MEMORY RECLAMATION IN EMERALD - AN OBJECT-ORIENTED
PROGRAMMING LANGUAGE

By
Xiaomei Han
M. Sc. Chinese Academic Institute of Software, 1989

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
M. SCIENCE

in
THE FACULTY OF GRADUATE STUDIES
DEPARTMENT OF COMPUTER SCIENCE

We accept this thesis as conforming
to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA
June 1994
© Xiaomei Han, 1994
In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science
The University of British Columbia
Vancouver, Canada

Date July 11, 1994
Abstract

Storage management is an important part of a programming system. There are two basic storage management techniques: explicit storage management and automatic storage management. This thesis investigates automatic storage management, garbage collection techniques, in the object-oriented programming language Emerald. In general, garbage collection algorithms fall into four categories, namely, reference counting, mark-and-sweep, copying, and generational copying based algorithms. Since these garbage collection techniques have been proposed, there has been a large amount of work done. However, the work has mainly focused on garbage collection of functional programming languages. How well will these garbage collection techniques perform in an object-oriented programming paradigm in which the behavior is rather different from that of functional programming languages? This thesis mainly aims at studying the most promising garbage collection technique — the generational garbage collection algorithm, and gathering and measuring the parameters that affect the generational garbage collection. We use both simulation and measurement of an actual implementation to compare the performance of several alternating garbage collection techniques in Emerald.
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>vii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>viii</td>
</tr>
<tr>
<td>Acknowledgement</td>
<td>x</td>
</tr>
<tr>
<td><strong>1 Introduction</strong></td>
<td>1</td>
</tr>
<tr>
<td>1.1 Storage Management</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Garbage Collection Techniques</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Garbage Collection Algorithms</td>
<td>3</td>
</tr>
<tr>
<td>1.4 Problems</td>
<td>4</td>
</tr>
<tr>
<td>1.5 Thesis Organization</td>
<td>4</td>
</tr>
<tr>
<td>1.5.1 Simulations</td>
<td>5</td>
</tr>
<tr>
<td>1.5.2 Performance Reports</td>
<td>6</td>
</tr>
<tr>
<td>1.6 Thesis Contents</td>
<td>7</td>
</tr>
<tr>
<td><strong>2 Overview of Garbage Collection Algorithms</strong></td>
<td>8</td>
</tr>
<tr>
<td>2.1 Reference Counting</td>
<td>8</td>
</tr>
<tr>
<td>2.1.1 Reference Counting Algorithm</td>
<td>8</td>
</tr>
<tr>
<td>2.1.2 Variants of Reference Counting Algorithm</td>
<td>10</td>
</tr>
<tr>
<td>2.1.3 Analysis of Reference Counting Algorithm</td>
<td>10</td>
</tr>
<tr>
<td>2.2 Mark-and-Sweep</td>
<td>12</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
</tr>
<tr>
<td>2.2.1</td>
<td>Mark-and-Sweep Algorithm</td>
</tr>
<tr>
<td>2.2.2</td>
<td>Analysis of the Mark-and-Sweep Algorithm</td>
</tr>
<tr>
<td>2.3</td>
<td>Copying-Based Garbage Collection Algorithms</td>
</tr>
<tr>
<td>2.3.1</td>
<td>Algorithm Description</td>
</tr>
<tr>
<td>2.3.2</td>
<td>Analysis of Copying Based Collection Algorithm</td>
</tr>
<tr>
<td>2.4</td>
<td>Generational Garbage Collection Algorithms</td>
</tr>
<tr>
<td>2.4.1</td>
<td>Algorithm Description</td>
</tr>
<tr>
<td>2.4.2</td>
<td>Analysis of Generational Collection Algorithm</td>
</tr>
<tr>
<td>2.5</td>
<td>Incremental Garbage Collection Algorithms</td>
</tr>
<tr>
<td>2.6</td>
<td>Distributed Garbage Collection</td>
</tr>
<tr>
<td>3</td>
<td>Simulations and Cost Metric</td>
</tr>
<tr>
<td>3.1</td>
<td>Evaluation Techniques</td>
</tr>
<tr>
<td>3.1.1</td>
<td>Implementation Reports</td>
</tr>
<tr>
<td>3.1.2</td>
<td>Analytic Evaluations</td>
</tr>
<tr>
<td>3.1.3</td>
<td>Simulations</td>
</tr>
<tr>
<td>3.2</td>
<td>Emerald Experiments</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Brief Introduction of the Emerald Language</td>
</tr>
<tr>
<td>3.2.2</td>
<td>The Existing Emerald Garbage Collector</td>
</tr>
<tr>
<td>3.2.3</td>
<td>Experiment Program</td>
</tr>
<tr>
<td>3.2.4</td>
<td>Our Experimental Model of Generational Garbage Collection</td>
</tr>
<tr>
<td>3.2.5</td>
<td>Cost of Generational Garbage Collection</td>
</tr>
<tr>
<td>3.2.6</td>
<td>Generational Garbage Collection Parameters</td>
</tr>
<tr>
<td>3.3</td>
<td>Cost Metric</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Total Cost Metric</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Cost Metric for Copying Live Objects</td>
</tr>
</tbody>
</table>
List of Tables

