UNIVERSITÉ DU QUÉBEC À MONTRÉAL

RAMASSE-MIETTES GÉNÉRATIONNEL ET INCÉMENTAL GÉRANT LES CYCLES ET LES GROS OBJETS EN UTILISANT DES FRAMES DÉLIMITÉS

MÉMOIRE

PRÉSENTÉ
COMME EXIGENCE PARTIELLE DE LA
MAÎTRISE EN INFORMATIQUE SYSTÈME

PAR
SÉBASTIEN ADAM

AVRIL 2008

UNIVERSITÉ DU QUÉBEC À MONTRÉAL

BOUNDED FRAME, CYCLE, AND LARGE OBJECT HANDLING IN GENERATIONAL OLDER-FIRST GARBAGE COLLECTION

THESIS

SUBMITTED
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS OF THE DEGREE OF
MASTER IN COMPUTER SCIENCE

BY
SÉBASTIEN ADAM

APRIL 2008

Acknowledgments

This work would not have been possible without the support and encouragement of many people. First, I would like to thank my supervisor, Professor Étienne Gagnon, for his guidance throughout the course of this thesis research. Without his common-sense, knowledge, and perceptiveness I would never have finished. Thanks, Étienne, for all your help and support! I would like to thank all the student members of the McGill Sable Research Group for their help discussing this work. I would like to thank Gregory Prokopski for building the scripts, for collecting empirical results and transforming the output into various formats. I would like to thank Éric Thé and my parents-in-laws for their support while writing my thesis. Finally, how could I thank my wife Anna Loevenich enough for the incredible amount of patience she had with me through the last six months, and for so much more she provided me? Thanks, thanks, thanks, and more thanks for all of you.

Contents

List of Figures

List of Tables

List of Acronyms

GC Garbage Collection

J2SE Java 2 Platform Standard Edition

JNI Java Native Interface.

JVM Java Virtual Machine.

LOS Large Object Space

SableVM Sable Virtual Machine

SOS Small Object Space

SPEC Standard Performance Evaluation Corporation.

TLB Translation Lookaside Buffer

VM Virtual Machine.

Résumé

Ces dernières années, des recherches ont été menées sur plusieurs techniques reliées à la collection des déchets. Plusieurs découvertes centrales pour le ramassage de miettes par copie ont été réalisées. Cependant, des améliorations sont encore possibles.

Dans ce mémoire, nous introduisons des nouvelles techniques et de nouveaux algorithmes pour améliorer le ramassage de miettes. En particulier, nous introduisons une technique utilisant des cadres délimités pour marquer et retracer les pointeurs racines. Cette technique permet un calcul efficace de l'ensemble des racines. Elle réutilise des concepts de deux techniques existantes, *card marking* et *remembered sets*, et utilise une configuration bidirectionelle des objets pour amcliorer ces concepts en stabilisant le surplus de mémoire utilisée et en réduisant la charge de travail lors du parcours des pointeurs. Nous présentons aussi un algorithme pour marquer récursivement les objets rejoignables sans utiliser de pile (eliminant le gaspillage de mémoire habituel). Nous adaptons cet algorithme pour implémenter un ramasse-miettes copiant en profondeur et améliorer la localité du *heap*. Nous améliorons l'algorithme de collection des miettes *older-first* et sa version générationnelle en ajoutant une phase de marquage garantissant la collection de toutes les miettes, incluant les structures cycliques réparties sur plusieurs fenêtres. Finalement, nous introduisons une technique pour gérer les gros objets.

Pour tester nos idées, nous avons conçu et implémenté, dans la machine virtuelle libre Java SableVM, un cadre de développement portable et extensible pour la collection des miettes. Dans ce cadre, nous avons implémenté des algorithmes de collection *semi-space*, *older-first* et *generational*. Nos expérimentations montrent que la technique du cadre délimité procure des performances compétitives pour plusieurs *benchmarks*. Elles montrent aussi que, pour la plupart des *benchmarks*, notre algorithme de parcours en profondeur améliore la localité et augmente ainsi la performance. Nos mesures de la

performance générale montrent que, utilisant nos techniques, un ramasse-miettes peut délivrer une performance compétitive et surpasser celle des ramasses-miettes existants pour plusieurs *benchmarks*.

Mots clés: Ramasse-Miettes, Machine Virtuelle, Java, SableVM

Abstract

Over the years, research has been done on several techniques related to garbage collection. Many key insights for copying-based generational garbage collection techniques have been revealed. Yet, there is still room for improvement.

In this thesis, we introduce various new techniques and algorithms to improve garbage collection. In particular, we introduce the bounded frame marking technique for tracking pointers. This technique allows for efficient computation of the root set. It reuses concepts from two existing techniques, card marking and remembered sets, and uses a bidirectional object layout to improve them by regulating space overhead and reducing the pointer scanning workload. We also present an algorithm to recursively mark reachable objects without using a stack (eliminating the usual space overhead). We adapt this algorithm to implement a depth-first copying collector and increase heap locality. We improve the older-first garbage collection algorithm and its generational variant by adding a mark phase that guarantees the collection of all garbage, including cyclic structures spanning many windows. Finally, we introduce a technique to deal with large objects.

In order to test our ideas, we have designed and implemented a portable and extensible garbage collection framework within the SableVM open source Java virtual machine. In it, we have implemented semi-space, older-first, and generational copying garbage collection algorithms. Our experiments show that the bounded frame technique yields competitive performances on many benchmarks. They also show that, for most benchmarks, our depth-first traversal algorithm improves locality and thus increases performance. Our overall performance measurements show that, using our techniques, a garbage collector can deliver competitive performance and surpass existing collectors on various benchmarks.

Key words: Garbage Collection, Virtual Machine, Java, SableVM

Chapter I

Introduction and Contributions

1.1 Introduction

1.1.1 Automatic Memory Management

Modern virtual machines use automatic memory management and free the programmers from explicit memory allocation and deallocation. This software engineering principle of abstraction is the method software engineers use to manage the complexity of systems. Hiding details from the programmers improves programming productivity. As shown in Figure 1.1 (see [Gagnon 03b]), the memory manager module is part of current state-of-the-art virtual machines.

Modularity is another software engineering principle that engineers use to build extensible, reusable, and portable system components. For example, the Boehm-Demers-Weiser conservative deallocators [Boehm 01] are modules allowing C++ programmers to allocate memory without explicitly deallocating memory that is no longer useful. Programmers are increasingly choosing object-oriented languages to take advantage of these software engineering benefits. Commonly proposed estimates are that up to 40 percent of the development time is spend implementing memory management procedures and debugging errors related to explicit storage [Rovner 85].

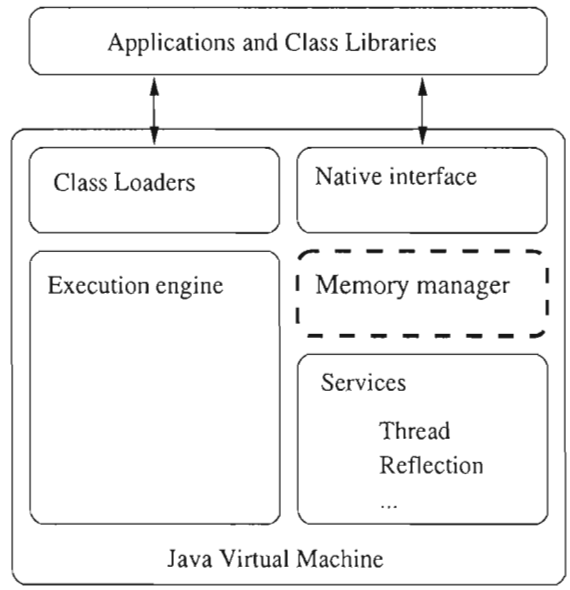

Figure 1.1 Java Virtual Machine Overview

1.1.2 Garbage Collection Algorithm

Memory allocation is often quite simple. The virtual machine allocates a large contiguous block of memory called the heap. Every time a program request memory, the virtual machine reduces the large block accordingly to the requested memory size. When all the usable memory is exhausted, the deallocation mechanism is triggered.

Memory deallocation is far more complex. Copying garbage collection is a form of automatic memory deallocation. A *garbage collector* is a piece of software that recycles unreachable memory or garbage. Current state-of-the-art Java virtual machines implement advanced garbage collection techniques to deliver high-performance execution of Java bytecode [Gagnon 03b; IBM 03]. Popular techniques implement both non-copying (reference counting and mark-and-sweep) and copying (mark-and-compact, semi-space, generational, and incremental) collectors.

1.1.3 Garbage Collector Implementation

A garbage collector should provide *completeness*, the guarantee to reclaim all garbage eventually. On that account, standard copying collectors require a copy reserve equal to twice the size of the maximum live data for a program. In order to provide responsiveness, older-first and generational copying algorithms collect only a region of the heap at a time. But sometimes they do full collections which require a copy reserve equal to the usable memory. Some algorithms, as the mark-copy collection algorithm [Sachindran 03], use memory space more efficiently. It marks live data and always copies a number of survivors that fit into a fixed-sized region. The mark phase, however, requires additional space for a mark stack [Sachindran 03].

Collectors that partially collect the heap, track pointers that refer from one region to another, and include these pointers in the root set. These collectors mainly use card marking or remembered set methods to maintain their root set. Card marking increases work scanning at collection time. On the other hand, remembered sets can have a significant space overhead (up to 25% of the maximum live size for the mark-copy algorithm which further does not track all pointers [Sachindran 03]).

1.1.4 Garbage Collection Costs

Current garbage collectors still have significant performance overhead [Blackburn 02a]. The costs of copying garbage collection include (1) the cost of copying objects when they survive a collection, (2) the cost of pointer-tracking, and (3) the cost of the interaction between the cache and memory behaviors of both the program and the garbage collector [Stefanović 99a].

Research [Hertz 05a] has shown that when physical memory is scarce, paging causes garbage collection to run an order of magnitude slower than explicit memory management. Garbage collection can degrade overall performance by nearly 70%. Overall performance is highly dependent upon the behavior of the application as well as on

the available resources.

1.1.5 Larger Space and Poor Locality

No single garbage collector enables the best performance for all programs and all heap sizes [Soman 04]. Garbage collection can comprise 35% of execution time when the heap space is tight on some systems [Blackburn 02a]. The best total execution time is not always achieved simply by using a larger heap, however. Locality may degrade with large heaps, increasing paging and cache miss rates.

Programs often access objects of a similar age closely together [Stefanović 99a; Blackburn 02a]. Because copying garbage collectors move objects, they have the opportunity to improve locality with consequent benefits for cache and translation lookaside buffer (TLB) behavior. At collection time, for an object-oriented language such as Java, garbage collectors may execute a depth-first or a breadth-first traversal of the object tree to better tune memory layout for program traversal. Static orders are problematic, with large differences of up to 25% in total time for some benchmarks [Huang 04], when traversal patterns do not match the collectors single order. Achieving high performance always remains a challenge.

1.1.6 Customisable Memory Manager

Several techniques have been developed to make garbage collection feasible in many situations, including real time applications. Optimal performance cannot always be achieved by a uniform general purpose solution [Attardi 98].

A current general trend in software development is towards customisable systems to give the developer greater flexibility and control over the functionality and performance of their application. MMTk [Blackburn 04b] is an extensible framework for building garbage collectors. MMTk uses design patterns and compiler cooperation to combine modularity and efficiency. Beltway [Blackburn 02a] is another framework that significantly generalizes existing copying collectors. This generality enables developers

to exploit a larger design space and develop better collectors.

1.2 Research Motivation and Objectives

We should note that various methods exist for memory allocation. Our research focuses solely on copying garbage collection algorithms.

1.2.1 Copying Garbage Collectors

The main motivation behind this thesis was to study and understand some copying garbage collectors. In order to address these objectives, we have implemented and tested some popular algorithms for semi-space, older-first, and generational copying garbage collection on SableVM [Gagnon 03b], an extremely portable, efficient, and specification-compliant Java virtual machine.

We have exploited the bidirectional-object layout furnished in SableVM to implement new techniques for pointer tracking and object traversals. Our pointer-tracking technique combines both card marking and remembered set methods to regulate space overhead and reduce work scanning. We also have designed a new algorithm to mark the object tree recursively without space overhead. We use this algorithm to implement a depth-first copying collector. In this thesis we report our overall performance measurements and show that, using our techniques, a garbage collector can deliver competitive collection performance and even surpass that of a traditional collector on some benchmarks.

1.2.2 Research Framework

Many academic research projects have limited resources. Sometimes, the human resources dedicated to a project are limited to a very small team of researchers. One of the objectives of this research is the development of an openly available memory manager framework suitable for performing research experiments with minimal effort. In order to achieve this goal, this framework must be easily extensible, thus allowing

experiments with new algorithms for memory allocation and deallocation. This research framework must also be easily portable to new platforms with minimal effort, in order to perform experiments on a variety of systems. Finally, the memory manager must also deliver an acceptable performance, so that experiments can be done running real-world applications, rather than toy benchmarks.

We have designed, and implemented on SableVM [Gagnon 03b], a portable and extensible framework for building and testing garbage collectors. SableVM provides a logical partitioning of runtime memory that simplifies memory management. This memory partitioning allows SableVM to use a configurable collector to manage the Java heap, and to use partition-specific memory managers for the rest. The collectors share all common mechanisms, policies, and functionalities and use the exact same implementation, allowing us to obtain more accurate experimental results.

1.2.3 Specific Research Objectives

Research has identified many key insights for copying garbage collection. Current state-of-the-art collectors implement generational [Ungar 84; Appel 89; Lin 92; Baker 93; Chilimbi 98; Ali 98; Blackburn 02a] and incremental [Deutsch 76; Baker 78; Baker 92; Hansen 00; Hansen 02; Stefanovic 02] algorithms and exploit data locality [Courts 88; Chilimbi 98; Boehm 00; Blackburn 04a; Huang 04], object lifetime [Lieberman 81; Lieberman 83; Hanson 90; Barrett 93; Inoue 03; Blackburn 07], and object segregation [Caudill 86; Ungar 92; Hicks 98; Colnet 98] to provide better performances. Current software implementations also use modular design to provide extensibility and flexibility to the developers [Blackburn 02a; Blackburn 03a; Blackburn 04b], even in the presence of critical performance software. Given these key insights the specific objectives of this research are to:

- research new garbage collection techniques to address these key insights,
- evaluate the relative performances achievable by copying collectors implementing

our innovative techniques,

- measure the performances of the proposed techniques,
- design and implement a portable and easily modifiable memory manager.

A less formal objective is to keep the framework as simple as possible, leaving the development of more advanced techniques to future interested users of this framework.

1.3 Contributions

In this section, we list the contributions of this thesis.

One contribution of this thesis is the memory manager framework itself. During this research, we have integrated the memory manager framework on SableVM [Gagnon 03b], an open-source virtual machine for Java. We think that the clearness and sharpness of its internal design makes it a valuable framework for conducting research projects on memory manager techniques.

The 2 kinds of object layouts implemented on SableVM enable designers to explore a larger design space and develop better collectors. The memory manager framework has embedded debbuging and testing features suitable for the evaluation of new ideas.

Thus, the contributions of this thesis are:

- The introduction of an innovative technique to track pointers efficiently in presence of a partitioned heap, without creating too much space overhead. Experiments show that using our technique to reduce both the tracing costs and the space overhead of the card marking and the remembered set techniques respectively, we can engender competitive performances.
- The introduction of an algorithm, which exploits the bidirectional object layout that groups together all reference fields, to traverse the objects in the heap using

a depth-first order without space overhead. We use this algorithm to implement the two following methods.

- The implementation of a method to copy objects in a depth-first order without space overhead. Our experiments show that, exploiting the static class-oblivious copying orders (e.g., breadth-first and depth-first), we can tune memory layout to program traversal and improve performance, instead of always using the same static order.

- The implementation of a method to mark objects without space overhead. This method guarantees that collectors will collect all garbage.

- The introduction of a large object policy that regroups large objects in memory and makes assumptions about their lifetime. Our experiments show that large object segregation yields performance improvements.

- The implementation of many garbage collectors which offer the developers the ability to customise the memory manager framework on a per-application basis. The framework allows the developers to exploit application specific behavior, and our experiments show that it may also improve performance. The collectors share all common mechanisms and provide reusable components for future implementations, which improve the programming productivity and reduce the development effort.

- The development and the planned public release of the memory manager research framework.

1.4 Thesis Organization

The remainder of this thesis is structured as follows. In Chapter 2, we describe some popular garbage collection algorithms. We also present points of comparison often used to compare these garbage collection algorithms. In Chapter 3, we describe the older-first algorithm in more details. In Chapter 4, we introduce a new method that

combines the remembered set and the card marking mechanism to improve efficiency and precision while tracking pointers. In Chapter 5, we discuss the garbage structures and introduce a depth-first traversal algorithm for marking objects without space overhead. In Chapter 6, we describe a technique for segregating long-lived large objects in a space which is less often collected. In Chapter 7, we describe a generational older-first algorithm which provides incremental collection for all of its generations. In Chapter 8, we introduce a depth-first semi-space copying collector that uses the objects traversal previously introduced to improve program locality. In Chapter 9, we discuss the implementation of our memory manager framework. In Chapter 10, we describe our experimentation setting, and present our overall performance measurements. In Chapter 11, we present some related works. Finally, in Chapter 12, we discuss possible future work and present our conclusions.

Chapter II

GARBAGE COLLECTION

A garbage collection system is a form of automatic memory management. It eliminates a significant source of software defects by freeing programmers from explicit memory allocations and deallocations. By removing the burden of explicitly reclaiming memory from the programmer, well-known classes of errors, including dangling references and certain kinds of memory leaks, can be avoided [Wilson 92; Wadler 87]. Collectors also improve memory modularity and programmer productivity [Röjemo 95]. Because of these advantages, garbage collection has been incorporated as a feature in a number of mainstream programming languages.

The implementation of an efficient garbage collector is complex. Many algorithms have been proposed to provide completeness, i.e. the guarantee to reclaim garbage eventually. Basic algorithms include *mark-and-sweep* collection [Smith 98; McCarthy 60], *semi-space copying* collection [Smith 98; Fenichel 69], and *generational copying* collection [Cheadle 04]. Each algorithm has a number of advantages over the others. The best performance for a program (also called a *mutator*) and a heap size is achieved by selecting and properly tuning the appropriate garbage collection algorithm. In the best case, each program should have its own garbage collector tuned specifically to meet the program's needs and the technical restrictions of the system.

In this section, we introduce the memory manager, the allocator, and the collector. Then we describe specific points of comparison often used in the literature dealing with

garbage collection algorithms. Finally, we highlight the most popular garbage collection algorithms. Some related constructs such as write barrier, remembered set, and card marking are also presented. We further compare algorithms and describe some of their implementations.

2.1 Memory Manager

The memory manager can be divided in two parts: the allocator which allocates memory and the collector which recycles memory that the mutator is unable to reach. This unreachable memory is also called garbage. The mechanism that recycles memory is called garbage collection (GC).

2.1.1 Memory Allocator

Dynamic memory allocation is an important part of many programs. Unnecessary allocation can decrease program locality and thus increase fragmentation and execution time. A previous study [Grunwald 93] has shown how the design of a memory allocator can significantly affect the reference locality of various applications. Their measurements show that poor locality in sequential-fit algorithms reduces program performance, both by increasing paging and cache miss rates.

By dynamic memory allocation, we mean that the memory for a program is taken from a large area of previously reserved memory called the heap. The size of the allocation can be determined at run-time. The lifetime of the allocation does not depend on the current procedure or stack frame. The allocated memory region is accessed indirectly, usually via a reference. The precise algorithm used to organize the memory area and to allocate and deallocate chunks is hidden behind an abstract interface and may use any of the methods described above. The allocator is the component of software that dynamically allocates memory.

2.1.1.1 Contiguous Allocator

The basic algorithm is usually quite simple to implement. It uses a pointer to reference the next free space in the heap. When the mutator requests some heap space, the allocator checks if the requested space is available. If this is the case, the pointer is simply returned and updated to reference the following unused space. Otherwise, the collector is called to free the heap from the garbage. This is called a contiguous allocator. In an object-oriented environment, contiguous allocators append new objects to the end of a contiguous space by incrementing a bump pointer with the size of the new object.

2.1.1.2 Free-List Allocator

Some systems implement a different scheme of dynamic memory allocation. Sometimes, a free list is used to connect unused blocks of memory together in a linked list, using the first word of each block as a pointer to the next. This technique is most suitable for allocating from a memory pool, where all objects have the same size. In such a context, free lists make the allocation and deallocation operations very simple. To free a block, we just add it to the free list. To allocate a block, we simply remove it from the end of the free list and use it.

In a language like Java, memory allocation is done using object size. The blocks have variable sizes. This means that the allocator may have to search for a block that suits its request. This operation can be very time consuming. Also, free lists have the disadvantage of poor reference locality resulting in poor cache utilization. Handling the allocation requests for large blocks is even more time consuming. It has been shown that even algorithms attempting to be space-efficient by coalescing adjacent free blocks show poor reference locality [Grunwald 93]. All these flaws affect the benefits of space efficiency and consequently execution time.

Another approach uses a free-list allocator that organizes memory into k size-

segregated free-lists. Each free list is unique to a size class and is composed of blocks of contiguous memory. An object is allocated into a free block of the smallest size class that accommodates the object. The space-efficient free-list reduces total collector load. However, the mutator's locality in the context of contiguous allocation provides fewer misses at all levels of the cache hierarchy [Blackburn 04a]. As heap size increases, the spatial locality of objects allocated close together in time is key to better performance.

In this study, we opt for the simplicity and efficiency of the contiguous allocator strategy. The allocator is just a necessity for our work. However, we concentrate our efforts on the deallocation mechanism.

2.1.2 Garbage Collector

Dynamic memory deallocation is as important as allocation. If reclamation is not performed, or if some objects are accidently not reclaimed (memory leak), programs can fail when they reach the memory size limit. The collector is the component that deallocates memory. It determines which objects in the heap are either dead or unreachable. It reclaims the storage used by these objects and feeds back the unused space to the allocator. An implementation of efficient garbage collection is complex. There are lots of algorithms aimed at collecting the heap. Each of them present advantages and disavantages and none provide the best performances in all cases.

There exists two major classes of garbage collectors: *tracing* and *non-tracing* collectors. Non-tracing collectors (mainly using reference counting) cannot reclaim cyclic data structures. They are a poor fit for concurrent programming models and have a high reference count maintenance overhead.

All tracing collectors must trace a subset of the heap looking for reachable (or *live*) objects. They start from a precomputed set of references (called the *root set*) for finding reachable objects. They identify live objects by computing a transitive closure from the roots. All objects unreachable directly or indirectly from the roots are considered as garbage. In a Java virtual machine, the root set includes stack variables, registers,

and class variables. Figure 2.1 illustrates the concepts of roots, reachable objects, and garbage.

