the new collector. The reason for this can be seen by comparing Figure 5.7(c) with Figures 5.7(a) and 5.7(b). GenCC performs fewer than half the number of minor collections, and does not perform any major collections.

It is also interesting that with 20% nursery and mature copy reserves there are no compacting collections. This means that the new algorithm gains all the benefits of the increased usable heap area without the cost of compaction.

### 5.4.5 _228_jack

_228_jack is a Java parser generator based on the Purdue Compiler Construction Tool Set. It is an early version of javacc. The benchmark uses jack to generate the parser for itself multiple times.

### Traditional Collectors

As can be seen in Figure 5.10(a), _228_jack has many minor collections, but very few major collections. This indicates that objects tend to die young, with few being promoted to the mature space. Figure 5.10(b) shows a similar result.

As in the case of _202_jess, the high yield of minor collections should work to the advantage of GenCC. The space available for the nursery will be larger and, as a result, objects will have more time to become garbage.

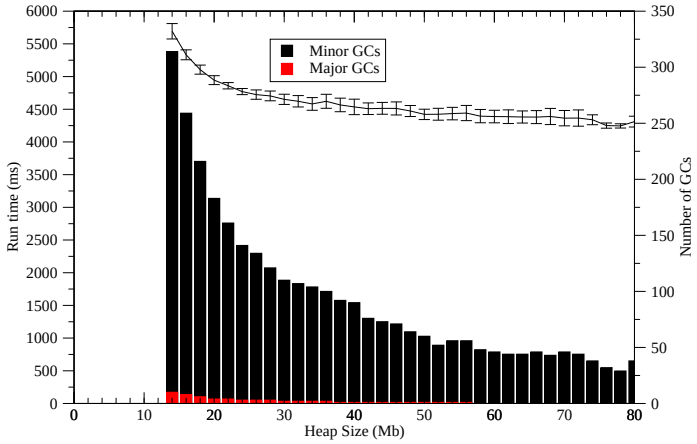
### Variable Copy Reserve

In Figure 5.11(a) it can be seen that the performance of the generational copying/compacting algorithm is better than the traditional collectors for small copy reserve sizes. When the nursery copy reserve is set at 20% and the mature copy reserve is less than 60% GenCC clearly outperforms the generational copying collector. When the mature copy reserve is larger, the performance is comparable to that of the generational copying algorithm, while still outperforming GenMS.

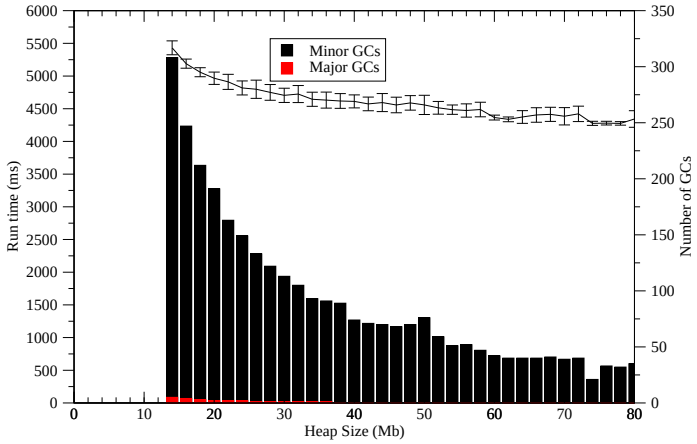
Figure 5.11(b) shows the performance improvement when the nursery copy reserve is larger. As would be expected, the improvement is less. However, for small mature copy reserves (those smaller than 30%) GenCC still outperforms both traditional collectors. However, the performance difference between the new collector and the generational copying collector is minimal. Only at mature copy reserve sizes larger than 90% does the performance become comparable to the GenMS collector.

### Variable Heap Size

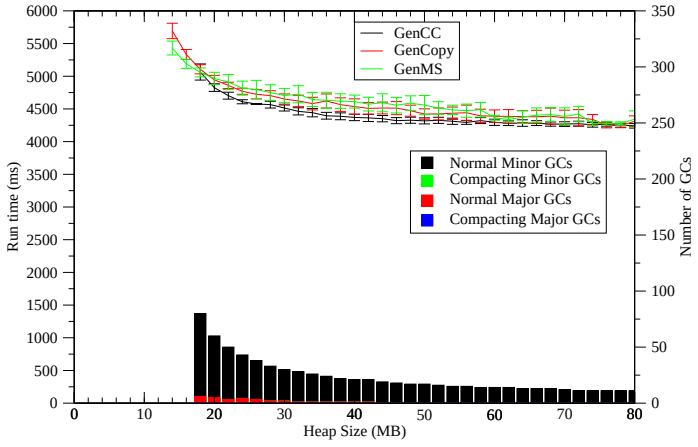
In Figure 5.10(c), it can be seen that GenCC outperforms the two traditional collectors for almost all heap sizes. At larger heap sizes (those greater than 60Mb) the elapsed times of the three collectors converge. A this point the heap has become large enough that the time spent in garbage collection is no longer significant.