3.1 Collection Number .................................................. 33
3.2 Store Operations and Remember Set Ratios in LISP ............ 39
List of Figures

2.1 Reference Counting .......................................................... 9
2.2 Reference Counting with Unreclaimable Cycle ....................... 11
2.3 A Simple Semispace Garbage Collector Before a Garbage Collection ... 15
2.4 A Simple Semispace Garbage Collector After a Garbage Collection .... 16
2.5 A Generational Collector Before a Garbage Collection ................. 19
2.6 A Generational Collector After a Garbage Collection ................ 20

3.7 Average Number of Live Objects as a Function of Threshold .......... 38
3.8 The Ratio of Remember Set over Total Memory Operations as a Function of Threshold .......................................................... 39
3.9 Copying Overhead of the Collection in Terms of Objects Allocated .... 40
3.10 Percentage of Live Objects Between Two Successive Collections ....... 41

4.11 Emerald Object Structure ................................................... 44
4.12 State of the New Generation Before a Garbage Collection ............ 46
4.13 State of the New Generation After a Garbage Collection .............. 47

5.14 Average Live Objects of the Compiler ................................... 56
5.15 CPU Overhead of Mark-and-Sweep Collector ............................ 58
5.16 CPU Overhead of Simple Copying Collector ............................. 60
5.17 CPU Overhead of Copying Compacting Collector ....................... 62
5.18 CPU Overhead of the Generational Collector ........................... 63
5.19 CPU Overhead of the Generational Collector (4MBytes Heap Space) ... 64
5.20 CPU Overhead of the Generational Collector (5MBytes Heap Space) . . 65
5.21 CPU Overhead of the Generational Collector (6MBytes Heap Space) . . 66
5.22 CPU Overhead of All the Collector with 4M Heap Space . . . . . . . 66
5.23 CPU Overhead of All the Collectors with 5M Heap Space . . . . . . . 67
5.24 CPU Overhead of All the Collectors with 6M Heap Space . . . . . . . 68
I would like to thank my supervisor Dr. Norm Hutchinson for all the guidance and support he has given me to make this thesis happen. I would also like to thank Dr. Gerald Neufeld for taking time to read this thesis. My thank also goes to Scott Hazelhurst for helping me revise this thesis. I would like to thank the graduate students, especially Helene Wong and Catherine Leung, and the faculty and staff of the University of British Columbia Department of Computer Science for their making my study in the department a wonderful experience. I want to thank my friend Ms. Yoko Ando for all the emotional support she has been offering during my stay in Vancouver.

I would like to thank my parents for the love and support they have given me.
Chapter 1

Introduction

1.1 Storage Management

Storage management is an important part of a programming system. A storage management system must ensure an ample supply of memory for new objects that are created by user programs. There are two basic storage management techniques: explicit storage management and automatic storage management.

Using explicit storage management, the programmer must explicitly reclaim the memory that is no longer in use. Explicit storage management suffers from two insidious disadvantages: memory leaks and early releases. A memory leak occurs when objects are allocated but then not released when they are no longer necessary. Eventually, such a leak can cause a long running program to consume the entire available space and therefore fail. The other drawback is that if a programmer frees a memory that is still reachable from other program objects, the system will behave unpredictably.

Due to these problems caused by explicit storage management, languages, such as LISP [YF69], Smalltalk [Ung84], and Emerald [Hut87] include automatic storage reclamation. In contrast to explicit storage management, automatic storage management frees the programmer from the burden of explicitly reclaiming the memory that is no longer in use. Garbage collection, which is included as part of the programming language implementation, takes care of the reclaiming of memory without programmer involvement. The garbage collector’s function is to trace all the objects that are still in use and make
space that is occupied by garbage objects available. An object is considered garbage if it is no longer reachable by the running program via any possible sequence of pointer traversals.

There are several advantages of using garbage collection techniques. Firstly, garbage collection improves modular programming by avoiding the introduction of unnecessary inter-module dependencies. For instance, an object can be freed without knowing if other modules are interested in it. Secondly, it avoids system failures caused by failing to explicitly reclaim free space.

1.2 Garbage Collection Techniques

Since garbage collection techniques were first proposed, they have drawn much attention. A lot of work has been done in this field, mostly in the areas of garbage collection algorithm design, correctness proof of the algorithms, and performance analysis of the algorithms.

As to the algorithm design, the garbage collection algorithms fall into four basic categories: reference counting, mark-and-sweep, copying based algorithms, and generational based algorithms.