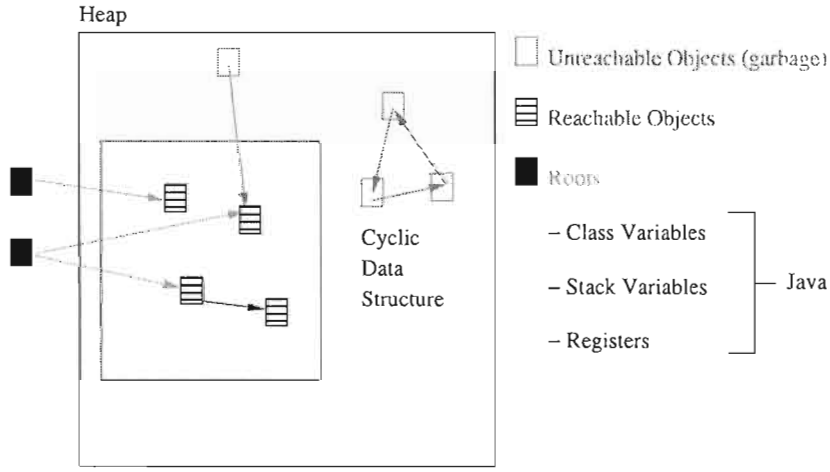

Figure 2.1 Roots, Reachable Objects, and Garbage

The rest of this section describes some garbage collection algorithms. We first present some points of comparison for garbage collection.

2.2 Points of Comparison

Users have distinct requirements of garbage collection. When choosing a particular garbage collection system, we must first identify our needs and then select and appropriately tune the collector that best fulfills our requirements. Table 2.1 non-exhaustively introduces some points of comparison for garbage collection usually found in the literature.

Pause Time

Pause time or latency refers to the time delay between the moment a collection is initiated and the moment it is completed. A pause is the time when an application appears unresponsive because garbage collection is going on. When a pause happens,

Point	Meaning
Pause Time	time used to collect objects
Throughput	fraction of time spent in the mutator
Promptness	time between when an object becomes dead and when the memory becomes available
Completeness	garbage must be completely collected
Space Overhead	amount of space used to realize a garbage collection
Unused Space	amount of allocatable space unused when garbage collection happens
Fragmentation	process of memory being divided into smaller fragments of memory
Temporal Locality	objects referenced at a nearby time are close to each other in memory
Spatial Locality	objects close to each other in memory are referenced at a nearby time

Table 2.1 Points of Comparison of Garbage Collection

we need to trace, copy, and sometime mark objects. Pause time thus includes time for copying, tracing, and marking.

Throughput

Throughput indicates the ratio between the mutation and application time, the latter being the summation of both pauses and mutation time. Depending on the context, someone may consider the right metric to be pause time. For instance, in a real-time or interactive environment even short pauses may be intolerable. For a web server however, the right metric is usually throughput. Pauses during garbage collection may be obscured by network latencies.

Completeness and Fragmentation

When choosing or implementing garbage collectors, certain properties, such as completeness to prevent memory leaks or prematured out-of-memory errors, should be attained. Others, such as fragmentation and promptness to maintain the overall

performance, should carefully be considered. Fragmentation happens when different sized memory blocks are allocated and freed repeatedly. The allocator may not find an empty block with exactly the wanted size, so it uses a larger block. The unused part may be used for other even smaller allocations. If this happens too often, many of these small unused parts (or fragments) appear.

Memory fragmentation increases the amount of unused space, resulting in more frequent collections. It also degrades the locality and the efficiency of object allocation, introduces premature garbage collections, and may even cause a failure to satisfy an allocation request [Ossia 04; Barabash 03]. Thus, memory fragmentation has a big impact on overall performance. A lack of promptness may cause fragmentation. So, allocators and collectors must cautiously manage both of these criteria to avoid performance degradation.

Space Overhead and Unused Space

On systems with limited physical memory, space overhead and unused space are crucial concerns to observe. In such a situation, a collector with a significant space overhead may cause an out of memory error, while another collector would still be able to realize its collections. Even a small amount of unused space can negatively affect throughput and consequently decrease overall performance in the presence of limited memory.

Locality

All collectors are affected by locality. Many studies have been made to develop collectors which use locality as its primary criteria [Shuf 02; Courts 88; Huang 04; Guyer 04; Chilimbi 98; Hirzel 03]. Most of these locality-based collectors aim at reducing the number of cache misses in order to improve overall performance. This phenomenon has long been recognized as an important characteristic of program behaviour. Current memory hierarchies exploit reference locality to reduce load latency and thereby improve proces-

sor performance [ching Ju 01]. These hierarchies exploit spatial locality by fetching a region of memory rather than just the accessed data. Caches exploit temporal locality by retaining recently accessed cache blocks. Most cache management schemes exploit locality to increase the fraction of memory accesses satisfied by the cache (i.e., cache hit ratio) and thus enhance performance.

When choosing and tuning a particular garbage collection system, one has to weigh all these considerations. Users must understand their environment and the application's behavior and consciously manage these concerns to generate better performance. The rest of this chapter introduces some algorithms which collect garbage and provides important points of comparison for each of them.

2.3 Reference Counting

Reference counting collectors count incoming pointers for each object, and reclaim this object when its count reaches zero. The object's reference count is incremented when the object is referenced, and decremented when such a reference is destroyed [Collins 60]. Thus, reference counting updates reference counts on each pointer stored.

Designers typically use a free-list to allocate memory on a per-object basis, and by doing so they allow the collector to reclaim garbage immediately. Reference counting performs recursive deletion when an object's reference count drops to zero. Intuitively, one can think of reference counting as operating upon dead objects; it traverses the object graph forward to find garbage, starting with the set of objects whose reference counts were reduced to zero.

Table 2.2 summarizes the impacts of reference counting for comparison purposes. Let us describe these impacts more precisely.

	RC
Pause Time	v
Throughput	v
Promptness	e
Completeness	-
Space Overhead	g
Unused Space	b
Fragmentation	b
Locality	b

Meaning
(b)ad, (g)ood, (v)ery good, (e)xcellent
(-) not provided, (+) provided

Table 2.2 Points of Comparison for Reference Counting

Fragmentation, Unused Space, and Locality

A memory manager which does not move objects may suffer from memory fragmentation. Designers often implement a free-list allocator with reference counting. Even though this technique is suitable for environments which create objects of equal size, it may cause significant fragmentation in an environment using a language such as Java which creates objects of variable sizes. This fragmentation implies more unused memory. Further degradation of the locality of objects usually leads to a loss of performance. However, reference counting employing a proper allocation strategy usually exploits memory space efficiently.

Throughput

Reference counting does not trace objects and often does not move them too much thanks to a generally used free-list allocator. Thus, the deallocation is usually very fast and yields a good throughput.

Pause Time, Space Overhead and Promptness

Reference counting produces very short pause times, which makes it suitable for real-time applications [Blackburn 03b]. In addition, it allows the collector to reclaim garbage promptly. However, the cost of updating reference counts every time a root

pointer is updated is often much too high for high-performance mutators. Consequently, some form of deferred reference counting has been proposed by authors which trade-off promptness for space overhead [Deutsch 76; DeTreville 90; Bacon 01a], thus improving overall performance at the expense of longer pauses.

Reference counting performs recursive deletions. The amount of space consumed by a recursive traversal increases space overhead as it is done by reference counts. The space for the traversal stack can be eliminated (see Chapter 5). Weizenbaum proposes a non-recursive method for freeing [Weizenbaum 63]; it reduces the space overhead at the expense of promptness.

Completeness

Reference counting inherently fails to collect cyclic garbage. If two objects refer to each other, neither will be collected as their mutual references never let their reference counts equal zero. However, some solutions such as the backup tracing collector [Weizenbaum 69] and the trial deletion algorithm [Bacon 01b; Martinez 90; Lins 92] have been proposed.

2.4 Mark and Sweep

Mark refers to the marking process during which reachable objects are marked. Unmarked objects are freed and the resulting memory is made available to the allocator during the *sweep*. A mark-and-sweep collector does not move objects. It coalesces garbage into free blocks of memory and feeds them back to a free-list allocator.

While sweeping the heap, the collector visits all the objects including all garbage. For that reason, copying collectors have been usually preferred to mark-and-sweep collectors. Its collection time is proportional to the size of reachable data and not to heap size [Zorn 90].

Table 2.3 summarizes the impacts of mark-and-sweep for comparison purposes.

	RC	MS
Pause Time	v	g
Throughput	v	g
Promptness	e	v
Completeness	-	+
Space Overhead	g	v
Unused Space	b	b
Fragmentation	b	b
Locality	b	b

Meaning
(b)ad, (g)ood, (v)ery good, (e)xcellent
(-) not provided, (+) provided

Table 2.3 Points of Comparison for Mark-and-Sweep

Let us describe these impacts in more detail.

Fragmentation, Pause Time, and Throughput

Mark-and-sweep collectors may suffer from memory fragmentation as they do not move objects. Often, designers implement algorithms to compact objects during a collection to recover from fragmentation (see Section 2.5). This strategy usually produces longer pauses. Mark-and-sweep collectors generally provide good program throughput and short pause times, however [Hertz 05b]. One of their disadvantages is that collecting overhead is proportional to the size of memory, which can be large in modern systems [Zorn 90].

In [Ben-Yitzhak 02], the authors propose a technique to reduce pause times. Their collector incrementally compacts small regions of the heap via copying. However, it uses additional space during compacting to store root set entries therefore increasing space overhead. It also requires many marking passes over the heap in order to compact it completely. This can lead to poor performance and poor throughput.

Locality, Unused Space, and Space Overhead

Mark-and-sweep collectors use free-list allocators which often provide poor object locality and bad memory use. However, a proper allocator may exploit the memory

space efficiently.

A mark-and-sweep algorithm combined with a compacting strategy improves the spatial locality of objects and reduces the amount of unused space. Compacting strategies require object relocation and add space and performance overhead to the algorithm [Jones 96]. Nonetheless, mark-and-sweep collectors do not need a copy reserve to hold survivors of a collection. This minimizes the space overhead, although the recursive traversal during the marking phase incurs a space overhead (see Chapter 5).

Completeness and Promptness

The mark phase visits all reachable objects and marks them as live. The sweep phase passes through all objects in memory, adding those not marked to the allocator's free-list. A mark-and-sweep algorithm does not reclaim dead objects as fast as reference counting does, but it reclaims them completely by marking all of the live objects.

2.5 Mark and Compact

Many systems carefully avoid fragmentation to maintain a good level of performance. There exists a negative correlation between memory fragmentation and overall performance. Allocation policies, such as free-lists, may increase fragmentation as well as collection strategies, such as mark-and-sweep. All collectors should have a strategy to combat memory fragmentation. Two strategies commonly used by collectors are compacting and copying. Let us first describe the compacting strategy followed by the copying strategy.

Compacting normally consists of two major activities: moving objects and updating their references. Collectors which implement a compacting strategy move objects to reduce fragmentation. They slide live objects toward one end of the heap to create a large contiguous free area at the other end. In this process, they update all of the object's references to refer to their new locations.

	RC	MS	MC
Pause Time	v	g	b
Throughput	v	g	b
Promptness	e	v	v
Completeness	-	+	+
Space Overhead	g	v	g
Unused Space	b	b	e
Fragmentation	b	b	e
Locality	b	b	v

Meaning
(b)ad, (g)ood, (v)ery good, (e)xcellent
(-) not provided, (+) provided

Table 2.4 Points of Comparison for Mark-and-Compact

Fragmentation may happen with many garbage collection algorithms. Compacting is often an efficient way of reducing fragmentation when using a mark-and-sweep collector [Ossia 04]. However, the collector usually visits all the live objects twice during a collection; it passes through all the live objects to mark them, and then visits them again to update their references. This may result in an important performance overhead.

Table 2.4 summarizes the impacts of the mark-and-compact collector for comparison purposes. Now, let us describe these impacts in more detail.

Fragmentation, Pause Time, and Throughput

Collectors compact objects at the expense of longer pauses. Some authors [Barabash 03; Ossia 04] propose a method of executing reference updating concurrently with the mark phase, thus eliminating a substantial portion of the pause times. Nevertheless, this method is detrimental to the throughput of the mutator and is most suitable when added to a mark-and-sweep garbage collector.

Space Overhead

A mark-and-compact collector does not need a copy reserve to maintain survivors. This minimizes the space overhead. Nonetheless, the collector recursively traverses the

tree of objects during the mark phase. This recursive traversal may produce some space overhead (see Chapter 5). Mark-and-compact collectors often maintain a table of offsets to realize the relocation of objects [Jones 96], thus increasing memory pressure.

Zorn [Zorn 90] proposes to simplify updating moved objects by adding a level of indirection to all object references. Object references refer to a table of object handles, which refer to the actual objects in the heap. When an object is moved, only its object handle shall be updated. This approach adds a notable space and performance overhead.

Locality and Unused Space

Mark-and-compact collectors almost completely eliminate the amount of memory unused by compacting. Furthermore, they preserve the locality of objects allocated closely in time. Compacting collectors do not reorder objects. Consequently, mutators accessing objects allocated together usually obtain a better overall performance.

Completeness and Promptness

A mark-and-compact collector reclaims garbage completely as it marks live objects. It also recycles dead objects promptly, not with delays as reference counting does.

2.6 Semi-Space

Semi-space collectors [Cheney 70; Blackburn 02a] are the simplest copying collectors. All copying collectors must reserve sufficient memory to maintain all the possible survivors of a collection. This reserve must be large enough to accommodate the worst case scenario, which happens when all objects survive. At collection time, copying garbage collectors move all live objects to the copy reserve. The memory collected is then recycled and fed back to a contiguous allocator.

Copying collectors are often called stop-and-copy collectors as they stop the exe-

	RC	MS	MC	SS
Pause Time	v	g	b	b
Throughput	v	g	b	g
Promptness	e	v	v	v
Completeness	-	+	+	+
Space Overhead	g	v	g	b
Unused Space	b	b	e	e
Fragmentation	b	b	e	e
Locality	b	b	v	v

Meaning
(b)ad, (g)ood, (v)ery good, (e)xcellent
(-) not provided, (+) provided

Table 2.5 Points of Comparison for Semi-Space

cution of the mutator while copying. Objects are allocated until all the usable space in the area has been exhausted. Program execution is then stopped and the heap is visited. Live objects are copied to the copy reserve as they are encountered during the traversal. Program execution resumes only when the copy procedure is finished. Memory is then allocated from the area which holds the survivors, and the other free area becomes the copy reserve.

Objects are copied as they are discovered during the traversal from the root set. While objects are copied to the copy reserve, forwarding pointers are left in their old locations. The objects encountered later in the traversal, that refer to already copied objects, can use the forwarding pointers to obtain the new locations of the copied objects. A simple test can establish whether a pointer refers to the old or the new region of memory.

Table 2.5 summarizes the impacts of semi-space for comparison purposes. We describe now these impacts in more detail.

Fragmentation and Locality

Semi-Space collectors place live objects side by side into the copy reserve during copying. By doing so, they eliminate fragmentation and further improve spacial locality. However, they reorder objects as they follow pointers breadth-first or depth-first instead

of in order of age. This reordering may engender a locality that does not fit the access pattern of the mutator, and consequently degrade the overall performance. We shall investigate this problem in Chapter 8.

Space Overhead and Unused Space

Semi-space collectors fail to minimize space overhead. They need a copy reserve that consumes half of the heap space. Collector designers propose new algorithms which uses the memory more efficiently. Generational and incremental collectors have emerged among other solutions. We present them in sections 2.7 and 2.8 respectively. There remains an unused space at the end of the usable memory when a collection is triggered.

Throughput and Pause Time

Semi-space collectors provide a good throughput when the heap is correctly sized. When the heap is too small, the collector collects too often, thus the throughput degrades. When the heap is excessively large, its collection requires more time causing longer mutator pauses. Pauses, however, are generally long because every collection must trace all the reachable objects.

Completeness and Promptness

During every collection, semi-space collectors copy all live objects into the copy reserve. They feed back the memory holding all dead objects to the allocator. Thus they reclaim all garbage completely. After each collection the heap contains live objects only. However, it recycles dead objects less promptly than reference counting does.

2.7 Generational

Generational collectors divide the heap into regions independently collected. A region contains objects of similar age and is referred to as a generation. The youngest

generation, where newly created objects are recorded, is known as the nursery. The older generations, holding collection's survivors, are usually referred to as the mature space.

Figure 2.2 presents a typical heap organization for a generational garbage collection. It shows a heap holding three generations. Distinct systems may require more or fewer generations. All generations may have distinct sizes [Ungar 84; Appel 89; Blackburn 02a].

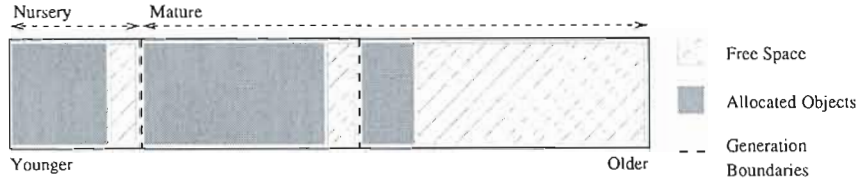

Figure 2.2 Heap Layout for Generational GC

The generational algorithm exploits the *weak generational hypothesis*, which asserts that most object die young [Hayes 91]. On that account, the algorithm most frequently recycles the nursery. It collects less often the older spaces assuming that older objects live longer. By doing so, it usually reduces copying costs and improves performance.

A generation is the unit of generational collection. At least one generation, always the nursery, is available for allocation after every collection. Sometimes two or more generations are collected at once. In many virtual machines the Java method *System.gc()* launches a full collection which scavenges all the heap for garbage [IBM 03; Gagnon 03b].

Most generational collectors interact with a contiguous allocator that places new objects into the nursery. They mainly use the semi-space algorithm to collect the youngest generation and copy all nursery survivors into the copy reserve of the older generation. Generational collectors may use other algorithms to collect the mature space.

2.7.1 Write Barrier Mechanism

In order to collect generations independently, generational collectors use a write barrier which keeps track of the inter-generational references. These references, while collecting, are used as roots. At every statement which stores a reference into an object field, the write barrier updates an inter-generational reference data structure, called the remembered set. The latter provides the collector all the references from objects in older generations to objects in younger generations. The references from younger to older objects or from and to same-aged objects are usually not remembered as younger objects are collected before older ones (see Figure 2.3).

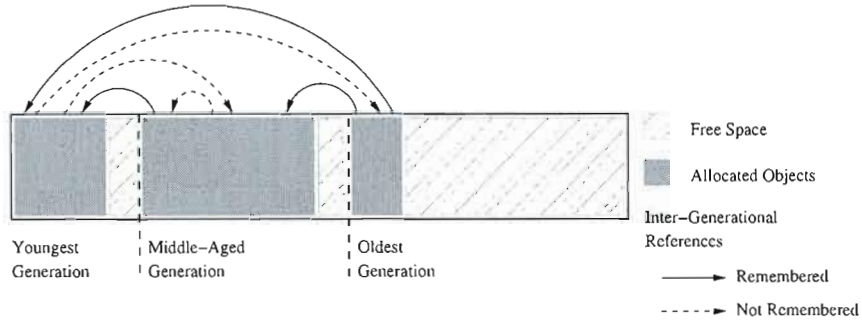

Figure 2.3 Inter-Generational References

The write barrier mechanism maintains remembered sets to avoid scanning the entire heap at collection time. During collection, the collector conservatively assumes that any remembered pointer refers to an object that the mutator can reach. Instead of tracing all the heap to find reachable objects, the collector sequentially visits all the references used as roots for the generation being collected.

Since the write barrier is called by the mutator after each reference is stored into the heap, this mechanism is very expensive. The global performance depends on the frequency of pointer stores, the number of stores remembered, and the benefits from independently scavenging regions. Altogether these tradeoffs improve performance because the pointer-tracking cost is offset by a much reduced copying cost

[Stefanović 99c; Tarditi 93].

Blackburn decomposes the write barrier into two parts: the fast path and the slow path [Blackburn 02b]. The fast path is typically short. It determines if the collector needs to remember a pointer update. The slow path records the pointer only when necessary. The write barrier implementation depends on the remembered set scheme, and often uses many instructions [Hölzle 93; Wilson 89a]. To reduce the write barrier cost, some collectors try to minimize the number of remembered pointer stores [Appel 89; Stefanović 99b]. Other collectors use an imprecise remembered set to trade off scanning time for a simpler unconditional barrier [Azagury 98; Wilson 89a].

2.7.1.1 Remembered Set

Many garbage collection systems depend on partitioning objects and need to handle references between various partitions. Keeping track of such references in a remembered set eliminates the need to scan the originating partition to find them. Such a system may vary in precision: an imprecise system requires the collector to do more work while tracing. Generally, a more precise remembered set requires a more elaborate write barrier.

We usually associate a remembered set with each generation [Ungar 84]. Any pointer stored that creates a reference from an older generation to a younger generation is recorded in the remembered set of the younger generation. At collection time, the remembered set of the collected generation is scanned. Each pointer in the remembered set is considered a root. Maintenance of this set is done by the mutator and by the collector when objects are promoted [Jones 96; Wilson 92]. Since this approach has an unbounded space overhead [Hosking 92; Hertz 05b], some studies have been done to provide alternatives.

As described by Hosking, Moss, and Stefanovic in [Hosking 92], one general approach is to keep all the tracked references in the remembered set. At collection time, the collector uses and rebuilds the remembered set, discarding any entries that do not

contain interesting pointers. Such entries can appear when a running system is imprecise about what is considered interesting, or when later changes override interesting pointers with uninteresting data.

As mentioned by the authors, an imprecise system attempts to put too many entries into the remembered set rather than too few. The system must allow the collector to find all the interesting pointers. A naive implementation of this technique may lead to an important space overhead. In the worst case scenario, the remembered set size can grow to be as large as the heap itself [Sachindran 04]. One strategy they propose implements the remembered set as a circular hashtable using linear hashing. They filter all pointer stores to keep only the interesting roots in the remembered set.

Advantages and Drawbacks

The advantage is that this strategy eliminates duplicated entries and bounds and can reduce space overhead even if only a portion of the table is full. Another apparent advantage of remembered sets is their conciseness and accuracy. A drawback is that the system may have to do a circular search to find an empty slot. This linear search must be added to the filtering and hashing processes, which respectively eliminate uninteresting pointer stores and duplicates. The overall performance is markedly affected. However, remembered sets countervail this flaw by allowing less root processing than other schemes do.

2.7.1.2 Card Marking

Many current systems use an efficient write barrier and maintain an imprecise remembered set. Card marking techniques maintain imprecise remembered sets [Azagury 98; Hosking 93; Wilson 89a]. Card marking partitions the heap into cards of equal size. Whenever the mutator modifies an object in a card, the card is marked as dirty. At collection time, the collector must scan all dirty cards to find the inter-generational pointers. This technique has the advantage of causing a fixed space overhead, unlike the

unbounded remembered set. However, it increases the tracing cost since the collector must trace all the dirty cards in order to find the roots.

Advantages and Drawbacks

All the card marking mechanisms have the benefit of fixing and at times reducing the space overhead required for recording inter-generational pointers. By unconditionally setting the appropriate table's entry at each store in the heap, we also improve barrier time. We defer the cost of checking references until scavenge time and check each location only once for each scavenge.

Systems need to scan more space in order to find the roots in the marked cards. Cards are unmarked only when they are collected. A marked card is scanned repeatedly. Depending of the size of the card, a lot of space is being traced needlessly thus increasing tracing cost. This strategy also constrains the allocator to place objects within the boundaries of a card. We must ensure that the first word of a card is a header word which allows collection. There is unused space at the end of each card. The object size is also bounded by the card size.

2.7.2 Points of Comparison

Table 2.6 summarizes the effects of generational copying for comparison purposes. Now, let us describe them in more detail.