(a) GenCopy

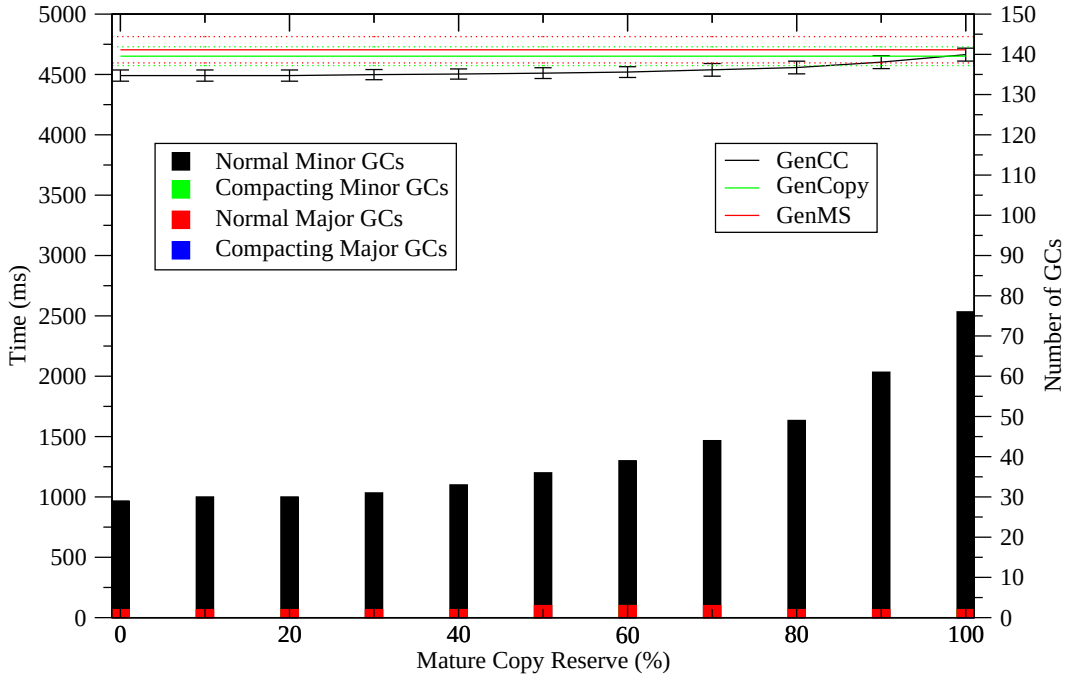



(b) GenMS

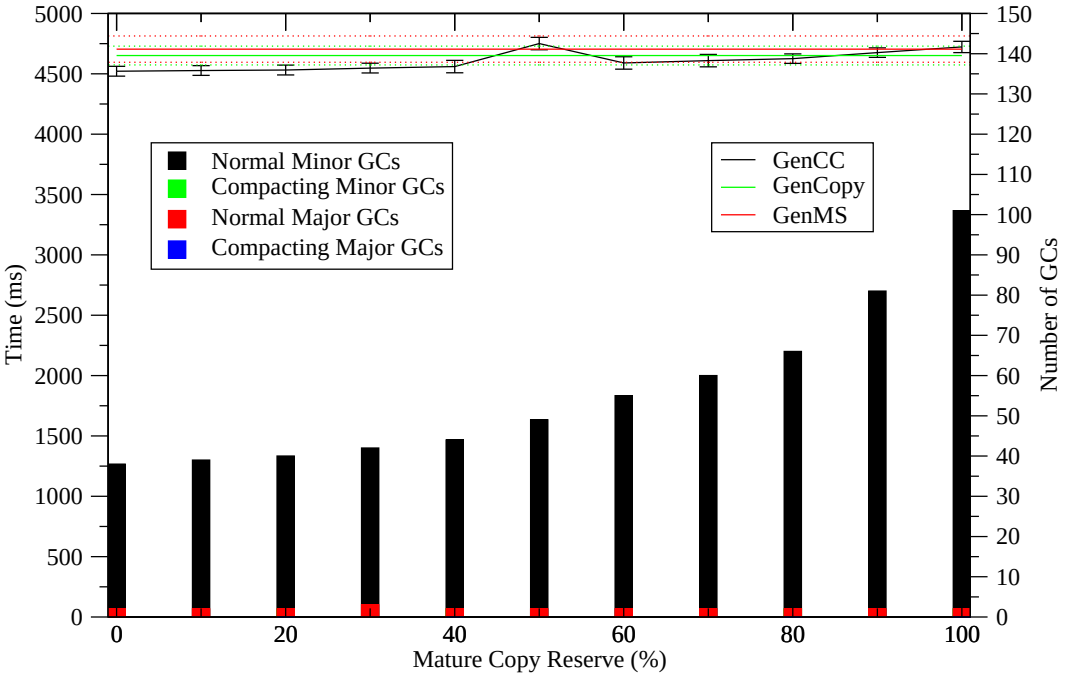



(c) GenCC

Figure 5.10. _228_jack




(a) 20% nursery copy reserve




(b) 60% nursery copy reserve

Figure 5.11. _213_javac with 36Mb heap

As seen before, there are no compacting collections required when the nursery and mature copy reserves are both set to 20%.

## 5.5 Summary of Results

In general, the results show that the generational copying/compacting collector with well-chosen parameters performs well when compared to the traditional collectors. In the majority of benchmarks, GenCC has elapsed time similar to, or better than, both GenCopy and GenMS.

It can be seen from the variable copy reserve graphs that when the copy reserve size is too small the compacting collector is triggered too frequently. This leads to a performance loss, since the overhead of the compaction is far higher than a normal collection. Similarly, when the copy reserve size is too large, the benefit gained by freeing up space is insufficient to account for the additional implementation overheads of the GenCC algorithm. Therefore, it can be seen that an optimal copy reserve size is as small as possible, while still preventing frequent compacting collections. The experiments have shown that this size is 20% for these benchmarks.

It can also be observed that the GenCC collector shows the most improvement over the GenCopy and GenMS collectors when there is a high rate of allocation and when the object lifetimes obey the weak generational hypothesis. In this case, the additional time taken for the larger nursery space to fill allows longer for objects to die. As a result, fewer objects are promoted to the mature space, and there is more space available to the nursery on subsequent allocation cycles.

# 6 CONCLUSIONS

*The run time performance of a generational copying garbage collector can be improved by reducing the size of the copy reserve.*

## 6.1 Summary

This thesis presented the design, implementation and measurement of an a new generational copying garbage collection algorithm, in which a traditional Appel-style collector was improved by reducing the necessary copy reserve. Correctness was maintained through the use of a compacting collector in the rare occasions that the smaller copy reserve was insufficient. The thesis outlined the foundations upon which the work was based, detailed the reasoning behind the design, outlined some interesting features of the implementation and presented experimental results.