Completeness and Promptness

Some systems collect the oldest generation with a mark-and-compact algorithm, others use mark-and-sweep, even reference counting is occasionally applied. Not all these techniques ensure the completeness of the collectors. Designers must pay close attention to ensure that all dead objects are reclaimed at some point in time.

Older generations are visited less often than the nursery, and dead objects may

	RC	MS	MC	SS	GC
Pause Time	v	g	b	b	v
Throughput	v	g	b	g	g
Promptness	e	v	v	v	b
Completeness	-	+	+	+	-
Space Overhead	g	v	g	b	b
Unused Space	b	b	e	e	v
Fragmentation	b	b	e	e	g
Locality	b	b	v	v	g

Meaning
(b)ad, (g)ood, (v)ery good, (e)xcellent
(-) not provided, (+) provided

Table 2.6 Points of Comparison for Generational Copying

not be recycled promptly. In Chapter 5 we present a new technique to mark live objects. In Chapter 7 we use this technique to provide promptness and completeness in a generational context.

Fragmentation, Locality, and Unused Space

Generational collectors minimize fragmentation and provide spacial locality as they place survivors side by side after the collection of a generation. They often collect the oldest generation using a non-moving algorithm, producing some fragmentation, reducing spacial locality, and increasing the amount of unused space. The amount of unused memory increases when generational collectors use card marking as the size of cards limits the size of objects allocated in the heap [Azagury 98; Hosking 93; Wilson 89a].

Pause Time, Space Overhead, and Throughput

Generational collectors reduce the average pause time. They do not collect all of the heap. They reclaim younger objects regularly and older ones less regularly since older objects are presumed to live longer. There remains an unused space at the end

of the usable memory of each generation when a collection is performed. Properly tuned generational collectors provide good throughput. However, older objects survive into older generations. The latter are typically larger and eventually require collection. Collecting older generations worsens latency and throughput. Generational collectors do not ensure constant throughput and maximum pause times.

2.8 Incremental

Stop-the-world algorithms, as proposed by Cheney [Cheney 70], completely halt execution of the mutator to perform a collection. Stopping the mutation guarantees that objects are not allocated or do not suddenly become unreachable while the collector is running. The fact that the mutator can perform no work while a collection is processing is a disadvantage. Within the context of interactive or real-time systems or when maintaining a large heap, such a pause may become intolerable.

Incremental garbage collectors are designed to reduce this disruption by interleaving their work with the activity of the mutator. Instead of scavenging the entire heap at once, collectors divide the heap into increments, which are usually equal-sized and independently collected. A possible heap shape for this scheme is shown in Figure 2.4. Increments are contiguously illustrated but this is not a requisite.

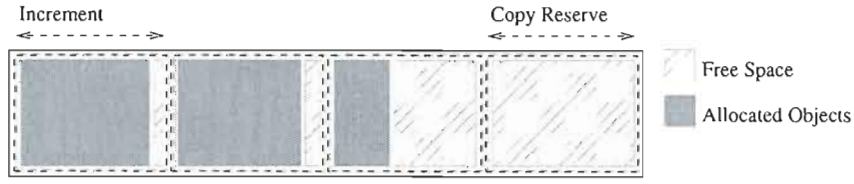

Figure 2.4 Heap Layout for Incremental GC

The number of increments may vary from one system to another. An increment is always the unit of collection. At least one increment is made available for allocation after each collection. Many increments can be collected and consequently freed at once.

Unlike the generational collection, the incremental algorithm does not constrain

	RC	MS	MC	SS	GC	IC
Pause Time	v	g	b	b	v	v
Throughput	v	g	b	g	g	v
Promptness	e	v	v	v	b	g
Completeness	-	+	+	+	-	-
Space Overhead	g	v	g	b	b	v
Unused Space	b	b	e	e	v	g
Fragmentation	b	b	e	e	g	v
Locality	b	b	v	v	g	g

Meaning
(b)ad, (g)ood, (v)ery good, (e)xcellent
(-) not provided, (+) provided

Table 2.7 Points of Comparison for Incremental Copying

the allocator to place new objects within a particular increment. Objects can be allocated into any increment. Collectors often use a copying strategy to scavenge each increment [Bishop 75; Hudson 92; Moss 96; Munro 99; Blackburn 02a]. Survivors are copied into another increment. All increments are repeatedly filled and collected using the same algorithm. Incremental collectors, like generational ones, need to keep track of pointers between increments in order to reduce tracing costs. A remembered set is maintained for each increment while processing.

Table 2.7 surveys the impacts of incremental copying for comparison purposes. We describe them now in more detail.

Completeness and Promptness

Incremental collectors have a disadvantage: they do not ensure the collection of cyclic structures. The incremental copying collector proposed in Chapter 4 reclaims garbage completely. Since each increment is collected in turn, dead objects are not always recycled promptly. The incremental collector, as depicted above, visits each increment as often as the other increments. Thus, it may reclaim garbage more quickly than generational collectors. In Chapter 5 we present a new technique to mark live

objects.In Chapter 4 we use this technique to provide promptness and completeness in an incremental context.

Fragmentation, Locality, and Unused Space

Incremental collectors minimize fragmentation and provide spacial locality as they place survivors of a collection side by side. There is some unused space at the end of each increment. This unused space may increase when incremental collectors use a card marking mechanism.

Pause Time, Space Overhead, and Throughput

Incremental collectors reduce the average pause time as they partly collect the heap. They need to reserve less space than semi-space collectors to maintain survivors and thus reduce space overhead. Finally, they usually provide a very good throughput by collecting small increments at once.

2.9 Conclusion

In this chapter we described concepts of such as a mutator, a memory manager, an allocator, and a collector. We further presented the concepts of roots, garbage, and live objects. We presented points of comparison often used in the literature on garbage collection. Some popular garbage collection algorithms have been detailed and compared. Finally, some related constructs as the write barrier, the remembered set, and card marking have also been presented.

Chapter III

OLDER-FIRST ALGORITHM

Bishop's work [Bishop 75] introduced the idea of collecting memory incrementally. He describes a technique to divide the memory into many areas and uses remembered sets to track references between them. His collector is able to collect large structures by migrating objects to areas containing references to these objects and by imposing no bounds on the sizes of the areas. It then becomes straightforward to collect a structure isolated within a single area or even cyclic garbage structures (see Chapter 5).

The *train algorithm*, as first proposed by Hudson and Moss [Hudson 92], also implements an incremental collection. It addresses a weakness found in generational collection algorithms which results in long and disruptive pauses when collecting the oldest generation. The basic idea is to divide the oldest generation into equal-sized increments and collect them independently. Increments, also called *cars*, are logically linked into diverse trains, which symbolize lists of cars. Trains are then linearly ordered and collected in turn. The order of the trains reflects the moment in time of their creation from oldest to youngest. Cars, added to a given train as objects, either leave or join the train.

The interest in this technique resides in its capacity to bound the size of the train being collected at any one step. This capacity gives the authors the hope of restraining pause times. The ability to collect large and cyclic garbage structures is guaranteed through conscious placement policies that retain a proper logical ordering of the trains.

Seligmann and Grarup have implemented and studied this algorithm [Seligmann 95]. Extensions have also been proposed for persistent [Moss 96; Munro 99] and distributed [Hudson 97] environments.

Other collectors also use incremental algorithms. The older-first collectors [Stefanović 99a; Stefanovic 02] segregate objects by age into a number of equal-sized windows (or increments) which they collect one by one from the older to the younger objects. Older-first collectors avoid copying the youngest objects, which have not yet had enough time to die. Many implementations of the older-first algorithm have been proposed over the years. In this chapter we carefully examine this incremental algorithm.

3.1 Basic Implementation

Figure 3.1 presents the possible shape of a heap worked on using an older-first algorithm. It helps to visualize a first-in-first-out circular queue of windows. The first window containing the older objects is next in line to be collected. Collections always happen at the head of the queue when objects consume all usable memory excluding the unusable copy reserve.

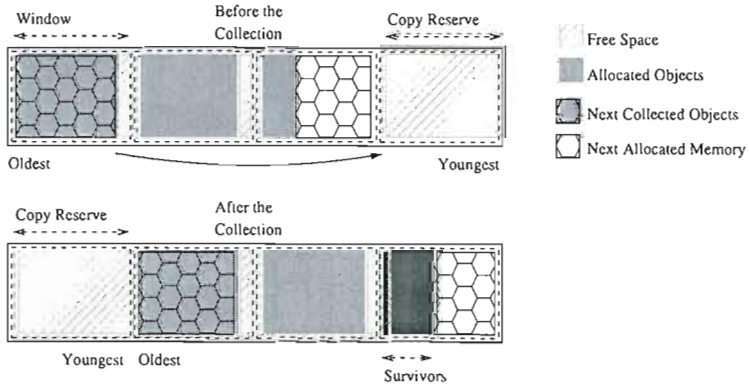

Figure 3.1 Heap Layout of the Older-First Algorithm

The leading window becomes the trailing one after its collection, the second window becomes the next to be collected, and so on. Windows are collected in circular

fashion. The tail always contains new allocated objects and survivors, which are equal-aged in the older-first algorithm. In [Blackburn 02a] the algorithm is named *older-first mix* by the authors as survivors and newly allocated objects are mixed together in memory.

3.1.1 Pointer-Tracking Cost

Collecting less than the whole heap requires tracking pointers into the collected region. The older-first collector uses write barrier mechanisms to compute the root set of each window. This usually results in much higher pointer-tracking costs than generational algorithms. The latter divide the heap into a number of generations. This number is usually less than the number of windows provided by older-first collectors. More boundaries make the write barrier remember more pointers, consequently increasing pointer-tracking costs.

Stefanovic, McKinley, and Moss [Stefanović 99b] have found that most pointer stores are among the youngest objects as well as among the objects they point to. Collecting regions outside the youngest objects causes more pointer tracking. They show that older-first collectors give objects more time to die. They do not collect the youngest objects which clearly are still alive. This results in much lower copying costs. The authors state that older-first algorithms usually have a total cost lower than the total cost of generational algorithms. The total cost refers to combined costs of the pointer tracking and copying collection phases.

3.1.2 Computing the Root Set

Remembered set strategies can significantly amplify space overhead. Older-first collectors generally record more pointers than generational collectors. If duplicated entries are not rejected, the size of the remembered sets may become problematic. This space overhead may even lead to a system failure. For example, the mark-copy algorithm have a space overhead representing up to 25% of the maximum live memory size

	RC	MS	MC	SS	GC	IC	OF
Pause Time	v	g	b	b	v	v	v
Throughput	v	g	b	g	g	v	v
Promptness	e	v	v	v	b	g	g
Completeness	-	+	+	+	-	-	-
Space Overhead	g	v	g	b	b	v	v
Unused Space	b	b	e	e	v	g	g
Fragmentation	b	b	e	e	g	v	v
Locality	b	b	v	v	g	g	g

Meaning
(b)ad, (g)ood, (v)ery good, (e)xcellent
(-) not provided, (+) provided

Table 3.1 Points of Comparison for Older-First Copying

[Sachindran 03]. Although remembered sets increase space overhead, designers use this technique for gaining time, as earlier experience suggests [Hosking 92; Blackburn 02a].

Card marking mechanisms have the benefit of reducing and fixing the root set size, thus the space overhead as well. Collectors often mark cards unconditionally to reduce time overhead. Since the older-first algorithm increases the number of pointers tracked, it pays off to simplify the write barrier. The barrier cost is reduced by using card marking. The imprecision of this mechanism increases tracing costs.

3.2 Points of Comparison

Table 3.1 surveys the impacts of older-first copying for comparison purposes. We describe them now in more detail.

Older-first algorithms provide all the characteristics of incremental garbage collection. They have short and constant pause times, reduced copying costs, less space overhead, fragmentation, and locality. However, older-first collectors visit the whole heap more regularly than generational collectors. This may decrease its locality in the cache and increase its paging activity [Stefanović 99b].

Older-first collectors improve responsiveness by reducing average pause times. The problem is that they do not provide completeness, causing memory leaks when cyclic and large garbage structures infest the heap. In Chapter 5 we present a technique to recycle garbage completely in a partitioned heap.

3.3 Conclusion

In this chapter we presented a basic implementation of the older-first algorithm. We explained how the algorithm negatively affects pointer-tracking costs as it collects less than the whole heap. We discussed the strategies (remembered set and card marking) available for computing the root set. We also briefly discussed the points of comparison for this algorithm.

Chapter IV

IMPROVING CARD MARKING USING BOUNDED FRAMES

In this section we present a new implementation of the older-first algorithm which uses a bounded frame marking scheme as the remembered set. We first introduce the bidirectional object layout exploited in our algorithm. We then present our remembered set approach. Next, we talk about the necessity to use the bidirectional object layout with our approach. We then discuss the efficiency of our write barrier, which maintains the remembered set. Finally, we explore existing remembered set approaches, their advantages and disadvantages, and the interests and drawbacks of our implementation.

4.1 Traditional and Bidirectional Object Layouts

Many studies have demonstrated that the layout of an object exerts an influence on its environment. Among other consequences, it can affect the execution time as well as the algorithm design. For instance, the object layout has been investigated as a means to provide efficient access to instance data and dispatch information in languages supporting multiple inheritance (most specifically C++) [Myers 95; Pugh 90]. Someone else has proposed a garbage collector which requires grouping pointers at the head of structures [Bartlett 88]. Gagnon has introduced a new object layout that optimizes the placement of reference fields to allow efficient garbage collection tracing [Gagnon 02a]. The most exploited object layout is the traditional layout. However, the bounded frame marking method that we present next uses the bidirectional layout. Then we present both object layouts: traditional and bidirectional.

In the traditional layout the fields are laid out consecutively after the object header, starting with super-class fields then subclass fields, as shown in Figure 4.1.

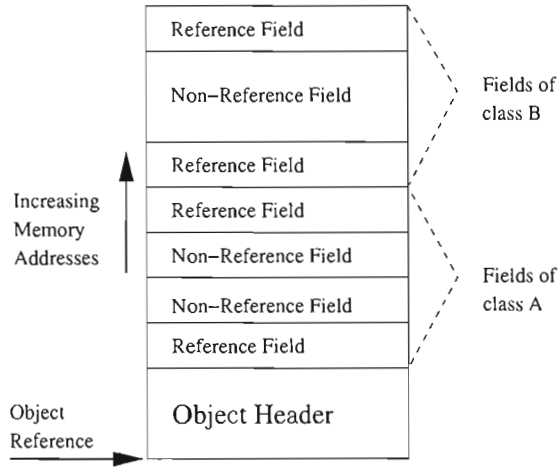

Figure 4.1 Traditional Object Layout

When tracing such an object, the garbage collector must access the object's class information to obtain the offsets of its reference fields, then access the superclass information to obtain the offsets of its reference fields, and so on. Since this process must be repeated for each traced object, it becomes quite expensive [Gagnon 03a].

The bidirectional object layout was first introduced in SableVM [Gagnon 03b; Gagnon 02a; Gagnon 03a], a research framework for efficient execution of Java bytecode. This layout groups all reference fields consecutively in front of the object header and all non-reference fields following the object header. The garbage collector is then freed of the burden of accessing the object's class information in order to obtain the offsets of its reference fields. In array instances, elements are placed in front or after the array instance header, depending on whether the element is a reference or a non-reference type. Figure 4.2 shows the bidirectional layout of an object.

With bidirectional layout, an object can be reached by a tracing collector through a reference that points to the object header or through the starting point of that object.

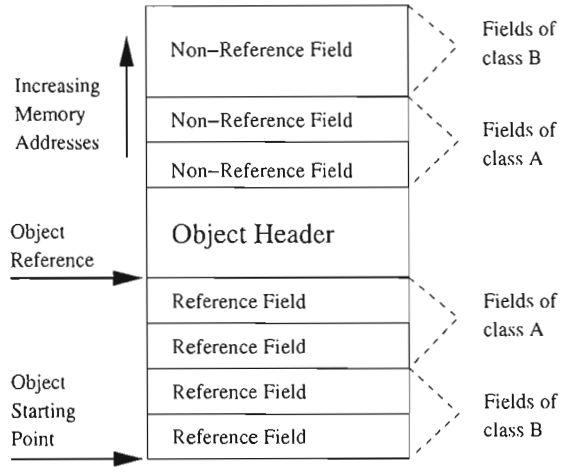

Figure 4.2 Bidirectional Object Layout

In the second case, the starting point might be either a reference field or the object header itself. At this point, the collector determines whether the initial word is a reference or a header word. If the last bit of the word equals 1, then the word is a header word. If it is zero, then the word is a reference since all references are aligned in memory.

While scanning the heap, the collector only needs to read words consecutively. It then checks the last bit of each word. When that bit is set to zero, the reference is traced. When it is set to 1 (i.e. object header), the end offset of the object is computed in order to find the starting point of the next object.

4.2 Bounded Frame Marking Scheme

Our older-first garbage collector exploits the bidirectional object layout to improve efficiency and precision when remembering inter-window references. We propose a method that combines the remembered set and card marking mechanism in a new way.

In our implementation, the windows are subdivided into many cards as proposed

by the card marking mechanism. But, the new idea behind our bounded frame marking technique is to keep two pointers for each card in a remembered set. This pair of pointers represents the first and last remembered pointers for a card. We use the addresses of each pointer to order them from first to last. The first pointer has the lowest address in a card and the last has the highest. Figure 4.3 represents our improved older-first garbage collector implementing the bounded frame marking mechanism.

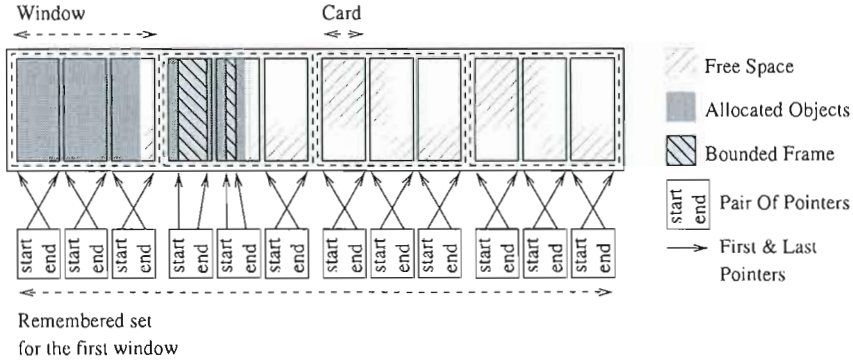

Figure 4.3 Bounded Frame Marking Mechanism

As you can deduce when observing Figure 4.3, each window has its own remembered set. The first window's remembered set is presented in the figure. This remembered set contains many pairs of pointers, one for each card in the heap. Each pair keeps track of the first and last pointers in the corresponding card that potentially points into the first window.

At collection time, the collector traces each card from the first to the last remembered incoming pointers for the window being collected. These pointers determine the region that must be traced into the card. We call this region a *frame*. Since the first and last pointers always belong to the card, we say that the frame is bounded by this card. When the last pointer is lower in value than the first one, the card is considered unmarked. Then, our bounded frame marking technique combines the card and remembered set mechanism by using cards to bound frames and pointers to delimit frames.

4.3 Bidirectional Layout Dependency

Our mechanism depends on the bidirectional object layout. Remember that the bidirectional object layout lays out all the references in front of the object header while the traditional layout spread them after the object header with the non-reference fields.

We should also remember that our mechanism maintains, for each window, a remembered set that keeps track of incoming pointers. To be more precise, what we save in the remembered set is the address of the first and last incoming pointers. Figure 4.4 shows this fact.

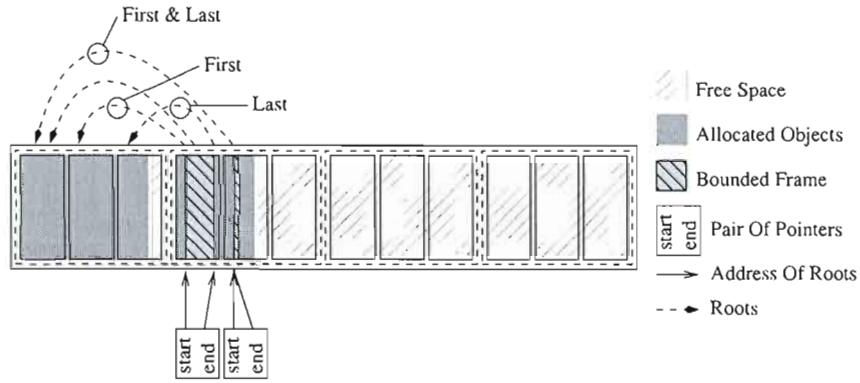

Figure 4.4 First and Last Incoming Pointers for a Bounded Frame

When the collector traces a frame to find the roots of reacheable objects in the window being collected, it starts at the address of the first remembered pointer and stops at the address of the last one. While scanning the frame, the collector needs only to read words consecutively and check the last bit of each word as previously explained.

All the reference fields between the first and last remembered pointers are traced. In Figure 4.4, the collector finds three roots in the first frame and one in the second. When a header word is found, we use it to skip all subsequent non-reference fields by computing the starting point of the next object. This starting point always refers to a reference field or a header word.

With the bidirectional object layout we are sure that the word after the first remembered pointer is either a reference field or a header word. But, the problem with the traditional object layout is that it cannot ensure what comes before and after the first remembered pointer. Thus, the collector cannot determine if the subsequent word is a reference field, a non-reference field, or a header word. This limitation of the traditional object layout does not allow the collector to scan the memory frames. So our method is dependent on the bidirectional object layout.

4.4 Write Barrier Efficiency

The older-first algorithm is incremental. So, to collect each increment independently we have no choice but to track the inter-window pointers. At collection time, we use these tracked references as roots to find the reachable objects. The write barrier is the collector's element which maintains this set of references. It is called by the mutator each time a reference is stored in the heap. Nonetheless, it is also called by the collector each time an object referenced by another object considered as a root is moved in the heap.

The number of calls to the write barrier is quite significant. Furthermore, older-first collectors usually track many more pointers [Stefanović 99b], thus, the importance of examining the efficiency of this component. It has a major impact on overall performance. This realization has guided us in the conception of our older-first garbage collector. Figure 4.5 shows the pseudocode for our write barrier.

To understand the pseudocode, one must have a good understanding of the bounded frame marking mechanism previously presented. Remember that each window maintains its own remembered set. This set allows the collector to find all the roots for the window being collected. For each window, we have a number of frames equal to the number of cards in the heap. This fact is illustrated in Figure 4.6.

When an object field is updated, the write barrier is called. While in the write barrier, we first verify if the value stored in the field is indeed a reference to another

```
_svmf_write_barrier (heap, slot)
{
    IF  the object referenced is in the heap  THEN

        compute the index of the frame to update
        // index_frame = ((slot – heap–>start) / heap–>frame_size)

        compute the index of the window to update
        // index_window = ((*slot – heap–>start) / heap–>window_size)

        access the frame to update
        // frame = heap–>windows [index_window] . frames [index_frame]

         IF  slot < frame–>start  THEN
            frame–>start = slot;
        END IF

        IF  slot > frame–>end  THEN
            frame–>end = slot;
        END IF

    END IF
}
```

Figure 4.5 Write Barrier Pseudocode

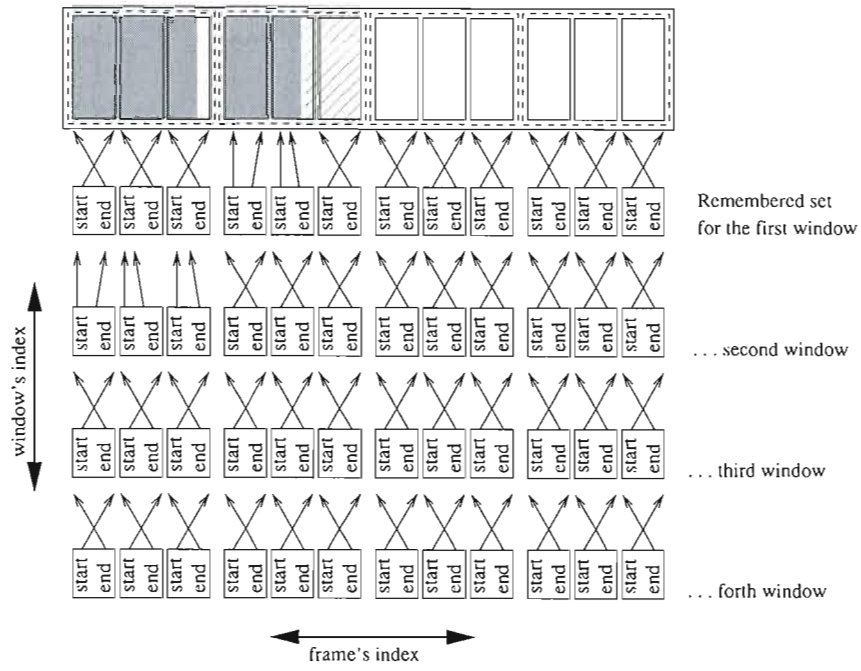

Figure 4.6 Remembered Sets for the Bounded Frame Marking Method

object in the heap. In that case, we compute the index of the bounded frame associated with the card that contains the field being updated. After that, we compute the index of the window that contains the object referenced by the field being updated. These indexes are represented in Figure 4.6. Then, we access the remembered set to obtain the pair of pointers that delimits this bounded frame. At that point, we update the first and/or the last pointers if necessary. All these operations are executed in constant time. So the execution time of our write barrier is very fast.

4.5 Comparison with Card Marking

Our new bounded frame marking scheme provides all the characteristics of the older-first garbage collection: short and constant pause times, reduced copying costs, less space overhead and fragmentation, and locality. But, it has some advantages and few drawbacks over the card marking mechanism.

One of the most signicant advantages of our method is to reduce the tracing costs of garbage collection. As pointed out by Jones and others in [Jones 96; Hosking 92] respectively, tracing is often one of the most expensive steps of garbage collection. Remember that with a card marking system whenever the mutator modifies an object reference field in a card, it calls the write barrier to mark the card as modified. At collection time, the collector scans all the marked cards to find pointers that point into the window being collected. These cards are unmarked only when they are collected. So, a marked card is scanned repeatedly. Depending of the size of the card, a lot of space is being traced uselessly hence increasing the tracing cost.

Our method is more precise when marking a card. If only one word is modified in a card, the corresponding frame will be bounded by this word. So, the collector will scan only one word at collection time instead of the entire card. This situation is presented in Figure 4.4 (the first and last pointers contain the same address). The worst case scenario happens when the first and last words of a card are modified and represent some references. Then the entire card is traced. But even in such a case our technique

is still as efficient as card marking.

Another advantage of our mechanism is to reduce the amount of unused heap space. With the card marking system, each marked card is scanned entirely at collection time. In addition to increasing the tracing cost, this strategy constrains the allocator to place an object inside the boundaries of a card. An object cannot spread across a card boundary since we must ensure, in order to allow the collection, that the first word of a card is a header word (or a reference word if the bidirectional object layout is used). Thus there is some wasted space at the end of each card of the heap.

Since tracing does not necessarily start at the beginning of a card, our method allows to spread objects over many cards, unlike the card marking system. In fact, it allows an object spreading over many cards to be partially traced, as shown in Figure 4.7.

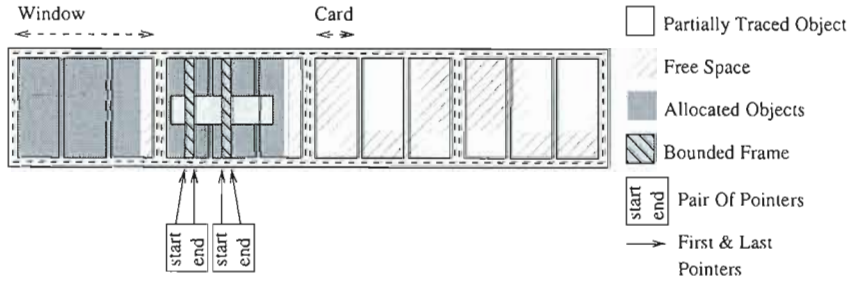

Figure 4.7 A Partially Traced Object

The collector traces only the parts of the object that potentially contain some source pointers. But, what we want to expose here is the fact that each card is entirely allocated. The only exception is the last card of each window, the unit of collection of the older-first garbage collector. We cannot spread an object over a window since the collector would corrupt the object at collection time. This limitation is not addressed within our system. So, there is still some unused space but only at the end of each window.

This discussion introduces the next advantage of our technique; the object size is no longer limited by the size of a card. As mentionned before, the traditional card

marking system does not allow an object to spread over many cards. Thus the size of an object is limited by the size of a card. Our bounded frame marking scheme eliminates this constraint by allowing an object to cover many cards. So, the object size and the card size are no longer dependent. Nevertheless, the object size is still limited by the size of a window, a constraint introduced by older-first algorithms.

These advantages come with at least two disadvantages. First, instead of using one bit by card as the card marking mechanism does, we use two words by card representing the frame's first and last pointers. Also, our method causes a space overhead that increases in a quadratic manner. Suppose the number of cards is n and the number of regions collected independently is m. In that case, the collector maintains (n * m) frames because each region has its own remembered set which contains a frame for each card.

The second point concerns write barrier costs. As explained before, our technique forces the write barrier to compare the first and last pointers with the pointer presently being stored. The card marking system does not need to make these comparisons since it only updates the bit no matter the exact address of the pointer being stored. On the other hand, it needs to manipulate the bit vector. This implies that the needed bit's position must be computed, this bit must be then reached, and finally updated. Depending on the specific processor, 3 to 6 instructions are needed to realize these operations [Hölzle 93; Wilson 89a]. Thus the overhead resulting from our comparisons is compensated by these bit manipulations.

4.6 Comparison with Remembered Set

Our new bounded frame marking method combines the card marking and remembered set techniques. This combination provides our system with at least one advantage over systems implementing only the remembered set mechanism. Our method reduces and fixes space overhead. Remember that the basic remembered set strategy uses one word for each store into the heap that represents a pointer from a younger to an older

window. This means that the size of a window's remembered set can be proportional to the number of references in the heap. If the strategy implemented does not track the duplicated entries (i.e. many stores into the same location), the size of the remembered set can grow significantly. However, the remembered set grows in a linear manner which is better than our earlier-stated quadratic complexity.

In our system, instead of using one word for each significant store into the heap, we use two bytes by card. Thus, we provide our system with a fixed space overhead that is potentially smaller than the one provided by the basic remembered set scheme. On the other hand, our method is less precise when tracing. We must scan more space to find the source references in a frame. Thus, our tracing cost is higher. Nevertheless, this drawback is less important for our method than for the card marking scheme.

4.7 Conclusion

In this chapter we introduced a new method that uses bounded frame to reduce the tracing costs of traditional card marking schemes. We described the bidirectional object layout which is exploited in our scheme. We discussed the need for an efficient write barrier mecanism, and explained how this need has guided us in the conception of our method. Finally, we compared our bounded frame marking technique with traditional remembered set and card marking strategies. Empirical evaluations of our method is given in Chapter 10.

Chapter V

DEALING WITH CYCLES, LARGE GARBAGE STRUCTURES, AND FLOATING GARBAGE

The incremental algorithm proposes to divide the heap into fixed-sized regions and incrementally collect each region in turn, bounding the amount of data copied at each collection step. The problem is that some structures may never be collapsed into the boundaries of a region, causing either multiple collections of the same garbage or memory leaks. The amount of garbage structures may represent up to 80 percent of all dead objects with certain applications [Adjih 96].

In this chapter, we closely examine some garbage structures, and present how collectors actually handle them. Then, we present a technique which provides completeness by marking live objects without space overhead.

5.1 Garbage Structures

Incremental collections as depicted in Chapter 3 fail when dealing with garbage spreading over regions not collected at the same time. Large garbage structures, cycles, and floating garbage stand as examples, let us introduce them now.

5.1.1 Large Garbage Structures

As an example, suppose we have a single-linked list of objects which overlay many increments. The list is placed in such a way that its tail is first collected, then comes

the element which precedes the tail, and so on backtraking through the list until the collection comes upon the head. Figure 5.1 presents the idea. In this illustration, increments are collected from left to right. Reachable objects are copied into the empty increment while collecting.

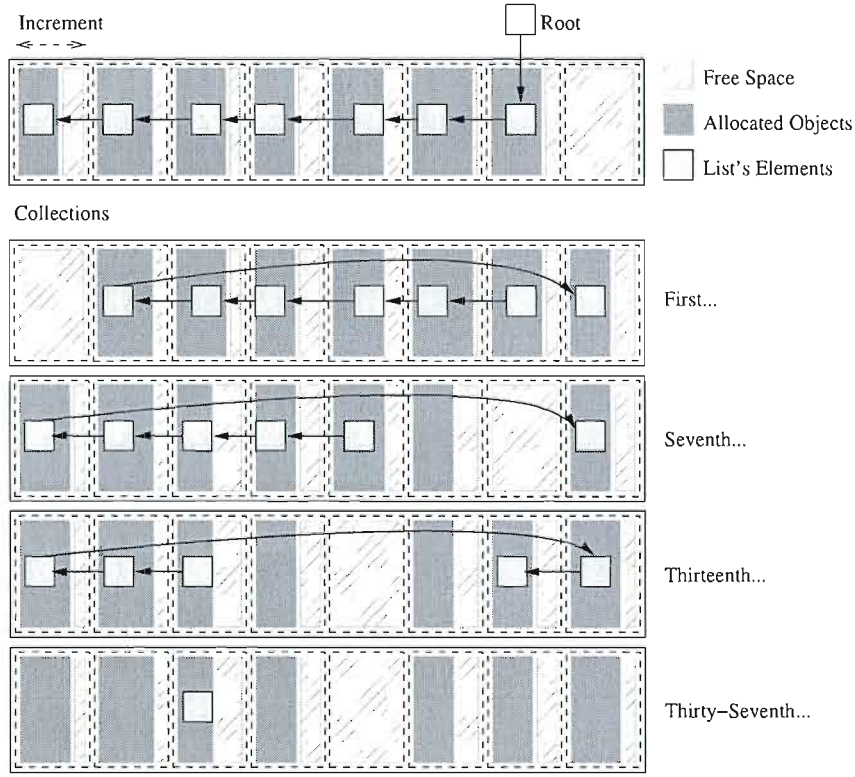

Figure 5.1 A Large Garbage Structure Overlaying many Increments

One root maintains the list away from collection. As Figure 5.1 shows, freeing the entire list may request collecting forty-three increments if only the root is nullified. The linked-list's elements are copied several times causing a negative effect on overall performance. The result is not a memory leak. Nonetheless, this temporary memory shortage may lead to premature out-of-memory errors.

5.1.2 Cycles

Cyclic structures may spread over many increments, and may never be freed if they cannot be collapsed into a single region. Look at the circularly-linked-list pictured in Figure 5.2.

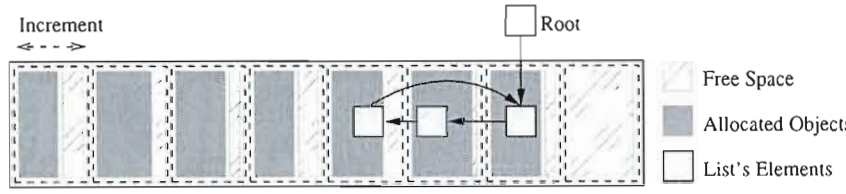

Figure 5.2 A Cyclic Structure Overlaying many Increments

If only the root which maintains the list away from collection is nullified, freeing the entire list becomes impossible. This memory leak occurs because each element is still reachable from the remembered set and thus can not be collected. By copying elements over and over again, cyclic structures negatively affect overall performance. Furthermore, this memory leak may lead to premature out-of-memory errors.

5.1.3 Floating Garbage

Garbage collectors are conservative when assuming that all objects in the remembered set are alive. However, objects may die after they have been inserted into the remembered set but before being collected. When the dead objects in the remembered set refer to other dead objects in the collected region, those dead objects are copied by the collector. They are called floating garbage, or simply floats [Hansen 02]. Cycles and large garbage structures may generate a lot of floats.

5.2 Providing Completeness

Collectors often mark the object graph starting from the root set, and conserve only marked objects while collecting. By marking objects, collectors ensure that all

garbage is detected. A mark stack is commonly used to avoid procedural recursions, this adds extra space and time overhead while marking. In the worst case, such a strategy may generate a space overhead proportional to the number of object references populating the heap.

Many systems have severe heap restrictions combined with complex functional requirements, which consume memory as a time-space trade-off. Embedded systems stand as an example; mark stacks may result in a premature out-of-memory error with systems severely limited in space. So, it is clearly undesirable to have garbage collectors consume too much memory while marking.

5.3 Marking without Space Overhead

We introduce here a technique to mark objects without space overhead. Figure 5.3 illustrates the method we used. Upon closer examination, one can see that we perform a depth-first policy.

We can imagine a forest which contains many trees. We use the root set to find these trees. Starting at the root, we go deeper and deeper down the first branch of the tree until we bump into a leaf, which is an object either without a handy reference or which is already marked. We consider marked objects as leaves to ensure the completeness of our algorithm which would loop over cyclic structures otherwise. When a leaf is encountered, we go back up to recurse into the next branch, and so on until the tree is completely marked. By processing the root set entirely, we thus mark all reachable objects.

Step a) in Figure 5.3 presents the starting point of our algorithm. As you can see, we maintain two pointers while marking: a parent which first points to the root of the tree, and a child which points to the first reference field of the object reachable from the root. We use these pointers to repair each branch we alter while traversing the tree.

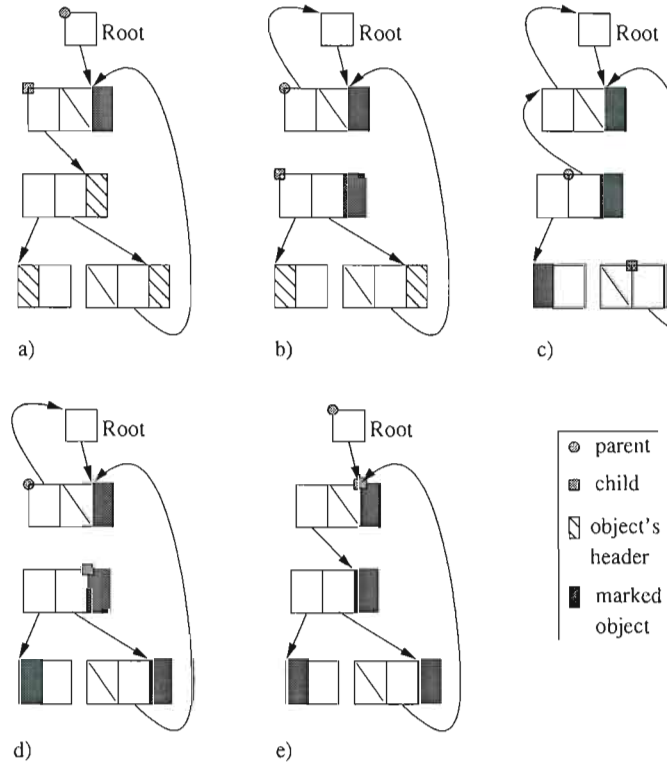

Figure 5.3 Mark without Space Overhead

5.3.1 Depth-First Marking Trace

Now, let us describe the process in detail using the example illustrated in Figure 5.3. Figure 5.4 reveals the pseudocode of our marking procedure, which receives a root as parameter. Our algorithm first checks if the root either is null or refers to an object already marked, in both cases we return immediately. Otherwise, the object is marked and we set the parent and child pointers as illustrated in step a) of our example. The remaining steps take place in the core loop of our procedure.

Our algorithm loops while there is a child to visit, or while the parent is not the root. In our example, the root refers to an object which has potentially two children, so we enter the loop. In the loop, we initially verify if the child pointer indirectly refers

```
IF the root is null or the object is marked THEN
    return
ELSE
    mark the object
    refer to the root using the parent pointer
    refer to the first child of the object using the child pointer
    WHILE there is a child OR the parent is not the root LOOP
        IF there is a child THEN
            IF the child is not marked THEN
                mark the child
                IF the child has another child THEN
                    go down the tree
                END IF
            END IF
        ELSE
            go back up the tree
        END IF
        refer to the next child of the parent using the child pointer
    END WHILE
END IF
```

Figure 5.4 Pseudocode for the Marking Procedure

to an object. This is the case, so we check if this object is already marked. As you can see in step a), the object is not marked. Therefore, we mark the object, and we go on down since this object has a child. In order to go back up later, we must remember where we came from when we go down. Normally, it is this information that yields space overhead. In our scheme, we use the heap space to stock this information, avoiding the space overhead.

Figure 5.5 presents what happens when we go down. We first remember the child pointer in a swap variable (see a.1). Next, we set the child pointer to the first reference field of the object indirectly referenced by the child pointer itself (see a.2). Then, we use the reference field of the object we leave behind to remember where we need to go when we will go back up later (see a.3). Finally, we set the parent equal to the swap variable. All these operations occur between steps a) and b), and as well each time we go down into the tree.

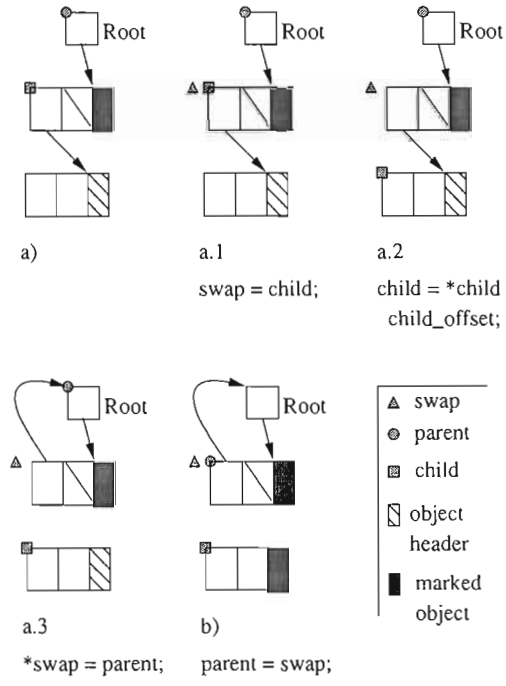

Figure 5.5 Going Down while Marking

All the following actions happen between steps b) and c). Starting from step b), we use the child pointer to access the next object, which is not marked. We mark it, but we do not go down since this object does not have a child. So, we move the child pointer over to the next reference field. Again, we use the child pointer to access the next object, which is also not marked. We mark it, but now we go down since this object has potentially two children. However, the first child is null, so we skip it. We are now at step c).

In step c), we bump into a leaf which is an object already marked. Since there are no more children, we have to go back up. Figure 5.6 illustrates what happens when we go back up. We first set the swap variable equal to the reference field pointed to by the parent pointer (see c.1). Next, we set this reference field equal to the child pointer (see c.2). Then, we set the child pointer equal to the parent (see c.3). Finally, we set the parent equal to the swap variable, and we move the child pointer over to the next

consecutive reference field. All these operations occur between steps c) and d), and each time we go up the tree.

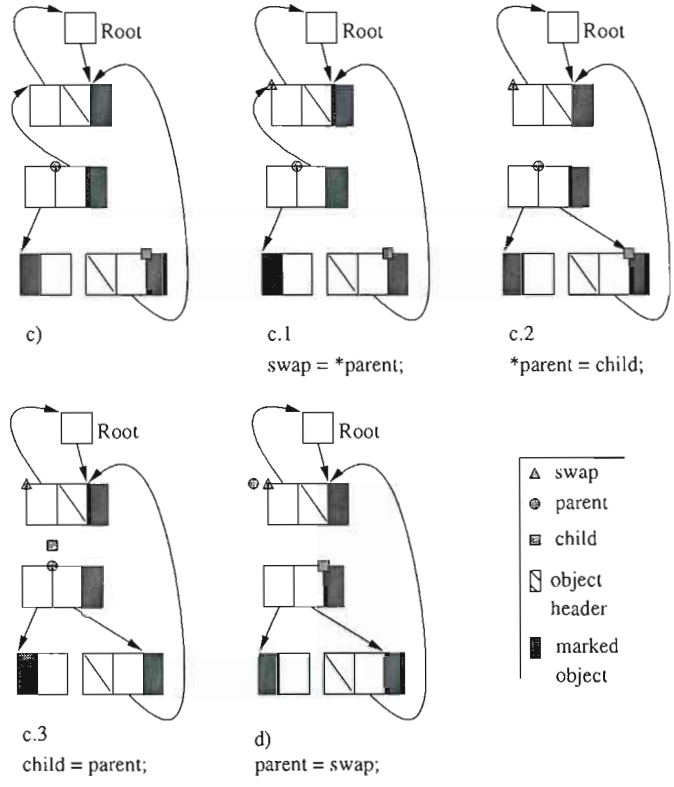

Figure 5.6 Going Up while Marking

In step d), the child pointer reaches an object header. We then must go up since no other child exists. At that point, the parent pointer refers to the root, and the child pointer refers to a nullified reference field. Thus, we move the child pointer over to the next word, which is also an object header. This last move brings us to step e), we then reach the end of the marking process.

5.4 Conclusion

In this chapter we introduced garbage structures such as cycles, large structrues, and floats. We discussed the space overhead generated by collectors which use mark

stacks to provide completeness. We proposed a technique to mark objects and provide completeness without space overhead. Finally, we presented a complete trace of our method using a small example.

Chapter VI

DEALING WITH LARGE OBJECTS

Collector designers make assumptions about the lifetime of objects, and they aspire to improve collection algorithms by tailoring them to these assumptions. In [Inoue 06], the authors define the lifetime of an object in garbage collection studies as the sum of the sizes of all objects allocated between the given object's allocation and death. They express its lifetime in bytes or words.

Most garbage collectors make relatively coarse-grained predictions (e.g., short-lived versus long-lived) and rely on general heuristics to predict the lifetime of objects [Lieberman 83; Hanson 90]. For instance, older-first collectors copy older objects first, and generational collectors copy the younger generation more often than the older one. Other systems implement more precise predictors which make their predictions based on application-specific training rather than application-independent heuristics [Inoue 03; Inoue 06].

Accurately predicting the lifetimes of objects can effectively improve memory management systems, but wrong assumptions may result in poor performances. A predictor is accurate when a great fraction of its predictions are correct, and it is precise when the granularity of its predictions becomes equal to a small unit of allocation. A fully precise predictor has a granularity of predictions equal to the smallest possible unit of allocation. Usually, predictors must trade off between accuracy and precision, because increasing precision often leads to less accuracy.

Many policies have been used to prophesize the lifetime of objects. For instance, many copying garbage collectors assume that large objects are long-lived. On that account, they handle large objects specially in a separate space to reduce the costs of managing them. Many implementations employ this strategy [Caudill 86; Hudson 91; Reppy 93], some of which do not collect the large objects at all, and some of which collect them using diverse algorithms.