Through the implementation and measurement of the garbage collector, the thesis has been proven to be correct. By reducing the copy reserve overhead of a generational copying collector the overall performance can be improved. There follow some conclusions drawn through the evaluation of the experimental results.

## 6.2 Copy Reserve Size

While it is important to minimize the space used for the copy reserve, it is not always beneficial to eliminate it completely. The experimental results shown in Chapter 5 indicate that if the copy reserve is too small performance suffers. This is understandable, since the compacting collection takes significantly longer than a normal collection, and so too many compactions will hurt performance.

Similarly, the performance of the collector suffers if the copy reserve is too large. While in theory a collector with nursery and mature copy reserves set to 100% should perform exactly as a traditional Appel-style collector the results show that this is not the case. The implementation adds overheads such as increased complexity in the space management in order to allow the block copy mechanism and additional work performed during the scanning stage. These overheads are necessary for proper space management in the presence of remapping; the traditional collectors are able to avoid them by mapping a virtual address range once and then never releasing it. When the copy reserve is too large, these overheads outweigh any performance improvement.

From the results in the previous chapter, it can be seen that in all benchmarks examined, a 20% nursery and mature copy reserve will accommodate all live objects without compaction. This copy reserve size generally allows enough space to be repartitioned over the traditional generational collector to show a performance improvement, while not suffering the overhead of compaction.

## 6.3 Mutator Effects

It can be seen from the results presented that the characteristics of the benchmark have a large effect on the performance of the collector. This is demonstrated even for the relatively simple benchmarks shown.