In this chapter, we describe our large object policy that regroups large objects in memory and makes assumptions about their lifetime. Large object segregation implies lack of promptness caused by delays in the collection of dead objects. In Section 6.2, we present a solution to that problem. Finally, we introduce a study that provides some helpful tips for managing large object spaces.

6.1 Large Object Policies

Predictors often use the characteristics of an object to predict its lifetime. Usually, object size is a good property to consider while predicting. Many systems segregate small and large objects into the heap, and employ distinct policies to manage them [Ungar 88; Hudson 92; Baker 92; Lim 98]. By isolating large objects into a specific area, we can handle them more efficiently, and studies show that this pays off [Hicks 98].

We follow the assumption that large objects are generally long-lived, and so we suggest collecting them less often than small objects. On that account, our older-first collector allocates small objects at the beginning of the heap, and large ones at the end. We maintain a reference pointer to the first word of the large object space. Initially, no large objects exist, so the pointer refers to the end of the heap. When the mutator requests a large block of memory, the pointer moves over a lower address. Figure 6.1 illustrates the heap layout we propose for our large object policy.

The large object space is always aligned with the boundary of a window, and it may spread over one or more windows. When the mutator creates a large object, we widen the large object space if necessary, and narrow the small object space accordingly.

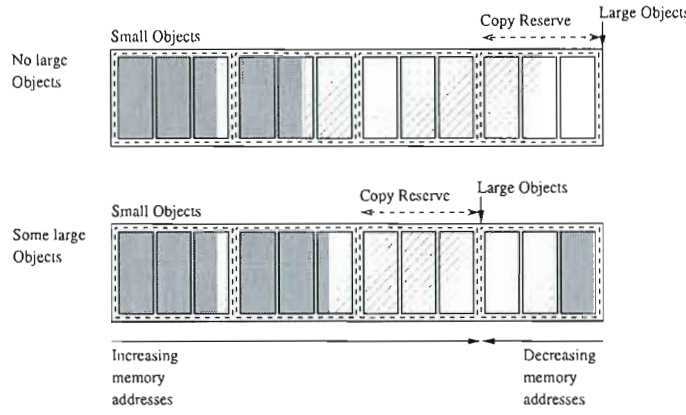

Figure 6.1 Heap Layout for Large Objects Policy

We always keep at least one window free as a copy reserve for our small object space. Such a reserve is needless for the large object space, we collect it only when the mutator exhausts the heap. In that case, we enlarge the heap and launch a full collection, possibly extending both the small and large object spaces. During a full collection we compact small objects at the beginning of the heap, and large objects at the end.

6.2 Improving Promptness

By copying only small objects over and over, we reduce the copying cost of the collector. We move large objects exclusively when a full collection occurs. However, if we take too long to reclaim large-sized garbage, we risk copying dead objects repeatedly. Many small objects shall be preserved only because they are referred to from objects that are undetected garbage. This lack of promptness may negatively affect overall performance, although this does not appear to be a problem in practice [Ungar 84].

We correct this deficiency using our mark algorithm described in Chapter 5. When the small object space becomes full, our older-first collector marks all reachable objects in the heap, just before the first minor collection occurs. Unmarked objects stay in place during copying, and since the window is fed back to the allocator after its collection, the unreachable memory consequently becomes freed. On the other hand, marked objects

are moved at collection time, and each one is unmarked after its displacement. The next marking step is triggered only when all of the small objects have been unmarked. For that reason, the collector must collect the small object space entirely before marking once again.

Large objects are collected and thus unmarked only when a full collection occurs. However, the collector may execute several marking steps before such a collection is initiated. Since our mark algorithm treats marked objects as leaves, the large objects may corrupt the process. We unmark all large objects before each marking step to remedy this situation. We maintain a table of pointers to rapidly find those objects while unmarking.

6.3 Large Object Space

A study of large object space is done in [Hicks 98]. This research provides some guidance about the best ways to implement a large object space policy. The authors examine the design space for copying garbage collectors in which large objects are managed in a separate space. They focus on how to determine the policy for classifying objects as large, and how to manage the large object space.

They compare the performance of a treadmill collection to that of a mark-and-sweep collection for managing the large object space. Their conclusion is that for some heaps there exists a minimum threshold below which adding objects to the large object space does not generate better performance, while for others no such cutoff exists. They also find that the exact method used to collect the large object space does not significantly influence overall performance.

6.4 Conclusion

In this chapter we suggested a segregation strategy to manage large objects which we assume to be longer lived. We proposed a way to fight lack of promptness caused by our strategy. We also introduced a study that provides some guidance on how to

implement a large object space policy.

Chapter VII

GENERATIONAL OLDER-FIRST ALGORITHM

Generational copying garbage collectors more frequently collect the youngest objects, and copy any survivors to a mature space which is collected less often. Many empirical studies have shown that such a younger-first policy usually outperforms non-generational collectors [Ungar 84; Appel 89; Hayes 91; Blackburn 02a]. In practice, young generations often contain a higher fraction of unreachable objects because most programs satisfy the *weak generational hypothesis*, which assumes that young objects die at a faster rate than older objects. Each collection consequently reclaims more garbage space.

7.1 Floating Garbage

Hansen and Clinger [Hansen 02] have measured the amount of floating garbage produced by their generational collectors. They found that promotion floats are often high, and comparable between collectors with either two or three generations. In the worst case, their collectors attain 48.5 percent of promotion floats, copying 278.7 Mb of garbage. The copying cost is therefore seriously affected.

7.2 Giving Objects Time to Die

Baker [Baker 93] used a model of object lifetimes to demonstrate that younger-first collectors would surprisingly perform worse than non-generational collectors when

most objects die young. In his model, each object has a 50% probability of being already dead when a collection happens. By processing objects which have not yet had sufficient time to die, Baker shows that generational strategies do not recover much storage at once. An older-first policy, which proposes to collect old objects more often than younger ones, would recover more storage for a similar amount of effort.

Older-first collection shows promise by collecting older objects first, and then giving younger ones enough time to die. Stefanovic, McKinley, and Moss use a garbage collection simulation to point out potential improvements by using an older-first algorithm [Stefanović 99c; Stefanović 99b], although older-first collectors collect older objects over and over again, which is costly.

Generational collectors do not manage the youngest objects efficiently, and older-first collectors fail to exploit the fact that older objects are generally longer-lived. We consider next a generational older-first (GOF) collector that combines the profits of both generational and older-first collectors to improve garbage collection.

7.3 3-GOF Collector

We propose here our 3-GOF collector which exploits the high mortality rate of young objects, avoids collecting the youngest objects, avoids collecting previously copied objects, and performs its collection incrementally. 3-GOF exploits the advantages of both the generational and older-first collectors. It partitions the heap into three generations, and manages them using the older-first algorithm presented in Chapter 4. Figure 7.1 shows how the 3-GOF collector organizes the heap.

As shown in this illustration, each generation contains many windows. The window is the unit of collection. When a window from a younger generation is collected, the surviving objects are copied into the next generation. When the oldest generation is collected, survivors are moved into another window in the same generation. Therefore, the youngest generation does not maintain a copy reserve, but the older generations hold the survivors, and hence we reserve a window from each older generation as a copy

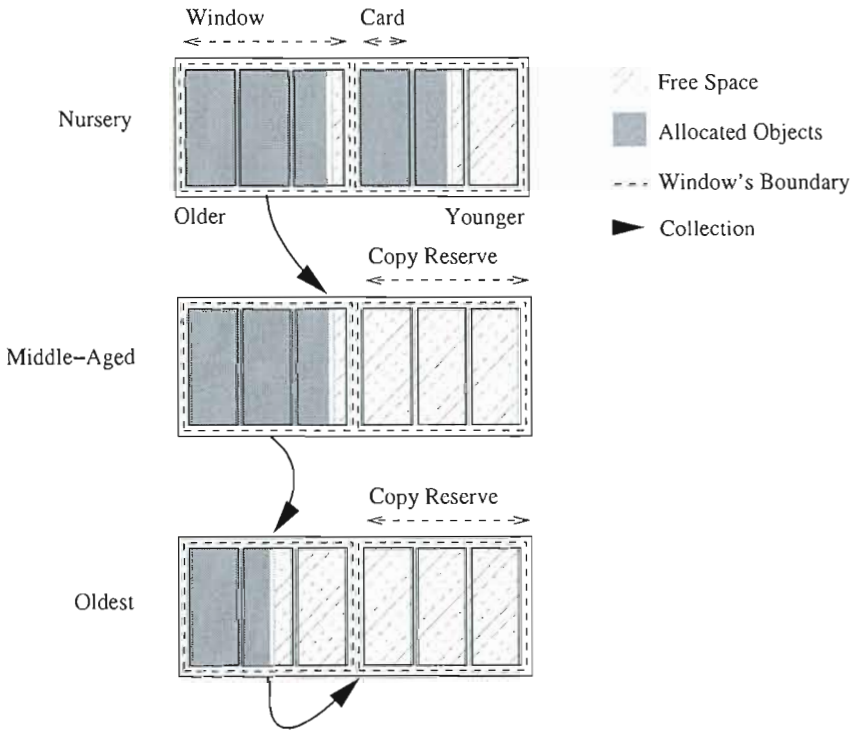

Figure 7.1 Heap Layout for the GOF Collector

reserve.

The older-first collector assumes that survivors are the youngest objects in the heap after a collection. However, the GOF collectors consider the survivors as the youngest objects in the generation which contains them.

7.3.1 Tracking Root Pointers

As discussed in Chapter 2, generational and older-first copying collectors trace roots to find reachable objects at collection time. In Chapter 4, we proposed a new method which combined card marking and a remembered set to track root pointers in our older-first collector. We employ the same technique in our GOF collector.

The GOF collector divides the heap into three generations; each of them contains many windows. It collects one window at time, so it must maintain a remembered set for each window in the heap, as the older-first collector does. Remember that a remembered set contains many pairs of pointers, one pair for each logical card partitioning the heap. A pair of pointers delimits a frame of memory which is bounded by the corresponding card. While collecting a window, GOF assumes that all frames may contain some root pointers, and thus it traces them to find live objects.

7.3.2 Handling Large Objects

In Chapter 6, we presented our policy to handle large objects in the older-first collector. This policy aims to reduce the copying costs of our collector. We isolate large objects to collect them less often than small objects; we assume that large objects are usually longer-lived. We employ the same policy to manage large objects in our GOF collector. However, only the oldest generation maintains a large object space, as shown in Figure 7.1.

7.3.3 Marking Garbage Structures

We described in Chapter 5 a new technique to mark reachable objects in a depth-first order without space overhead. We use a marking step to reclaim cycles and large garbage structures as soon as possible, and thus improve the promptness of our older-first collector. We employ the same technique to manage garbage structures in our GOF collector.

By marking objects before collecting them, we never copy the floating garbage produced by the collector. We only move marked objects during a collection, so unmarked dead objects are promptly fed back to the allocator. Hence, we improve promptness by reducing the copying costs of the collector.

7.4 Conclusion

In this chapter we discussed the impacts of floating garbage and long-lived objects on the performance of traditional collectors. We presented a basic implementation of a generational older-first algorithm, which exploits the high mortality rate of young objects, avoids collecting the youngest objects, avoids collecting previously copied objects, and performs collections incrementally. We explained how the collector handles large objects and recycles garbage structures.

Chapter VIII

DEPTH-FIRST SEMI-SPACE ALGORITHM

Copying garbage collectors implement either a breadth-first or a depth-first collection. The Cheney copying algorithm [Cheney 70] is breadth-first. It is traditionally used because it does not require any extra temporary storage such as a stack. Algorithms that use a stack risk not being able to allocate sufficient memory to hold this stack. The required stack depth cannot be reliably predicted in advance since it depends on the user data structure.

As a drawback, breadth-first collection may suffer from poorer locality than a depth-first collection. It tends to group unrelated objects in memory (e.g., cousins, rather than parents and children). This is especially true when the breadth of the tree is large, for instance, when the root from which the garbage collector starts tracing is a large array of objects. It has been shown that such a lack of locality may lead to substantial performance degradation. For this reason, copying algorithms often cluster related objects together in depth-first order, thus improving object locality [Schkolnick 77; Stamos 84; Wilson 91].

In this chapter, we first describe our depth-first copying algorithm, which does not create space overhead. We then present some techniques which try to improve locality, and finally talk about some points of comparison for these algorithms.

8.1 Depth-First Copying without Space Overhead

Copy-stacks are often used to avoid procedural recursion that create extra space and time overhead while depth-first copying. Nonetheless, such a strategy may generate a space overhead proportional to the number of live objects populating the heap. In systems severely limited in space, this strategy may be unreliable because it consumes too much memory. Consequently, collector designers often use a breadth-first algorithm which does not require any temporary storage. They trade the space overhead to a lack of locality, which causes a loss of performance.

We introduce here a new technique to copy objects in a depth-first manner without creating space overhead. Figure 8.1 illustrates the method we used. As one can see, we perform the same depth-first algorithm we use for marking objects (see Chapter 5). Instead of marking the objects, we copy them. By processing the root set entirely, we thus copy all reachable objects.

8.1.1 Depth-First Copying Trace

Now, let us describe the process in detail. Step a) in Figure 8.1 presents the starting point of our algorithm. Figure 8.2 reveals the pseudocode of our depth-first copying procedure, which receives a root as a parameter. In our procedure, we first check if the root is either null or refers to an object already copied, in both cases we return immediately. An object is already copied if its header contains a forward reference, which is the address of the object's copy, as the Cheney algorithm proposes (see Chapter 2). If the object has not yet been copied, we copy it and put the forward reference in its header. Then, we set the parent and child pointers as illustrated in step a) of our example. In Figure 8.1, objects with a plain header belong in the copy reserve, and objects with a striped header still belong in the old semi-space. The remaining steps take place in the core loop of our procedure.

Our algorithm loops while either there is a child to visit or the parent is not the

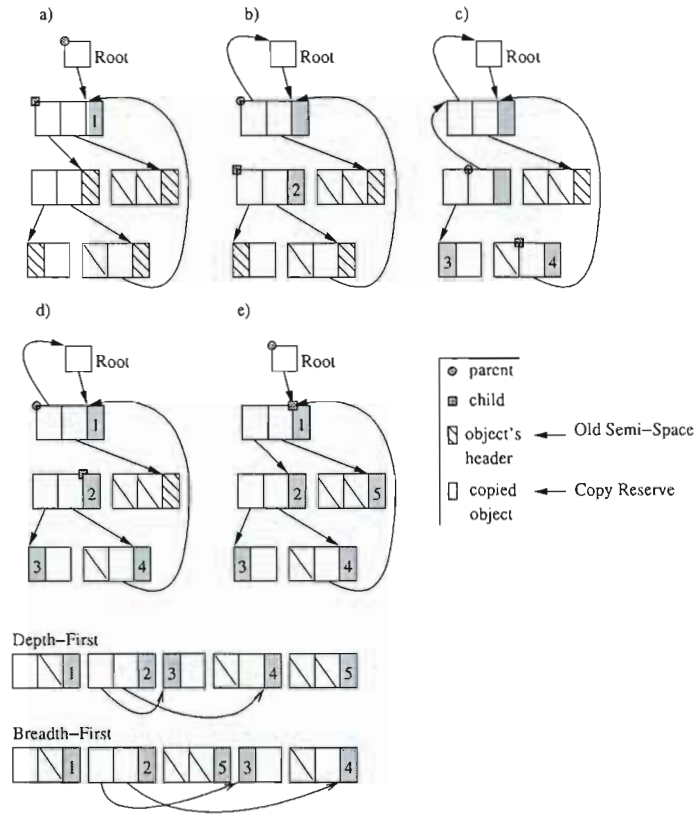

Figure 8.1 Depth-First Copying without Space Overhead

root. In our example, the root refers to an object which has potentially two children, so we enter the loop. In the loop, we first verify if the child pointer indirectly refers to an object. This is the case, so we check if this object has already been copied. As one can see in step a), the object has not yet been copied. Therefore, we copy the object, and we continue to go down since this object has a child. We are now at step b). In a breadth-first algorithm, we should move over the next adjacent child instead of going down the tree.

All the following actions occur between steps b) and c). Starting from step b), we use the child pointer to access the next object, which has not been copied. We copy it, but we do not continue on down since this object does not have a child. So, we move

```
IF the root is null or the object has been copied THEN
    return
ELSE
    copy the object
    refer to the root using the parent pointer
    refer to the first child of the object using the child pointer
    WHILE there is a child OR the parent is not the root LOOP
        IF there is a child THEN
            IF the child has not been copied THEN
                copy the child
                IF the child has another child THEN
                    go down the tree
                END IF
            END IF
        ELSE
            go back up the tree
        END IF
        refer to the next child of the parent using the child pointer
    END WHILE
END IF
```

Figure 8.2 Pseudocode for the Copying Procedure

the child pointer over to the next reference field. Again, we use the child pointer to access the next object, which also has not been copied. We then copy it, but now we do go down since the object has potentially two children. However, the first child is null, so we skip it. We are now at step c).

In step c), we bump into a leaf which is an object that has already been copied. Since there are no other children, we have to go back up. In step d), the child pointer reaches an object header. We then go back up since no other child exists. At that point, the parent pointer refers to the root, and the child pointer indirectly refers to an object. Thus, we copy that object, and we continue on down since this object has potentially two children. These childen are both null. Thus, we go back up and move the child pointer over to the next word, which is an object header. This last move brings us to step e), we then reach the end of the copying process.

8.2 Points of Comparison

At the bottom of Figure 8.1, we present the outcome of both the depth-first and breadth-first algorithms. One can see that the breadth-first algorithm places children away from their parents in memory, while the depth-first algorithm puts them next to each other. Studies have shown that some mutators tend to access objects which have references to each other closely in time [Moon 84; Huang 04], so the parent-child locality provided by a depth-first traversal may incur better performances.

Depth-first copy-stacks actually create a space overhead which may interrupt the mutator prematurely. Collector designers employ breadth-first algorithms specifically to avoid such a failure. We now propose a new method which does not require any temporary storage, avoiding the traditional space overhead. Our algorithm uses one word from each object visited to encode the copy-stack, and two pointers to remember the parent and the child currently treated.

8.3 Improving Locality

A mutator may exhibit predictable properties such as accessing objects in turn which were allocated in turn, or have references to each other. We can use this observation to improve overall performance. The locality of reference is a property often exploited in garbage collection system to give better performances.

In [Huang 04], the authors show that static copying orders result in wide variations in performance, which they consider a pathology. They propose a dynamic analysis that detects mutator traversal patterns and exploits them in a copying collector.

8.4 Conclusion

In this chapter, we described our depth-first copying algorithm. We further presented some works which try to improve locality. Finally, we discussed some points of comparison for the depth-first copying collector.

Chapter IX

IMPLEMENTATION

9.1 SableVM: A Virtual Machine for Executing Java Bytecode

All of our new techniques have been implemented on SableVM, a virtual machine for executing Java bytecode [Gagnon 03b; Gagnon 02a; Gagnon 03a]. SableVM is intended as a research framework for efficient execution of Java bytecode. This framework is essentially composed of five main components: interpreter, verifier, class loader, native interface, and memory manager.

This experimental framework publicly available is written in the C programming language. It has been designed to be a robust, extremely portable, efficient, and specification-compliant Java virtual machine. Its source code is easy to maintain and extend. This makes SableVM an ideal framework for testing new high-level implementation features or bytecode language extensions.

9.2 Memory Management Framework

Memory management is a critical issue for an increasing number of applications. There is no one correct way to configure heaps, collectors, and allocators. The best choice depends on how the application uses memory as well as on the user requirements, and so JVM's default garbage collection choices may not be optimal. In this section, we present a memory management framework that can be customized to specific application needs.

We have integrated a flexible memory manager in SableVM. It has been designed to provide a great level of reusability, modularity, portability, and performance, and enables to test a variaty of allocators, collectors, and write barrier mechanisms. The framework is easy to extend and provides a clean and meaningful experimental platform. It is implemented in C without any special support added to the language or the compiler.

9.3 Available Collectors

Currently, the framework provides five garbage collectors (breadth-first semi-space, depth-first semi-space, generational, older-first, and generational older-first), but only one contiguous allocator. All the collectors interact with this allocator. Nonetheless, the framework offers a platform to implement new collectors and allocators.

Our collectors share all the common mechanisms, policies, and functionalities, such as root processing, copying, tracing, allocation, and collection mechanisms, and use the exact same implementation, allowing to obtain very accurate experimental results. The breadth-first semi-space collector implements Cheney's algorithm [Cheney 70], and the depth-first collector implements the algorithm presented in Chapter 8. The generational collectors work with three generations of fixed and possibly distinct sizes. One implements the basic algorithm described in Chapter 2, the other puts into practice the generational older-first algorithm proposed in Chapter 7.

9.3.1 Older-First

The older-first collector implements the basic scheme presented in Section 3.1. By default, the collector maintains the root set using the bounded frame marking mechanism described in Chapter 4. It also provides completeness by marking objects as explained in Chapter 5. Finally, it assumes that large objects are longer-lived, and consequently puts into practice the large object space approach explained in Chapter 6.

	RC	MS	MC	SS	GC	IC	OF
Pause Time	v	g	b	b	v	v	v
Throughput	v	g	b	g	g	v	v
Promptness	e	v	v	v	b	g	v
Completeness	-	+	+	+	-	-	+
Space Overhead	g	v	g	b	b	v	v
Unused Space	b	b	e	e	v	g	v
Fragmentation	b	b	e	e	g	v	v
Locality	b	b	v	v	g	g	g

Meaning
(b)ad, (g)ood, (v)ery good, (e)xcellent
(-) not provided, (+) provided

Table 9.1 Points of Comparison for the Older-First Mix Collector

Points of Comparison

Table 9.1 surveys the impacts of our older-first collector using the points of comparison. We further examine what this table demonstrates.

Our older-first mix collector reduces the average pause time because it collects the heap incrementally. By doing so, it improves both the responsiveness and the throughput of the mutator.

We fight fragmentation by creating the heap in one chunk of memory. We also create the structures employed to manage the heap within that chunk of memory. We thereby hope to improve overall performance by reducing fragmentation.

We reduce and control the space overhead of the older-first mix collector. By collecting only one window at a time, we reduce the copy reserve needed to hold the survivors of a collection. By combining the card marking and remembered set techniques, we control the space overhead needed to remember the root pointers. We also reduce unused space by allowing objects to spread over many cards. However, objects cannot span across many windows, and so there is some unused space left at the end of each window.

The older-first mix collector provides some locatity by keeping close together in memory objects that were allocated one after the other. The overall performance should be improved by mutators accessing closely in time objects which were allocated together. However, the large object space degrades this locality by holding large objects away from the other objects they refer to.

The bounded frames allow the collector to trace objects even partially. It thus traces less space than collectors which use card marking. However, it still traces more space than collectors which use remembered sets.

By marking objects before collecting them, the older-first mix collector can reclaim all garbage. Using this technique, it further improves promptness by reclaiming garbage as well as cyclic and large structures promptly.

Finally, the older-first algorithm causes older objects to be repeatedly collected. Studies show that in practice the mortality rate is higher for younger than older objects [Hayes 91; Baker 93; Hayes 93; Stefanović 94]. So, we should collect older objects less often to reduce copying costs. We now present a solution that is the generational older-first collector.

9.3.2 Generational Older-First Collector

The generational older-first collector is implemented as explained in Chapter 7. By default, it uses the bounded frame marking mechanism (Chapter 4), provides completeness (Chapter 5), and maintains a large object space (Chapter 6) as the older-first collector does. In fact, our generational version has been implemented using very few lines of code (approximately 200 lines), since it shares its core with the older-first collector.

	RC	MS	MC	SS	GC	IC	OF	GO
Pause Time	v	g	b	b	v	v	v	v
Throughput	v	g	b	g	g	v	v	v
Promptness	e	v	v	v	b	g	v	v
Completeness	-	+	+	+	-	-	+	+
Space Overhead	g	v	g	b	b	v	v	v
Unused Space	b	b	e	e	v	g	v	g
Fragmentation	b	b	e	e	g	v	v	v
Locality	b	b	v	v	g	g	g	g

Meaning
(b)ad, (g)ood, (v)ery good, (e)xcellent
(-) not provided, (+) provided

Table 9.2 Points of Comparison for the Generational Older-First Collector

Points of Comparison

Table 9.2 presents the effects of our generational older-first collector using the points of comparison. Let us describe what this table demonstrates.

The generational older-first collector provides the same advantages as the older-first mix. It reduces the average pause time because it collects the heap incrementally, and consequently improves both the responsiveness and the throughput of the mutator. It also provides completeness by marking objects before collecting them.

The generational older-first collector further improves promptness by reclaiming garbage as well as cyclic and large structures promptly. It also reduces copying costs by holding a large object space, and minimizes fragmentation by creating and managing the heap within one block of memory. It reduces space overhead, minimizes the amount of unused space, and fixes the size of the remembered set. Finally, it conserves the locality of objects allocated closely in time. But, the generational older-first collector does not mix survivors with newly allocated objects. We therefore believe that the generational older-first collector should preserve a better locality than the older-first mix.

The generational older-first collector has an advantage over the older-first mix:

it does not copy both older and younger objects at the same rate. It copies younger objects more often than older objects which are generally longer-lived. Although, one should remember that it does not collect the youngest objects. By doing so, we hope to reduce the copying costs of the collector.

9.4 Marking Policy

The older-first and the generational older-first collectors mark objects only when every live object is unmarked. Thus, if the objects have already been marked, the whole heap is incrementally collected before the marking step is repeated. Consequently, we reduce the copying costs by moving only the objects marked as live, and minimize the marking costs by collecting all of the heap before a marking step occurs.

There is a problem, however. The marking traversal passes through both small and large objects, but small objects are collected repeatedly while large objects are not. This means that some objects may still be marked when a marking step is launched, thus corrupting the procedure. As a solution, we propose to maintain a table of pointers which reference large objects into the large object space. When a large object is allocated, the allocator adds the object's address to the table. The table is then traversed and updated as a remembered set while we unmark large objects.

9.5 Full Collection Policy

When the whole usable heap memory is exhausted, and collections do not succeed in freeing sufficient garbage memory, the memory manager launches a full collection and widens the heap. For the semi-space and older-first collectors, this strategy is staightforward. Survivors are compacted at the beginning of the heap, and may spread over many windows for the older-first collector, which ensures that no object crosses the windows boundaries.

The generational collectors copy objects to the older generations as they usually do, but they compact them at the beginning of each generation. Objects in the oldest

generation stay in that generation, but they are also compacted. After a full collection, all of the nursery is available for allocation. Objects in the large object space are also compacted at the end of the heap. Only the oldest generation is enlarged when a full collection occurs, so the large object space can also take up more space.

9.6 Internal Write Barrier

The memory manager provides a public (or external) write barrier procedure which remembers all pointer stores, and mutators call this procedure every time they store a reference into an object field. Collectors may use this write barrier as well to update remembered sets while moving objects, but they sometimes don't. Our generational and older-first collectors work with a more efficient private (or internal) write barrier.

Collectors trace frames sequentially, starting at the lowest address of the first frame. When they begin tracing a frame, they assume that the frame contains no addresses of survivors. On that account, they store the lowest and highest memory addresses into two temporary pointers, which shall indicate how to skip this frame during the next collection. However, some objects may still survive.

When survivors are encountered, collectors copy them. When collectors copy the first survivor, the address of the object field that refers to the survivor is stored into both temporary pointers, therefore indicating that the frame now contains a survivor. If the object is not the first survivor, the address is stored only in the temporary pointer which indicates the end of the current frame. The actions performed after copying a survivor represent our internal write barrier. This internal write barrier is clearly an improvement over the external one, which executes much more instructions (see Figure 4.5).

9.7 Fragmentation Policy

The memory manager minimizes fragmentation. It reserves a large chunk of memory, and places the heap and the structures used to manage it consecutively into

that reserved chunk. The heap is placed at the lowest address, remembered sets appear next, including the table of pointers employed when unmarking large objects. The size of the table is determined by computing the maximum number of large objects which the large object space can hold, and multiplying this number by the pointer size.

9.8 Command Line Options

SableVM enables users to override the default garbage collection setup by providing command line options at compile-time. Users can thus select and tune the collector and the write barrier mechanism they wish to employ. The memory manager actually provides two methods to compute the root set, the card marking approach using a word vector, and our novel bounded frame marking scheme. Users can also customize the heap, and decide whether or not they want to compute memory management statistics.

9.9 Conclusion

In this Chapter, we described collectors that we have implemented on our memory manager framework. We further presented points of comparison for these collectors. Finally, we explained the policies and highlighted related constructs which our collectors employ.

Chapter X

EXPERIMENTAL RESULTS

Our experiments provide comparison points between all the algorithms presented so far. In this chapter, we present the platform we used to realize our experiments, and the benchmarks employed to conduct them. Then, we describe our experimental setting and show our results, which were obtained using diverse implementations of semi-space and older-first algorithms. More importantly, we discuss our results to reveal the inherent space-time trade-offs of collector algorithms.

10.1 The Test Platform

Hardware - We used two processor architectures to conduct our experiments (AMD and Pentium 4). We first performed all our experiments on a 1.86GHz Pentium II based system, with 2 Gb of RAM, 2 Mb of cache memory, and a 7,200 RPM hard disk. Then, we performed them on a 0.8GHz AMD Duron based system, with 512 Mb of RAM, 512 Kb of cache memory, and a 7,200 RPM hard disk.

SableVM - We ran our benchmarks using a modified version of SableVM 1.13, which integrates the memory manager framework described in Chapter 9. We provided SableVM with a variety of configuration parameters, which allowed us to tune collector algorithms, control write barrier mechanisms, vary heap sizes, and measure our results. We executed all our experiments using SableVM with gcc version 3.3.6, the real life brokenness features enabled, and with the direct-threaded interpreter.

Software and Operating System - All execution time measurements are based on *system + user* times returned by the GNU time command. We obtained times by executing some benchmarks on a machine running Debian GNU/Linux 3.1 (a.k.a. sarge) with kernel version 2.6.8. All daemon processes were turned off during these tests.

10.2 Benchmark programs

We have performed our experiments on a Pentium IV based workstation, running SPECjvm98 benchmarks and two object-oriented applications: Soot version 1.2.3 17 and SableCC version 2.17.3 18 .

We present here some properties of the benchmarks used to get our exprimental results.

SPECjvm98 - This benchmark suite [Corporation 98] is intended to measure the performance of Java clients, or the speed of execution by the Java Virtual Machine of Java byte codes. This suite requires basic byte code execution, graphics, networking, and I/O, but SPEC implies that former functions will normally dominate benchmark performance.

Soot - Soot is a Java optimization framework [Group 06]. It provides intermediate representations for analyzing and transforming Java bytecode, and can be used as a stand-alone tool to optimize or inspect class files. Soot is a free software licensed under the GNU Lesser General Public License.

Soot uses multiple structures and creates many objects to analyze, inspect, transform, and optimize class files. Its high allocation rate is a great behavior to exploit when configuring the memory manager. In our experimental study, we used Soot with javac 1.3 as its input. This software provides a lot of class files thus producing a higher allocation rate.

SableCC - SableCC is a Java compiler generator [Gagnon 02b]. It is an object-

oriented framework that generates compilers (and also interpreters) in the Java programming language. It consists of a deterministic finite automaton (DFA) based lexer generator, an LALR(1) parser, an abstract syntax tree (AST) builder generator, and an object-oriented AST framework generator. SableCC is a free software licensed under the GNU Lesser General Public License.

SableCC, just like Soot, uses multiple structures and creates many objects to generate a compiler front-end for a compiled grammar. Its high allocation rate is also a great characteristic to exploit when configuring the memory manager. In our experimental study, we used SableCC to compute a compiler for the Java 1.4 language. Since it is described using several lexical definitions and grammar productions, this grammar leads to a higher allocation rate.

10.3 Overall Measurements

We realized a variety of measurements to expose the space-time trade-offs of our collectors. Using many benchmarks, we computed times by taking the average of five runs on SableVM. For fair comparisons, each experiment fixed the heap size, and triggered a collection when the program exhausted all the usable memory.

When evaluating a garbage collection system, we can measure the time spent executing distinct parts of the system. These times provide another method of comparison between systems. Table 10.1 presents times that are considered during our experiments.

The pause time includes copying, tracing, and marking time. It excludes the allocation and write barrier time, which are part of the mutation time. The mutator spends the write barrier time to concurrently compute a subset of the root set used by the collector to find reachable objects.

Time	Meaning	Formula
Tracing	time spent by the collector to trace objects	
Marking	time spent by the collector to mark objects	
Copying	time spent by the collector to copy objects	
Pause	time spent by the collector to collect objects	Tracing + Marking + Copying
Barrier	time spent by the mutator either to maintain root set or to update reference counts	
Allocation	time spent by the mutator to allocate objects	
System	time spent by the system to execute the system's code	Total - Application
Total	time spent by the system to execute the program	Application + System
Application	time spent by the system to execute the application's code	Total - System
Mutation	time spent by the mutator to mutate objects	Application - Pause

Table 10.1 Times to Consider when Evaluating Garbage Collection Systems

10.4 Performance Measurements

We performed execution time measurements with SableVM to measure the efficiency of garbage collecting using our techniques.

Overall Performance Using a Large Heap

In a first set of experiments, we measured the relative performances of the breadh-first semi-space (BSS), depth-first semi-space (DSS), older-first (OF), and older-first large object space (OFL) copying collectors. In order to perform these experiments, four separate versions of SableVM were compiled with identical configuration options (heap size: 250MB, generation size: 75MB (younger) - 75MB - 100MB (older), window size: 25MB, card size: 32KB, large object size: 100KB, large object space size: 50MB). For each collector, the total execution time is reported in Table 10.2. The second is the unit of time used to present our results. Values between parentheses are used to express the length of the collectors' execution timesin comparison with both BSS's and OF's execution times.

The six columns of Table 10.2 contain respectively: (a) the name of the executed benchmark, (b) the execution time in seconds using the breadth-first semi-space copying collector, (c) the execution time in seconds using the depth-first semi-space copying collector, (d) the execution time in seconds using the older-first copying collector, (e) the execution time in seconds using the older-first copying collector with a large object space, and (f) the execution time in seconds using the generational older-first copying collector. As both platforms generate comparable results, we report only those obtained using the AMD machine.

The Depth-First Order Provides Improvements

The overall time variation between the depth-first and the breadth-first copying orders is only 2%, there is no consistent winner. Traversing the objects in a depth-first

benchmark	BSS	DSS	OF	OFL	GOF
compress	376.09	376.51	375.41	367.56 (0.97) (0.97)	365.43
db	148.60 (0.97)	147.68 (0.99) (0.96)	152.71	152.83	153.21
jack	47.40 (0.96)	47.14 (1.00) (0.96)	49.42	48.42 (1.02) (0.98)	49.57
javac	113.40 (0.92)	112.71 (0.99) (0.91)	123.81	119.20 (1.05) (0.96)	117.23
jess	84.09 (0.92)	84.19 (1.00) (0.87)	96.78	92.41 (1.10) (0.95)	85.45
mpegaudio	303.84	303.63	304.59	305.00	306.34
mtrt	102.20	101.90	102.30	103.49	102.01
raytrace	99.31	99.23	99.80	100.70	101.21
sablecc	45.33 (0.96)	45.21 (1.00) (0.96)	47.38	47.38	47.23
soot	823.52 (0.43)	807.54 (0.98) (0.42)	1896.16	909.97 (1.10) (0.47)	-

Table 10.2 GC Performance Measurements Using a Large Heap (AMD)

order improves overall performance or provides one similar to a breadth-first order for all the benchmarks.

The Large Object Space Provides Improvements

The older-first collector which segregates the large objects (OFL) provides performance improvements over the basic older-first collector. The large object space improves total execution time by up to 53% for the *Soot* benchmark, and on average by 2% to 5%, for the heap sizes we have tested. Even more, segregating large objects enables the older-first collector to achieve better performances than the semi-space collectors, the improvement is up to 3% for the *Compress* benchmark.

The Older-First Collector Produces Pathological Cases

The older-first collector provides suitable performances on average, but does not engender significant improvements on the benchmarks we have tested. The older-first

algorithm collects the same objects repeatedly, including floating garbage. Sometimes, clustering many garbage collections might prevent mutator progress over a longer period of time. For the *Soot* benchmark, we found five clusterings of garbage collections, resulting in a 57% performance overhead over the breadth-first semi-space collector. On the Pentium machine, this variation is as much as 17% on the overall performance. It is quite possible that the data cache characteristics of certain machines have some effects on overall performance.

The Older-First Collector Uses Memory Efficiently

One should remember that older-first garbage collectors need to reserve less memory space than semi-space collectors to maintain survivors and thus reduce space overhead. Results, on the *mtrt* and *raytrace* benchmarks, show that no garbage collections occur using the older-first algorithm. Only one collection occurs, for each benchmark, using the semi-space collectors.

The Write-Barrier Performance Overhead

Now we examine the pointer-maintenance costs. When using a heap size of 250 MB, no garbage collections occur on both the *db* and *sablecc* benchmarks. The total execution time includes both mutator and write-barrier times, the latter equals 0 for semi-space collectors. Recall that our collectors share all common mechanisms, policies, and functionalities, such as root processing, copying, tracing, allocation, and collection mechanisms, and use the exact same implementation. Thus, the performance overhead of the write-barrier explains the 4% variation in execution time between the older-first and the semi-space collectors.

With the *db* benchmark, 34.52 million write barriers (see code in Figure 4.5) were executed. The number of interesting pointers, which must be remembered, was 34.5 million for the older-first collector. With the *sablecc* benchmark, 16.03 million write barriers were executed. The number of interesting pointers was 15.76 million for the

older-first collector. This engenders a significant time overhead. However, benefits of reduced copying costs generate performance improvements.

The Generational Older-First Collector Reduces Copying Costs

The generational older-first algorithm does not collects the older objects repeatedly. Benefits of reduced copying costs generate performance improvements. For the *compress* and *javac* benchmarks, we found that the generational older-first collector engenders performance improvements over the older-first collector. However, the generational mechanism produces a time overhead that has some effects on overall performance. Therefore, the generational older-first collector provides suitable performances on average, but does not engender significant improvements on the benchmarks we have tested.

Impact of the Card Size on Performances

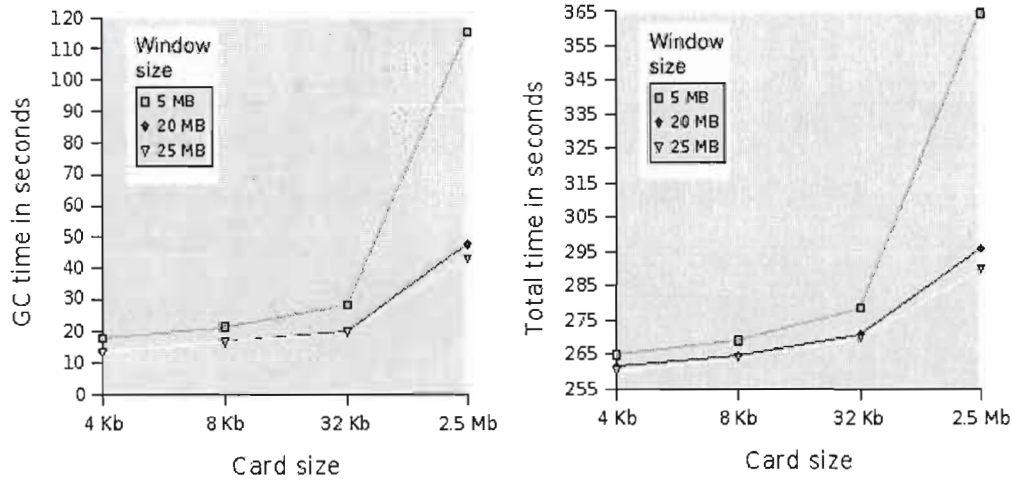

Figure 10.1 Impact of the Card Size on Performance of the Older-First Collector

Figure 10.1 compares both GC time and total time for the older-first collector running on *Soot*. On the left performance graph, the left y-axis is the GC time. On

Card Size (KB)	Data Traced (MB)
4	924
8	998
32	1227
2560	2532

Table 10.3 Average Amount of Megabytes Traced by the Older-First Collector

the right performance graph, the left y-axis is the total time. The bottom x-axis of each graph is the card size, and the values used for the plots are the window sizes. As both platforms engender comparable results, we report only those obtained using the Pentium machine.

These results show that small card sizes do indeed obtain performance improvements over large card sizes. Garbage collecting, using a large card size of 2.5 MB, produces a garbage collection time (115.37 seconds) that is up to 6.6 times higher than the garbage collection time (17.49 seconds) resulting from using a card size of 4 KB, for a fixed heap size (250 MB) and a fixed window size. When using a card size of 2.5 MB, the total time is 364 seconds. This performance is 1.38 times higher than the best performance (264 seconds) obtained using a card size of 4 KB.

A large card size degrades performance mainly by increasing the amount of data traced during a collection, as shown in Table 10.3. Garbage collection time is significantly affected by the card size with our tested benchmarks. Recall that total execution time includes the time spent in garbage collection, mutator time, and write-barrier time. For the execution time, the relative difference is not as pronounced as the garbage collection time alone. This dilution of differences is to be expected, because garbage collection time is considerably less than the mutator time, especially with larger heaps.

Impact of the Window Size on Performances

Figure 10.2 compares both GC time and total time for the older-first collector running on *Soot*. On the left performance graph, the left y-axis is the GC time. On the

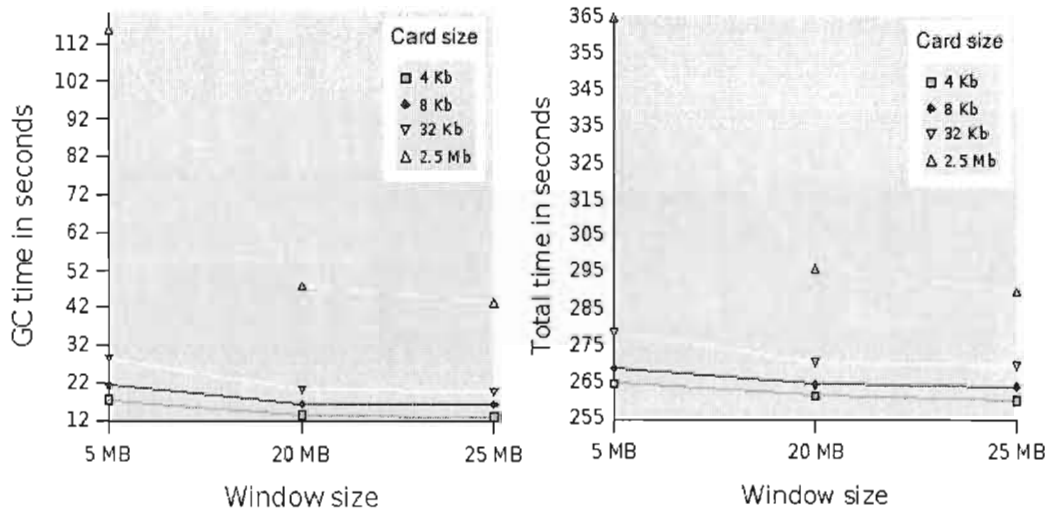

Figure 10.2 Impact of the Window Size on Performances of the Older-First Collector

right performance graph, the left y-axis is the total time. The bottom x-axis of each graph is the window size, and the values used for the plots are the card sizes. We report only results obtained using the Pentium machine.

These results show that large window sizes do obtain performance improvements over small window sizes. Garbage collecting, using a small window size of 5 MB, produces a garbage collection time (115.37 seconds) that is up to 2.68 times higher than the garbage collection time (42.97 seconds) resulting from using a window size of 25 MB, for a fixed heap size (250 MB) and a fixed card size (2.5 MB). When using a window size of 5 MB, the total time is 364.54 seconds. This performance is 1.26 times higher than the best performance (289.63 seconds) attained using a window size of 25 MB.

Garbage collection time is significantly affected by the window size with our tested benchmarks. In absolute value, the difference in garbage collection times is much bigger than the difference in execution times. A small window size degrades performance mainly by increasing both the amount of data copied during a collection and the total number of collections, as shown in Table 10.4.

Window Size (MB)	Data Copied (MB)	Object Copied	Average Object Size (KB)	Collection Count
5	1028	30958	36.38	682
20	931	189009	5.05	166
25	993	230953	4.4	136

Table 10.4 Average Amount of Megabytes Copied by the Older-First Collector

Impact of the Window Size on Copying Costs

In the presence of small windows, the amount of floating garbage appears to be significantly higher, hence increasing copying costs. Also, the objects copied seem to be significantly larger. When the window size is small (5 MB), the average size of the objects copied is 36.38 KB, which is up to 8.27 times the average size (4.4 KB) of the objects copied when a larger window size (25 MB) is used. Many studies [Caudill 86; Ungar 92; Hicks 97; Hicks 98] have already shown that large objects generally have longer lives. Our results confirm this realization.

Copying costs do not explain all of the performances obtained. The older-first collector makes many more collections when the window size is small, hence decreasing overall performance. Although a larger number of collections may be good for reducing pause times, it increases the execution time, since stacks must be scanned more often and garbage collection startup overhead occurs more frequently. Furthermore, object locality may be better when using a large window size, producing performance improvements over small window sizes.

Responsiveness of the Older-First Collector

Simple measures, such as the length of the longest garbage collection pause or a distribution of pause times, do not take into account clustering of garbage collections, which might prevent mutator progress over a longer period of time [Blackburn 02a]. However, Table 10.5 presents results obtained when executing *Soot* using a heap size of 280 MB, which does not generate clustering of garbage collections. We report only

GC Algo.	Total Time (sec.)	GC Time (sec.)	Coll. Count	Relative Av. Pause	Relative Min Pause	Relative Max Pause
DSS	253	12	39	2.65	25.12	1.58
BSS	256	15	39	3.26	35.99	2.01
OF	260	16	136	1	8.05	1
OFL	264	114	681	1.4	1	2.08

Table 10.5 Responsiveness of Collectors

results obtained using the Pentium machine.

The seven columns of Table 10.5 contain respectively: (a) the name of the algorithm used, (b) the execution time in seconds, (c) the garbage collection time in seconds, (d) the number of collections, (e) the average pause time of the collector relative to the best average pause time, (f) the minimum pause time of the collector relative to the best minimum pause time, and (g) the maximum pause time of the collector relative to the best maximum pause time.