As described in _201_compress benchmark (Section 5.4.1), the new collector does not perform as well on benchmarks with a high survival rate as it does on those with a low survival rate. Even when the survival rate does not trigger compaction it leaves less space for the collector to use. In these cases the savings gained by reducing the copy reserve are smaller, and the implementation overheads dominate.

Benchmarks such as _213_javac, however, allow the advantages of the new algorithm to be exploited. In this case, the object lifespans closely conform to the generational hypothesis. Few nursery objects are promoted, meaning that the mature space does not

grow quickly. This increases the space available for the collector, and maximizes the improvement seen by reducing the copy reserve.

## 6.4 Future Work

There remain several avenues of research that could be performed after this work.

An adaptive system could be implemented. This would measure the survival rate of a benchmark over the course of its execution and tune the copy reserve appropriately. The design of the algorithm allowed for this possibility by separating the mature and nursery copy reserves. This would allow flexibility in tailoring each copy reserve in response to the benchmark. It would be interesting to see how great an improvement would be measured in performance over the current system with fixed 20% copy reserves.

It would also be of interest to run larger benchmarks using this system. The stability of Jikes RVM at the time of writing meant that it was not possible to run large benchmarks for enough iterations to gain a statistically significant result. However, it may be that improvements in Jikes RVM make this possible in the future.

# LIST OF REFERENCES

# LIST OF REFERENCES


[1] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, May 1996.

[2] Doug Lea. A memory allocator. http://gee.cs.oswego.edu/dl/html/malloc.html, 1997.

[3] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, November 1970.

[4] Xianlong Huang, Stephen M. Blackburn, Kathryn S. McKinley, J. Eliot B. Moss, Z. Wang, and Perry Cheng. The garbage collection advantage: Improving program locality. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 39, October 2004.

[5] H. B. M. Jonkers. A fast garbage compaction algorithm. *Information Processing Letters*, 9(1):25–30, July 1979.

[6] David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the ACM Symposium on Practical Software Development Environments*, pages 157–167, 1984.

[7] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. Myths and reality: The performance impact of garbage collection. In *Sigmetrics - Performance 2004, Joint International Conference on Measurement and Modeling of Computer Systems*, New York, NY, June 2004.

[8] Sunil Soman, Chandra Krintz, and David Bacon. Dynamic selection of application-specific garbage collectors. In *Proceedings of the ACM International Symposium on Memory Management*. ACM, 2004.

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

[10] Stephen M. Blackburn and Kathryn S. McKinley. Ulterior reference counting: Fast garbage collection without a long wait. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 38, November 2003.

[11] Stephen M. Blackburn, Richard E. Jones, Kathryn S. McKinley, and J. Eliot B. Moss. Beltway: Getting around garbage collection gridlock. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, volume 37, pages 153–164, May 2002.

[12] Stephen Blackburn and Kathryn S. McKinley. In or out?: Putting write barriers in their place. In *Proceedings of the ACM International Symposium on Memory Management*, volume 38, pages 281–290. ACM, February 2003.

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

[14] Matthew Arnold, Stephen J. Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 35, pages 47–65, October 2000.

[15] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. Oil and water? high performance garbage collection in java with MMTk. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 137–146, Washington, DC, USA, 2004. IEEE Computer Society.

[16] Michael G. Burke, Jong-Deok Choi, Stephen J. Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio J. Serrano, Vugranam C. Sreedhar, Harini Srinivasan, and John Whaley. The Jalapeño dynamic optimizing compiler for Java. In *Proceedings of the ACM Java Grande Conference*, pages 129–141, 1999.

[17] Jos Manuel Velasco, Antonio Ortiz, Katzalin Olcoz, and Francisco Tirado. Adaptive tuning of reserved space in an Appel collector. In Martin Odersky, editor, *Proceedings of the European Conference on Object-Oriented Programming*, volume 3086 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004.

[18] Bowen Alpern, C. R. Attanasio, John J. Barton, Anthony Cocchi, Susan Flynn Hummel, Derek Lieber, Ton Ngo, Mark Mergen, Janice C. Shepherd, and Stephen Smith. Implementing Jalapeño in Java. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 34, pages 314–324, October 1999.

[19] Henry G. Baker. The Treadmill: Real-time garbage collection without motion sickness. *ACM SIGPLAN Notices*, 27(3):66–70, March 1992.

[20] SPEC. SPECjvm98 benchmarks, 1998. http://www.spec.org/osg/jvm98.