All these collectors provide comparable overall performances. Older-first collectors offer better responsiveness and throughput than the other configurations. Since the increment size of both the large object space and the semi-space collector is larger, both the average and maximum pause times are longer. Semi-space collectors produce an average pause time that is up to 3.26 times higher than the average pause time of older-first collectors.

Overall Performance Using a Small Heap

In this set of experiments, we measured the relative performances of the breadth-first semi-space (BSS), depth-first semi-space (DSS), older-first (OF), and older-first large object space (OFL) copying collectors. To perform these experiments, four separate versions of SableVM were compiled with identical configuration options (heap size: 40MB, window size: 5MB, card size: 32KB, large object size: 5KB, large object space size: 10MB). Our results are shown in Table 10.6.

benchmark	BSS	DSS	OF	OFL
compress	98.66	102.32	96.80	94.72
db	44.76	40.86	46.16	44.69
jack	20.29	20.16	20.38	20.36
javac	37.90	37.62	38.48	38.52
jess	25.47	25.64	26.85	26.56
mpegaudio	90.15	90.63	87.81	87.79
mtrt	33.37	33.41	32.20	32.13
raytrace	32.16	32.09	31.04	31.02
sablecc	13.80	13.72	14.61	14.37

Table 10.6 GC Performance Measurements Using a Small Heap (Pentium)

The five columns of Table 10.6 contain respectively: (a) the name of the executed benchmark, (b) the execution time in seconds using the breadth-first semi-space copying collector, (c) the execution time in seconds using the depth-first semi-space copying collector, (d) the execution time in seconds using the older-first copying collector, and (e) the execution time in seconds using the older-first copying collector with a large object space. We report only the results obtained using the Pentium machine. In the presence of a small heap size, the overall time variation between the semi-space and older-first collectors is up to 3%. The older-first collector algorithm was the consistent winner.

10.5 Discussion

We believe that better implementations of the generational older-first and the older-first collection algorithms are possible. It is clear that some configurations of these collectors offer better responsiveness than others. We have not yet explored the configuration space fully. For example, the program itself and the garbage collection algorithm have different cache and memory behaviors, which interact in complex ways. Relating performance improvements to the characteristics of various benchmarks, to offer a tuning strategy, is beyond the scope of this thesis and is left for future work.

Within the limits of the effects that we have studied, the primary limitation of our experiments is the small set of heaps sampled. Over time we expect to gather heaps from

a wider range of applications. Also, we expect to implement more garbage collectors and report on them, including the generational collectors described previously, which are still not fully operational.

Finally, there are some marking policy considerations with older-first collection that we did not explore in our experiments. This is left for future work. One consideration is when to trigger the marking phase. If we mark objects too often, the marking phase significantly degrades overall performance. This is the case of the naive marking policy that we explained in Chapter 9. If we mark too soon in advance, objects have not had as much time to die, so we reclaim fewer dead objects.

10.6 Conclusion

In this chapter, we presented our experimental results which provide comparison points between the algorithms presented so far. We presented the platform we used to realize our experiments, and the benchmarks employed to conduct them. Then, we described our experimental setting and showed our results, which were obtained using diverse implementations of the semi-space and the older-first algorithms. More importantly, we discussed our results to reveal the inherent space-time trade-offs of all collector algorithms. Using our techniques, we have shown that a garbage collector can deliver competitive collection performances and even surpass that of a traditional collector on some benchmarks.

Chapter XI

RELATED WORK

Research has been done on several techniques related to garbage collection. In [Blackburn 02a], the authors identify five key insights for copying garbage collection. (1) Generational algorithms exploit the hypothesis that most objects die young. (2) They assume that older objects are longer-lived and thus collected less often. (3) Using small nurseries and incremental algorithms can improve response times. (4) Small nurseries and copying collectors can improve data locality. (5) Giving the very youngest objects time to die can improve collector performances. Studies [Caudill 86; Ungar 92; Hicks 97; Hicks 98] have further shown that (6) segregating large objects can provide performance improvements. (7) The choice of a garbage collection algorithm can improve mutator locality [Blackburn 04a]. (8) Performance-critical software can embrace modular design and high-level languages [Blackburn 04b].

In this chapter, we review some previous related works.

11.1 Garbage Collection Algorithms

In this section, we look at other implementations of both the older-first collector algorithm and the generational collector algorithm. We non-exhaustively highlight some of the most important advantages and drawbacks of these implementations.

11.1.1 Older-First Collectors

Simulation and Prototyping

In [Stefanović 99c; Stefanović 99b], the authors are among the first to put forward the older-first collector algorithm. They use a combination of simulation and prototyping to evaluate both the algorithm and the write barrier mechanism that they suggest. Their write barrier applies directional filtering to ignore useless stores. Sometimes only 5% of the total number of stores are remembered when their filter is used. The authors measured the space overhead caused by their older-first collector to be 1% of the heap size. They found that the costs of filtering and remembering stores offsets the copying cost reduction.

Their design is based on dynamically allocating fixed-size blocks to the various heap regions, using a block table to map addresses to remembered sets. This strategy may increase the fragmentation and nullify the overall performance in practice. It needs cooperation from the operating system to acquire and release address space as the heap progresses from higher to lower addresses. Furthermore, their directional filtering tactic is possible only when the collector uses an environment with a large address space. These design choices greatly reduce the number of compatible platforms.

Older-First Collector in Jikes RVM

The same authors [Stefanović 99b] join forces with others [Stefanovic 02] to put into practice, evaluate, and report on their ideas. They propose some modifications to their former design, mostly because their collector uses a 32-bit environment. The directional filtering strategy employed forces the collector to simulate a larger address space.

Their older-first collector uses an allocation region and a copy reserve. Both can be viewed as first-in-first-out queues of windows. Whenever all usable memory is consumed in the allocation region, the collector collects the oldest window, copying all

survivors to the copy reserve. When all usable space is consumed and only survivors fill the heap, the collector interchanges the roles of the two queues before collecting again. Their write barrier audits and filters pointer stores involving two windows, reducing the number of remembered pointers. A *time-of-death* value is coupled with each window and used to order the collection of windows.

The filter introduces variability and stochastic behavior in the collector. When the time-of-death values of both the allocation and the copy queues conflict, the collector collects an unfixed number of windows to reset these values. The special case, where the time-of-death value reaches the largest value allowed by the operating system, is eliminated.

Their study shows that older-first collectors can perform as the simulation results suggest. The collector they propose does not bound pauses, increases write barrier costs, and fails to provide completeness and promptness. Therefore, their solution is incomplete.

11.1.2 Generational Collectors

Renewal-Older-First

Hansen and Clinger [Hansen 02] propose a *renewal-older-first* (ROF) generational collector which divides the heap into two generations, and always collects the older generation. The ROF algorithm groups objects according to their renewal age, which is defined as the time that has passed since the object was last classified as reachable by a collection within its generation, or as its actual age if it has never been considered for collection. After each collection, they assume that the survivors are the youngest objects in the heap.

They implement the ROF algorithm by dividing the heap into steps that contain objects of similar age. The steps are arranged from youngest to oldest. Additional steps are kept as a copy reserve. A policy parameter determines the dividing line

between the younger and older generations. When the older generation is collected, the ROF collector evacuates the live data into the reserve. Then the younger generations become the oldest steps of the ROF heap, and the steps that hold the survivors become the youngest steps. Some of the free steps are used to replenish the reserve, and the remaining free steps become available for allocation.

Hansen and Clinger also provide a 3-generational collector (3ROF) that consists of two generations that are collected by the ROF algorithm, plus a nursery collected as part of every collection. Their hybrid algorithm is younger-first in the sense that it collects the nursery most often, but it is also older-first in the sense that the oldest generation is collected more often than the intermediate generation.

All the collectors that the authors present use a remembered set strategy for pointer-tracking. Their older-first collectors require objects to be stored in several subsets at the same time. That requirement makes card marking and header marking less attractive, since each card or object would need one mark bit for each subset of the remembered set in which it might be stored.

Beltway

In [Blackburn 02a], the authors present the *Beltway* framework. Beltway collectors use increments and belts. An increment is their unit of collection. A belt groups one or more increments into a first-in-first-out queue. Each increment on a belt is collected independently in FIFO order, and each belt is also collected independently. Belts are more general than generations since all objects within a generation must be collected at once, but increments are independently collected and there may be multiple increments on one belt.

Beltway proposes a range of copying collectors that exploit the key insights presented so far. The generality of the Beltway framework enables the implementation of new copying collectors but increases pointer tracking costs. However, some Beltway configurations lack completeness because they fail to collect garbage cycles that span

more than one increment. The authors propose a three-belt generational collector as an alternative approach to their lack of completeness. This collector collects the third belt in its entirety only once it has grown to consume all of the usable memory. This configuration achieves completeness at the expense of incrementality, longer pauses, and space overhead.

Beltway is integrated in Jikes RVM, which produces a high write-barrier activity in the nursery due to the initialization of every object's type pointer. To eliminate this write-barrier overhead, the authors use a single nursery increment and extend the basic Beltway barrier to filter any pointer whose source lies in the nursery. This optimization foregoes older-first behavior within the nursery. Thus, Beltway is not able to benefit from multiple nursery increments. Furthermore, Jikes RVM lays out arrays and scalar objects in different directions in the heap. The beginning of one object cannot be determined from the previous object. Consequently, Beltway collectors cannot use card tables, limiting the number of key ideas in the garbage collection literature that can be implemented and tested.

11.2 Pointer-Tracking Using Card Marking

In this section, we look at other implementations of the card marking mechanism. We non-exhaustively highlight some of the most important advantages and drawbacks of these implementations.

Basic Algorithm

One implementation of this scheme divides the heap into equal-sized logical cards, each covering 2^k bytes of the heap space for a fixed value k. A table indicates whether each card might contain pointers to younger generations. To map an address to an entry in the table, one shifts the address to the right by k bits and uses the result as an index into the table. Whenever a reference is stored into an object, the corresponding card becomes *dirty*. At collection time the dirty cards of all generations not being collected

are scanned [Hosking 92].

Bit Vector

In order to reduce space overhead, Wilson implements the card table as a bit vector. Each generation has its own bit vector, which contains a bit for each logical card. In this scheme, a store check simply marks the bit corresponding to the location being updated. At collection time, the collector scans the bit vector and, whenever it finds a marked bit, examines all the pointers in the corresponding card of the heap [Wilson 89b]. Card marking, as just described, can be fairly slow. Since a bit must be inserted into the bit vector, the corresponding word has to be read from memory, updated and then written back. In addition, bit manipulations usually require several instructions on RISC processors [Hölzle 93].

Figure 11.1 illustrates a generational heap using a card marking mechanism. The nursery's bit vector indicates that the fourth and fifth cards possibly contain pointers referring to the nursery. Both of these cards must be traced at the next nursery collection.

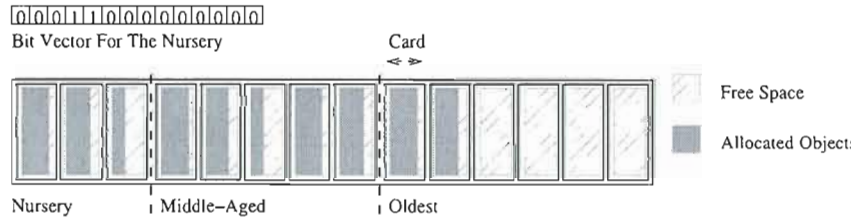

Figure 11.1 Heap Layout for a Generational GC Using Card Marking

Byte Vector

Chambers and Ungar use bytes instead of bits to implement the card table [Chambers 92]. Every card in the heap has one byte associated with it. A card is dirtied simply by storing a special value into its corresponding byte. Although this technique uses eight

times more memory to represent a card than Wilson's scheme, the space overhead is usually still small. By using bytes instead of bits, barrier time is reduced by eliminating word and bit manipulations.

Word Vector

A straightforward per-word bitmap implementing the card table would be prohibitively large in terms of memory for large systems, thus increasing the space overhead. Per-word storage is used by Sobalvarro in [Sobalvarro 88]. It maintains a sparse data structure to avoid large space costs. Manipulating this two-level data structure slows down the system significantly, requiring almost twice as many additional instructions for each store in the heap.

Page Protection

Another version of card marking matches a logical card to a page of virtual memory. It uses the page protection mechanism of the operating system to detect all stores to these pages. All the pages are first protected from storing. When a store occurs to a page, the trap handler dirties the card and unprotects the page. Subsequent stores to this page incur no extra overhead. Page trapping performs poorly in comparison to card marking because pages are generally too large and thus they fail to reach the optimum card size [Shaw 87; Hosking 92].

As mentionned in Chapter 2, all these strategies force the allocator to place objects within the boundaries of a card. We must ensure that the first word of a card is a header word which allows collection. There is unused space at the end of each card. The object size is also bounded by the card size. Systems using these strategies need to scan more space in order to find the roots. These strategies force the collector to trace all the marked cards completely, unlike the technique we propose in Chapter 4.

11.3 Large Object Space

In [Caudill 86], the authors are among the first to work on LOS. They introduced the term large object space in the literature. They propose a garbage collector which represents large array objects with a small header object called a proxy. They use a free list to allocate large objects but do not clearly report how the LOS is collected or if the LOS is even collected at all. It is unclear how large an object should be before it is included in the LOS. Moreover, they do not point out whether data containing pointers could be included in the LOS.

Ungar and Jackson [Ungar 92] simulated a system that represents pointer-less objects in the heap with proxies. Objects larger than 1024 bytes are considered large objects. The authors write that for their study large objects are not longer lived than smaller ones. They do not implement a LOS collector in their simulation. Further, they do not give details of the LOS implementation they added to ParcPlace Smalltalk.

In [Hudson 91], the authors propose a system that divides the heap into many blocks. Objects larger than a block are allocated to the LOS. These blocks are of variable sizes. Consequently, the authors stipulate the heuristic that objects larger than 8 kb should be allocated to the LOS. The latter is collected using a treadmill style collector [Baker 92] and is never compacted. Treadmill collectors use segregated free lists to allocate and reclaim memory in constant time but at the cost of under-utilizing memory, poor locality, and fragmentation [Lim 98].

11.4 Depth-first Pointer Traversal

Depth-first pointer traversal is usually implemented as a recursive algorithm which uses stack space proportional to the longest path in the graph of all reachable objects. In [Schorr 67], the authors present the Deutsch-Schorr-Waite pointer reversal algorithm which use a pointer inversion technique to avoid management of the stack during the traversal of all accessible pointers. The algorithm builds an explicit stack and threads

this stack through the objects encountered while traversing all the pointers. In addition to marking each object, the algorithm needs to record, for each object, the point within the object that is currently being marked. Consequently, two bits for every pointer on the heap are reserved for the garbage collection algorithm, increasing the space overhead. Many works have proved the correctness of the algorithm [Kowaltowski 79; Gerhart 79; Lee 79]. Thomas [Thomas 95] also proposes an algorithm for a recursive depth-first copying garbage collection with no extra stack.

11.5 Conclusion

In this chapter, we described some previous related works, and discussed the advantages and drawbacks of each work.

Chapter XII

FUTURE WORK AND CONCLUSIONS

In this final chapter we discuss future work on the memory manager of SableVM and present our overall conclusions. This chapter is structured as follows. In Section 12.1, we discuss various future research avenues, and in Section 12.2, we present the overall conclusions of this thesis.

12.1 Future Work

12.1.1 Memory Manager Framework in the Field

The first part of our future work consists of releasing the instrumented version of SableVM publicly, gathering feedback from the research community, and establishing new research and development collaborations to share ideas and develop them further.

We hope to attract and develop collaborations between SableVM and other research projects for building a stable, robust, extensible, and efficient memory manager. We believe that our work on building an extensible memory manager research infrastructure can benefit others, and enable them to develop and improve their own methods of research to concentrate their development efforts only on chosen specialized areas.

We also hope to attract graduate students to work specifically on improving parts of the memory manager by implementing existing and innovative techniques. For example, the current heap allocator of SableVM is a naive contiguous allocator. Improving

SableVM's allocator is a suitable project for early graduate courses covering garbage collection and memory management [Gagnon 03b].

12.1.2 Profiling Memory Usage

Developers often make memory leak tracking a low priority because common language runtime takes care of garbage collection. What few developers realize, however, is that both the collection and allocation behaviour of their program affect memory manager performance.

A longer term project is to build a complete memory profiling framework on SableVM. This memory profiling framework could be used as a tool for both researchers and Java developers to better understand memory usage, and more specifically garbage collection in Java programs [Gagnon 03b]. This would be useful for detecting the causes of memory leaks, memory management technique inefficiencies, and poor runtime performances of application.

In general, more careful studies of issues concerning application programs would be a good complement to our memory manager studies and we would hope to pursue this avenue in the future.

12.1.3 Investigating Deeper Garbage Collection Techniques

The primary limitation of our current work is the small set of heaps sampled. Over time we expect to gather heaps from a wider range of applications. This will allow for a better validation and application of our results.

12.1.4 Selecting Garbage Collectors Based on Dynamic Observation

Several runtime systems now offer a choice of multiple garbage collectors. Eventually we can expect our memory manager to select a garbage collector based on dynamic observation of the programs that the machines execute.

12.2 Conclusions

In this thesis, we have introduced our memory manager research framework. One objective of our research was to design and implement a portable and easily modifiable memory manager that could be used for research on various aspects of Java bytecode execution. We also wanted to evaluate the performances achievable by such an extensible framework.

More specifically, in this thesis we introduced the bounded frame marking technique for pointer-tracking, to allow the efficient computation of the root set. Our experiments show that the bounded frame technique reduces both the tracing costs and the space overhead of traditional methods, and even generates competitive performances on many benchmarks.

We also introduced a depth-first object traversal algorithm that exploits the bidirectional object layout, and eliminates the space overhead caused by a recursive stack. We described the implementation of a method to copy objects in a depth-first order without space overhead. Our experiments show that, exploiting the static class-oblivious copying orders (e.g., breadth-first and depth-first), we can tune the memory layout to program traversal and thus improve performance, instead of always using the same static copying order. Our results show that our depth-first traversal algorithm generates better locality on almost all benchmarks, yielding improvements in performance.

We further described the implementation of a method to mark objects without space overhead. This method guarantees that collectors will collect all garbage. Future work should show that our method can improve performance by reducing copying costs.

Finally, we described our large object policy that regroups large objects in memory and makes assumptions about their lifetime. Our experiments show that segregating large objects enables collectors to achieve better performances.

The portability of our memory manager framework was demonstrated by the simplicity of implementing novel collectors. In particular, implemening the breadth-

first semi-space collector took less than 24 hours and less than 400 lines of code.

We hope that these findings and developments shall inspire and stimulate other members of the community in their efforts to develop better and more efficient garbage collectors.

Bibliography

[Adjih 96] Cédric Adjih. Mesure et caractérisation d'applications réparties. Master's thesis, Université Paris Sud, 1996.

[Ali 98] K.A.M. Ali. *A Simple Generational Real-Time Garbage Collection Scheme.* Computing Paradigms and Computational Intelligence (New Generation Computing), vol. 16, no. 2, 1998.

[Appel 89] Andrew W. Appel. *Simple Generational Garbage Collection and Fast Allocation.* Software Practice and Experience, vol. 19, no. 2, pages 171–183, 1989.

[Attardi 98] Giuseppe Attardi, Tito Flagella & Pietro Iglio. *A Customisable Memory Management Framework for C++.* Software Practice and Experience, vol. 28, no. 11, pages 1143–1183, November 1998.

[Azagury 98] Alain Azagury, Elliot K. Kolodner, Erez Petrank & Zvi Yehudai. *Combining Card Marking with Remembered Sets: How to Save Scanning Time.* In Jones [Jones 98], pages 10–19.

[Bacon 01a] David F. Bacon, Clement R. Attanasio, Han B. Lee, V. T. Rajan & Stephen Smith. *Java Without the Coffee Breaks: A Nonintrusive Multiprocessor Garbage Collector.* In Proceedings of SIGPLAN 2001 Conference on Programming Languages Design and Implementation, ACM SIGPLAN Notices, Snowbird, Utah, June 2001. ACM Press.

[Bacon 01b] David F. Bacon & V.T. Rajan. *Concurrent Cycle Collection in Reference Counted Systems.* In Jørgen Lindskov Knudsen, editeur, Proceedings of 15th European Conference on Object-Oriented Programming, ECOOP 2001, volume 2072 of *Lecture Notes in Computer Science*, Budapest, June 2001. Springer-Verlag.

[Baker 78] Henry G. Baker. *List Processing in Real-Time on a Serial Computer.* Communications of the ACM, vol. 21, no. 4, pages 280–94, 1978. Also AI Laboratory Working Paper 139, 1977.

[Baker 92] Henry G. Baker. *The Treadmill, Real-time Garbage Collection without Motion Sickness.* ACM SIGPLAN Notices, vol. 27, no. 3, pages 66–70, March 1992.

[Baker 93] Henry G. Baker. *'Infant Mortality' and Generational Garbage Collection.* ACM SIGPLAN Notices, vol. 28, no. 4, April 1993.

[Barabash 03] Katherine Barabash, Yoav Ossia & Erez Petrank. *Mostly Concurrent Garbage Collection Revisited.* In OOPSLA'03 ACM Conference on Object-Oriented Systems, Languages and Applications [OOPSLA 03].

[Barrett 93] David A. Barrett & Benjamin G. Zorn. *Using Lifetime Predictors to Improve Memory Allocation Performance.* In Proceedings of SIGPLAN'93 Conference on Programming Languages Design and Implementation [PLDI 93], pages 187–196.

[Bartlett 88] Joel F. Bartlett. *Compacting Garbage Collection with Ambiguous Roots.* Rapport technique 88/2, DEC Western Research Laboratory, Palo Alto, CA, February 1988. Also in Lisp Pointers 1, 6 (April–June 1988), 2–12.

[Bekkers 92] Yves Bekkers & Jacques Cohen, editeurs. Proceedings of international workshop on memory management, volume 637 of *Lecture Notes in Computer Science*, St Malo, France, 16–18 September 1992. Springer-Verlag.

[Ben-Yitzhak 02] Ori Ben-Yitzhak, Irit Goft, Elliot Kolodner, Kean Kuiper & Victor Leikehman. *An Algorithm for Parallel Incremental Compaction.* In Detlefs [Detlefs 02], pages 100–105.

[Bishop 75] Peter B. Bishop. *Garbage collection in a Very Large Address Space.* Working paper 111, AI Laboratory, MIT, Cambridge, MA, September 1975.

[Blackburn 02a] Stephen M. Blackburn, Richard Jones, Kathryn S. McKinley & J. Eliot B. Moss. *Beltway: Getting Around Garbage Collection Gridlock.* In Proceedings of SIGPLAN 2002 Conference on Programming Languages Design and Implementation, ACM SIGPLAN Notices, pages 153–164, Berlin, June 2002. ACM Press.

[Blackburn 02b] Stephen M. Blackburn & Kathryn S. McKinley. *In or Out? Putting Write Barriers in Their Place.* In Detlefs [Detlefs 02], pages 175–184.

[Blackburn 03a] Stephen M. Blackburn, Perry Cheng & Kathryn S. McKinley. *A Garbage Collection Design and Bakeoff in JMTk: An Extensible Java Memory Management Toolkit.* Rapport technique TR–CS–03–02, Australian National University, February 2003.

[Blackburn 03b] Stephen M. Blackburn & Kathryn S. McKinley. *Ulterior Reference Counting: Fast Garbage Collection without a Long Wait.* In OOPSLA'03 ACM Conference on Object-Oriented Systems, Languages and Applications [OOPSLA 03].

[Blackburn 04a] Stephen M. Blackburn, Perry Cheng & 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.

[Blackburn 04b] Stephen M. Blackburn, Perry Cheng & Kathryn S. McKinley. *Oil and Water? High Performance Garbage Collection in Java with MMTk.* In ICSE 2004, 26th International Conference on Software Engineering, Edinburgh, May 2004.

[Blackburn 07] Stephen M. Blackburn, Matthew Hertz, Kathryn S. Mckinley, J. Eliot B. Moss & Ting Yang. *Profile-based pretenuring.* ACM Trans. Program. Lang. Syst., vol. 29, no. 1, page 2, 2007.

[Boehm 00] Hans-Juergen Boehm. *Reducing Garbage Collector Cache Misses.* In Tony Hosking, editeur, ISMM 2000 Proceedings of the Second International Symposium on Memory Management, volume 36(1) of *ACM SIGPLAN Notices*, Minneapolis, MN, October 2000. ACM Press.

[Boehm 01] Hans Boehm. *A garbage collector for C and C++.* World Wide Web, `http://www.hpl.hp.com/personal/Hans_Boehm/gc/`, 2001.

[Caudill 86] Patrick J. Caudill & Allen Wirfs-Brock. *A Third-Generation Smalltalk-80 Implementation.* In Norman Meyrowitz, editeur, OOPSLA'86 ACM Conference on Object-Oriented Systems, Languages and Applications, volume 21(11) of *ACM SIGPLAN Notices*, pages 119–130. ACM Press, October 1986.

[Chambers 92] Craig Chambers. *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for an Objected-Oriented Programming Language.* PhD thesis, Stanford University, March 1992.

[Cheadle 04] Andrew M. Cheadle, Anthony J. Field, Marlow Simon, Simon L. Peyton-Jones & Lyndon While. *Exploring the Barrier to Entry — Incremental Generational Garbage Collection for Haskell.* In Diwan [Diwan 04].

[Cheney 70] C. J. Cheney. *A Non-Recursive List Compacting Algorithm.* Communications of the ACM, vol. 13, no. 11, pages 677–8, November 1970.

[Chilimbi 98] Trishul M. Chilimbi & James R. Larus. *Using Generational Garbage Collection To Implement Cache-Conscious Data Placement.* In Jones [Jones 98], pages 37–48.

[ching Ju 01] Roy Dz ching Ju, Alvin R. Lebeck & Chris Wilkerson. *Locality vs. criticality.*

In Srikanth T. Srinivasan, editeur, ISCA '01: Proceedings of the 28th annual international symposium on Computer architecture, pages 132–143, New York, NY, USA, 2001. ACM Press.

[Collins 60] George E. Collins. *A Method for Overlapping and Erasure of Lists.* Communications of the ACM, vol. 3, no. 12, pages 655–657, December 1960.

[Colnet 98] Dominique Colnet, Philippe Coucaud & Olivier Zendra. *Compiler Support to Customize the Mark and Sweep Algorithm.* In Jones [Jones 98], pages 154–165.

[Corporation 98] Standard Performance Evaluation Corporation. *SPEC JVM98 Benchmarks.* World Wide Web, `http://www.spec.org/osg/jvm98/`, 1998.

[Courts 88] Robert Courts. *Improving Locality Of Reference In A Garbage-Collecting Memory Management-System.* Communications of the ACM, vol. 31, no. 9, pages 1128–1138, 1988.

[Detlefs 02] David Detlefs, editeur. ISMM'02 proceedings of the third international symposium on memory management, ACM SIGPLAN Notices, Berlin, June 2002. ACM Press.

[DeTreville 90] John DeTreville. *Experience with Garbage Collection for Modula-2+ in the Topaz Environment.* In Eric Jul & Niels-Christian Juul, editeurs, OOPSLA/ECOOP '90 Workshop on Garbage Collection in Object-Oriented Systems, Ottawa, October 1990.

[Deutsch 76] L. Peter Deutsch & Daniel G. Bobrow. *An Efficient Incremental Automatic Garbage Collector.* Communications of the ACM, vol. 19, no. 9, pages 522–526, September 1976.

[Diwan 04] Amer Diwan, editeur. ISMM'04 proceedings of the third international symposium on memory management, ACM SIGPLAN Notices, Vancouver, October 2004. ACM Press.

[Fenichel 69] Robert R. Fenichel & Jerome C. Yochelson. *A Lisp Garbage Collector for Virtual Memory Computer Systems.* Communications of the ACM, vol. 12, no. 11, pages 611–612, November 1969.

[Gagnon 02a] Etienne Gagnon. *A Portable Research Framework for the Execution of Java Bytecode.* PhD thesis, McGill University, Montreal, Quebec, December 2002.

[Gagnon 02b] Etienne Gagnon. Sablecc, an object-oriented compiler framework. Master's the-

sis, McGill University, Montreal, Quebec, December 2002.

[Gagnon 03a] Etienne Gagnon. *SableVM: A Research Framework for the Efficient Execution of Java Bytecode.* World Wide Web, `http://www.sable.mcgill.ca/publications/techreports/#report2000-3`, 2003.

[Gagnon 03b] Etienne Gagnon. *SableVM: a robust, clean, easy to maintain and extend, extremely portable, efficient, and specification-compliant Java virtual machine.* World Wide Web, `http://sablevm.org/`, 2003.

[Gerhart 79] S. L. Gerhart. *A Derivation Oriented Proof of Schorr–Waite Marking Algorithm.* Lecture Notes in Computer Science, vol. 69, pages 472–492, 1979.

[Group 06] Sable Research Group. *Soot: a Java Optimization Framework.* World Wide Web, `http://www.sable.mcgill.ca/soot/`, 2006.

[Grunwald 93] Dirk Grunwald, Benjamin Zorn & Robert Henderson. *Improving the Cache Locality of Memory Allocation.* In Proceedings of SIGPLAN'93 Conference on Programming Languages Design and Implementation [PLDI 93], pages 177–186.

[Guyer 04] Samuel Guyer & Kathryn McKinley. *Finding Your Cronies: Static Analysis for Dynamic Object Colocation.* In OOPSLA'04 ACM Conference on Object-Oriented Systems, Languages and Applications [OOPSLA 04].

[Hansen 00] Lars Thomas Hansen. *Older-first Garbage Collection in Practice.* PhD thesis, North-eastern University, November 2000.

[Hansen 02] Lars Thomas Hansen & William D. Clinger. *An Experimental Study of Renewal-Older-First Garbage Collection.* In Proceedings of the 2002 ACM SIGPLAN International Conference on Functional Programming (ICFP02), volume 37(9) of *ACM SIGPLAN Notices*, pages 247–258, Pittsburgh, PA, 2002. ACM Press.

[Hanson 90] David R. Hanson. *Fast Allocation and Deallocation of Memory Based on Object Lifetimes.* Software Practice and Experience, vol. 20, no. 1, pages 5–12, January 1990.

[Hayes 91] Barry Hayes. *Using Key Object Opportunism to Collect Old Objects.* In Andreas Paepcke, editeur, OOPSLA'91 ACM Conference on Object-Oriented Systems, Languages and Applications, volume 26(11) of *ACM SIGPLAN Notices*, pages 33–46, Phoenix, Arizona, October 1991. ACM Press.

[Hayes 93] Barry Hayes. *Key Objects in Garbage Collection.* PhD thesis, Stanford University,

March 1993.

[Hertz 05a] Matthew Hertz & Emery Berger. *Quantifying the Performance of Garbage Collection vs. Explicit Memory Management.* In OOPSLA'05 ACM Conference on Object-Oriented Systems, Languages and Applications, ACM SIGPLAN Notices, San Diego, CA, October 2005. ACM Press.

[Hertz 05b] Matthew Hertz, Yi Feng & Emery D. Berger. *Garbage Collection Without Paging.* In Proceedings of SIGPLAN 2005 Conference on Programming Languages Design and Implementation, ACM SIGPLAN Notices, Chicago, IL, June 2005. ACM Press.

[Hicks 97] Michael W. Hicks, Jonathan T. Moore & Scott M. Nettles. *The Measured Cost of Copying Garbage Collection Mechanisms.* In Proceedings of Second International Conference on Functional Programming, pages 292–305, Amsterdam, June 1997. ACM Press.

[Hicks 98] Michael Hicks, Luke Hornof, Jonathan T. Moore & Scott Nettles. *A Study of Large Object Spaces.* In Jones [Jones 98], pages 138–145.

[Hirzel 03] Martin Hirzel, Amer Diwan & Matthew Hertz. *Connectivity-based Garbage Collection.* In OOPSLA'03 ACM Conference on Object-Oriented Systems, Languages and Applications [OOPSLA 03].

[Hölzle 93] Urs Hölzle. *A Fast Write Barrier for Generational Garbage Collectors.* In Moss et al. [Moss 93].

[Hosking 92] Anthony L. Hosking, J. Eliot B. Moss & Darko Stefanović. *A Comparative Performance Evaluation of Write Barrier Implementations.* In Andreas Paepcke, editeur, OOPSLA'92 ACM Conference on Object-Oriented Systems, Languages and Applications, volume 27(10) of *ACM SIGPLAN Notices*, pages 92–109, Vancouver, British Columbia, October 1992. ACM Press.

[Hosking 93] Antony L. Hosking & Richard L. Hudson. *Remembered Sets Can Also Play Cards.* In Moss et al. [Moss 93].

[Huang 04] Xianlong Huang, Stephen M. Blackburn, Kathryn S. McKinley, J. Eliot B. Moss, Z. Wang & Perry Cheng. *The Garbage Collection Advantage: Improving Program Locality.* In OOPSLA'04 ACM Conference on Object-Oriented Systems, Languages and Applications [OOPSLA 04].

[Hudson 91] Richard L. Hudson, J. Eliot B. Moss, Amer Diwan & Christopher F. Weight.

A Language-Independent Garbage Collector Toolkit. Rapport technique COINS 91-47, University of Massachusetts at Amherst, Department of Computer and Information Science, September 1991.

[Hudson 92] Richard L. Hudson & J. Eliot B. Moss. *Incremental Garbage Collection for Mature Objects.* In Bekkers & Cohen [Bekkers 92].

[Hudson 97] Richard L. Hudson, Ron Morrison, J. Eliot B. Moss & David S. Munro. *Garbage Collecting the World: One Car at a Time.* In OOPSLA'97 ACM Conference on Object-Oriented Systems, Languages and Applications — Twelth Annual Conference, volume 32(10) of *ACM SIGPLAN Notices*, Atlanta, GA, October 1997. ACM Press.

[IBM 03] IBM. *Tuning Garbage Collection with the 5.0 Java[tm] Virtual Machine.* World Wide Web, `http://java.sun.com/docs/hotspot/gc5.0/gc_tuning_5.html`, 2003.

[Inoue 03] H. Inoue, Darko Stefanović & S. Forrest. *Object Lifetime Prediction in Java.* Rapport technique TR–CS–2003–28, University of New Mexico, May 2003.

[Inoue 06] Hajime Inoue, Darko Stefanovic & Stephanie Forrest. *On the Prediction of Java Object Lifetimes.* IEEE Transactions on Computers, vol. 55, no. 7, pages 880–892, 2006.

[Jones 96] Richard E. Jones. Garbage collection: Algorithms for automatic dynamic memory management. Wiley, Chichester, July 1996. With a chapter on Distributed Garbage Collection by R. Lins.

[Jones 98] Richard Jones, editeur. ISMM'98 proceedings of the first international symposium on memory management, volume 34(3) of *ACM SIGPLAN Notices*, Vancouver, October 1998. ACM Press.

[Kowaltowski 79] T. Kowaltowski. *Data Structures and Correctness of Programs.* Journal of the ACM, vol. 26, no. 2, pages 283–301, April 1979.

[Lee 79] S. Lee, W. P. De Roever & S. Gerhart. *The Evolution of List Copying Algorithms.* In 6th ACM Symposium on Principles of Programming Languages, pages 53–56, San Antonio, Texas, January 1979. ACM Press.

[Lieberman 81] Henry Lieberman & Carl E. Hewitt. *A Real-Time Garbage Collector Based on the Lifetimes of Objects.* AI Memo 569a, MIT, April 1981.

[Lieberman 83] Henry Lieberman & Carl E. Hewitt. *A Real-Time Garbage Collector Based on the Lifetimes of Objects.* Communications of the ACM, vol. 26(6), pages 419–429, 1983. Also report TM–184, Laboratory for Computer Science, MIT, Cambridge, MA, July 1980 and AI Lab Memo 569, 1981.

[Lim 98] Tian F. Lim, Przemyslaw Pardyak & Brian N. Bershad. *A Memory-Efficient Real-Time Non-Copying Garbage Collector.* In Jones [Jones 98], pages 118–129.

[Lin 92] Sheng-Lien Lin. *Performance Evaluation of a Generation Scavenging Algorithm,* 1992.

[Lins 92] Rafael D. Lins. *Cyclic Reference Counting with Lazy Mark-Scan.* Information Processing Letters, vol. 44, no. 4, pages 215–220, 1992. Also Computing Laboratory Technical Report 75, University of Kent, July 1990.

[Martinez 90] A. D. Martinez, R. Wachenchauzer & Rafael D. Lins. *Cyclic Reference Counting with Local Mark-Scan.* Information Processing Letters, vol. 34, pages 31–35, 1990.

[McCarthy 60] John McCarthy. *Recursive Functions of Symbolic Expressions and their Computation by Machine.* Communications of the ACM, vol. 3, pages 184–195, 1960.

[Moon 84] David A. Moon. *Garbage Collection in a Large LISP System.* In Guy L. Steele, editeur, Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming, pages 235–245, Austin, TX, August 1984. ACM Press.

[Moss 93] Eliot Moss, Paul R. Wilson & Benjamin Zorn, editeurs. OOPSLA/ECOOP '93 workshop on garbage collection in object-oriented systems, October 1993.

[Moss 96] J. Eliot B. Moss, David S. Munro & Richard L. Hudson. *PMOS: A Complete and Coarse-grained Incremental Garbage Collector for Persistent Object Stores.* In Proceedings of the Seventh International Workshop on Persistent Object Systems, pages 140–150. Morgan Kaufmann, June 1996.

[Munro 99] David Munro, Alfred Brown, Ron Morrison & J. Eliot B. Moss. *Incremental Garbage Collection of a Persistent Object Store using PMOS.* In Ron Morrison, Mick Jordan & Malcolm Atkinson, editeurs, Advances in Persistent Object Systems, pages 78–91. Morgan Kaufman, 1999.

[Myers 95] Andrew C. Myers. *Bidirectional object layout for separate compilation.* In Proceedings of the ACM OOPSLA symposium on Object-Oriented Programming Systems, Languages, and Applications, pages 124–139, October 1995.

[OOPSLA 03] OOPSLA'03 ACM conference on object-oriented systems, languages and applications, ACM SIGPLAN Notices, Anaheim, CA, November 2003. ACM Press.

[OOPSLA 04] OOPSLA'04 ACM conference on object-oriented systems, languages and applications, ACM SIGPLAN Notices, Vancouver, October 2004. ACM Press.

[Ossia 04] Yoav Ossia, Ori Ben-Yitzhak & Marc Segal. *Mostly Concurrent Compaction for Mark-Sweep GC.* In Diwan [Diwan 04].

[PLDI 93] Proceedings of SIGPLAN'93 conference on programming languages design and implementation, volume 28(6) of *ACM SIGPLAN Notices*, Albuquerque, NM, June 1993. ACM Press.

[Pugh 90] William Pugh & Grant Weddell. *Two-directional record layout for multiple inheritance.* In ACM SIGPLAN Notices, pages 85–91, June 1990.

[Reppy 93] John H. Reppy. *A High-Performance Garbage Collector for Standard ML.* Technical memorandum, AT&T Bell Laboratories, Murray Hill, NJ, December 1993.

[Röjemo 95] Niklas Röjemo. *Garbage Collection, and Memory Efficiency, in Lazy Functional Languages.* PhD thesis, Chalmers University of Technology, Goteborg, Sweden, 1995.

[Rovner 85] Paul Rovner. *On Adding Garbage Collection and Runtime Types to a Strongly-Typed, Statically-Checked, Concurrent Language.* Technical Report CSL–84–7, Xerox PARC, Palo Alto, CA, July 1985.

[Sachindran 03] Narendran Sachindran & Eliot Moss. *MarkCopy: Fast Copying GC with Less Space Overhead.* In OOPSLA'03 ACM Conference on Object-Oriented Systems, Languages and Applications [OOPSLA 03].

[Sachindran 04] Narendran Sachindran, J. Eliot B. Moss & Emery D. Berger. MC^2*: High-Performance Garbage Collection for Memory-Constrained Environments.* In OOPSLA'04 ACM Conference on Object-Oriented Systems, Languages and Applications [OOPSLA 04].

[Schkolnick 77] M. Schkolnick. *A clustering algorithm for hierarchical structures.* ACM Trans. Database Syst., vol. 2, no. 1, pages 27–44, 1977.

[Schorr 67] H. Schorr & W. Waite. *An Efficient Machine Independent Procedure for Garbage Collection in Various List Structures.* Communications of the ACM, vol. 10, no. 8, pages 501–506, August 1967.

[Seligmann 95] Jacob Seligmann & Steffen Grarup. *Incremental Mature Garbage Collection using the Train Algorithm.* In O. Nierstras, editeur, Proceedings of 1995 European Conference on Object-Oriented Programming, Lecture Notes in Computer Science, pages 235–252, University of Aarhus, August 1995. Springer-Verlag.

[Shaw 87] Robert A. Shaw. *Improving Garbage Collector Performance in Virtual Memory.* Rapport technique CSL–TR–87–323, Stanford University, March 1987. Also Hewlett-Packard Laboratories report STL–TM–87–05, Palo Alto, 1987.

[Shuf 02] Yefim Shuf, Manish Gupta, Hubertus Franke, Andrew Appel & Jaswinder Pal Singh. *Creating and Preserving Locality of Java Applications at Allocation and Garbage Collection Times.* In OOPSLA'02 ACM Conference on Object-Oriented Systems, Languages and Applications, ACM SIGPLAN Notices, Seattle, WA, November 2002. ACM Press.

[Smith 98] Frederick Smith & Greg Morrisett. *Comparing Mostly-Copying and Mark-Sweep Conservative Collection.* In Jones [Jones 98], pages 68–78.

[Sobalvarro 88] Patrick Sobalvarro. *A Lifetime-Based Garbage Collector for Lisp Systems on General-Purpose Computers.* Rapport technique AITR-1417, MIT AI Lab, February 1988. Bachelor of Science thesis.

[Soman 04] Sunil Soman, Chandra Krintz & David Bacon. *Dynamic Selection of Application-Specific Garbage Collectors.* Rapport technique 2004–09, UCSB, January 2004.

[Stamos 84] James W. Stamos. *Static Grouping of Small Objects to Enhance Performance of a Paged Virtual Memory.* ACM Transactions on Computer Systems, vol. 2, no. 3, pages 155–180, May 1984.

[Stefanović 94] Darko Stefanović & J. Eliot B. Moss. *Characterisation of Object Behaviour in Standard ML of New Jersey.* In Conference Record of the 1994 ACM Symposium on Lisp and Functional Programming, pages 43–54. ACM Press, June 1994.

[Stefanović 99a] Darko Stefanović. *Properties of Age-Based Automatic Memory Reclamation Algorithms.* PhD thesis, University of Massachusetts, 1999.

[Stefanović 99b] Darko Stefanović, Kathryn S. McKinley & J. Eliot B. Moss. *Age-Based Garbage Collection.* In OOPSLA'99 ACM Conference on Object-Oriented Systems, Languages and Applications, volume 34(10) of *ACM SIGPLAN Notices*, pages 370–381, Denver, CO, October 1999. ACM Press.

[Stefanović 99c] Darko Stefanović, J. Eliot B. Moss & Kathryn S. McKinley. *Age-Based Garbage*

Collection. Rapport technique, University of Massachussets, April 1999. preliminary version of a paper to appear in OOPSLA'99.

[Stefanovic 02] Darko Stefanovic, Matthew Hertz, Stephen Blackburn, Kathryn McKinley & J. Eliot Moss. *Older-First Garbage Collection in Practice: Evaluation in a Java Virtual Machine.* In ACM SIGPLAN Workshop on Memory System Performance (MSP 2002), Berlin, June 2002.

[Tarditi 93] David Tarditi & Amer Diwan. *The Full Cost of a Generational Copying Garbage Collection Implementation.* In Moss et al. [Moss 93].

[Thomas 95] Stephen P. Thomas. Having your cake and eating it: Recursive depth-first copying garbage collection with no extra stack. Personal communication, May 1995.

[Ungar 84] David M. Ungar. *Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm.* ACM SIGPLAN Notices, vol. 19, no. 5, pages 157–167, April 1984. Also published as ACM Software Engineering Notes 9, 3 (May 1984) — Proceedings of the ACM/SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, 157–167, April 1984.

[Ungar 88] David M. Ungar & Frank Jackson. *Tenuring Policies for Generation-Based Storage Reclamation.* ACM SIGPLAN Notices, vol. 23, no. 11, pages 1–17, 1988.

[Ungar 92] David M. Ungar & Frank Jackson. *An Adaptive Tenuring Policy for Generation Scavengers.* ACM Transactions on Programming Languages and Systems, vol. 14, no. 1, pages 1–27, 1992.

[Wadler 87] Philip L. Wadler. *Fixing Some Space Leaks with a Garbage Collector.* Software Practice and Experience, vol. 17, no. 9, pages 595–609, September 1987.

[Weizenbaum 63] J. Weizenbaum. *Symmetric List Processor.* Communications of the ACM, vol. 6, no. 9, pages 524–544, September 1963.

[Weizenbaum 69] J. Weizenbaum. *Recovery of Reentrant List Structures in SLIP.* Communications of the ACM, vol. 12, no. 7, pages 370–372, July 1969.

[Wilson 89a] Paul R. Wilson & Thomas G. Moher. *A Card-Marking Scheme for Controlling Intergenerational References in Generation-Based Garbage Collection on Stock Hardware.* ACM SIGPLAN Notices, vol. 24, no. 5, pages 87–92, 1989.

[Wilson 89b] Paul R. Wilson & Thomas G. Moher. *Design of the Opportunistic Garbage Col-*

lector. In OOPSLA'89 ACM Conference on Object-Oriented Systems, Languages and Applications, volume 24(10) of *ACM SIGPLAN Notices*, pages 23–35, New Orleans, LA, October 1989. ACM Press.

[Wilson 91] Paul R. Wilson, Michael S. Lam & Thomas G. Moher. *Effective Static-Graph Reorganization to Improve Locality in Garbage Collected Systems.* ACM SIGPLAN Notices, vol. 26, no. 6, pages 177–191, 1991.

[Wilson 92] Paul R. Wilson. *Uniprocessor Garbage Collection Techniques.* In Bekkers & Cohen [Bekkers 92].

[Zorn 90] Benjamin Zorn. *Comparing Mark-and-Sweep and Stop-and-Copy Garbage Collection.* In Conference Record of the 1990 ACM Symposium on Lisp and Functional Programming, Nice, France, June 1990. ACM Press.