Correctness proof of the algorithms is mainly aimed at incremental garbage collection in which a user program interleaves with the garbage collection program. A garbage collection algorithm has to be proved both complete (all the garbage in the system is collected) and sound (only the garbage is collected).

Performance analysis is another important research topic in garbage collection. There are three approaches which can be used for performance analysis of garbage collection algorithms, namely, simulations, implementation reports, and analytic evaluations.
1.3 Garbage Collection Algorithms

There are many garbage collection algorithms that have been proposed and studied. They can be broadly categorized as either single generation garbage collection or multiple-generation collection in a non-incremental garbage collection context, in which there is not interference between the user program (mutator) and garbage collection program (collector).

It has been proved that single generation garbage collection can be made very efficient if sufficient heap space is available. However, within a realistic amount of memory, the efficiency of a single space collection is limited by the fact that the system must copy all live objects from one region to another at each collection. The time that is taken to finish a collection is proportional to the number of live objects at the time the collection is done. For time critical applications, such as real-time applications, the resulting long collection times can be disruptive.

*Generational garbage collection*, a promising garbage collection algorithm, was proposed to address the problem with copying large numbers of live objects. It attempts to make garbage collection more efficient by taking advantage of observed object lifetime distributions. For many programming languages, it turns out that most objects in a system live for a very short time, and the objects that have lived for a long time tend to be permanent. Therefore, significant performance gain can be made by avoiding the frequent copying of objects that live for a long time. Generational garbage collection achieves this by dividing the space into several regions, with each region garbage collected at a different frequency. The objects that have experienced a certain number of collections are promoted into a less frequently collected area.
Chapter 1. Introduction

1.4 Problems

By overviewing the work on performance of garbage collection that has been done, however, the three performance analysis garbage collection techniques, especially the simulation technique have been explored mainly on functional languages such as FL, ML, and LISP.

The object-oriented programming paradigm appears to be a very powerful programming framework, in which data and data types are abstracted as objects and objects are created whenever they are needed. From our experience with the Emerald language — an object-oriented programming language, we observe that its behavior is rather different from that of the functional programming languages that have been explored in the literature. The questions that arise are: (1) how can the garbage collection techniques that have existed be efficiently applied to object-oriented programming languages? (2) How well will these techniques perform on object-oriented programming languages? In order to answer the questions that were raised above, this thesis is aimed at the performance analysis of garbage collection algorithms, emphasizing the generational garbage collection algorithm on object-oriented programming languages. The goal of this thesis is to first examine the difference between functional programming and object-oriented programming styles, and then understand generational garbage collection in object-oriented programming languages. Two techniques: simulation and measurement of an actual garbage collector implementation are used.

1.5 Thesis Organization

As stated previously, there are mainly three techniques that can be applied for performance analysis of garbage collection algorithms: simulations, implementation reports, and analytic evaluations. In this thesis, we explore two of these performance analysis
techniques, namely, simulations and performance reports. Consequently, the thesis comprises of two parts: a simulation which is used to derive a cost metric, and measurement of an actual implementation.

• Simulation

In this part, we first measure the characteristics of Emerald that affect the performance of garbage collection, and then define an analysis model of generational garbage collection. Following that, the cost metric of generational garbage collection is presented. Finally, we estimate the generational garbage collection cost based on both the simulation results and the cost metric.

• Performance Measurement

In order to verify the simulation results, in the second half of the thesis, we implement a generational garbage collector for Emerald and report on its performance.

1.5.1 Simulations

In order to answer the questions that were raised previously, we started our work doing simulations of Emerald, gathering all the parameters about Emerald that affect the performance of garbage collection.

The simulation examines the difference between the behavior of the Emerald language and that of functional languages. Once we know the difference between these two different programming styles, we then explore how these differences of the system behavior of these two different programming styles affect garbage collection. We have also constructed an analysis model to estimate the cost of generational garbage collection. Based on our cost analysis model and the simulation results, the cost of generational garbage collection of Emerald can then be estimated.
The two assumptions that objects die at a fairly young age and that objects tend to be permanent after they have lived for a certain time have been made about functional languages. The generational garbage collection technique was defined to take advantage of them. Our simulation of Emerald forced us to conclude that these two assumptions do not hold for Emerald. Consequently, we expected that generational garbage collection would not benefit Emerald as much as it does functional languages. In order to support that conclusion, we felt that it is necessary to pursue our work by building a real generational garbage collector for the Emerald language so that real performance measurements can be obtained.

1.5.2 Performance Reports

Since all the simulation reports are obtained by running a conservative mark-and-sweep based collector, these reports might not be accurate. Furthermore, in our cost estimate of the generational collection, the cost of the old generation collection is ignored. In order to validate both the simulation results and the accuracy of the cost estimate, the second objective of this thesis is to build a real generational garbage collector in Emerald, so that real performance measurements can be obtained.

To gain as much understanding as possible of the garbage collection in Emerald, we made several variants of our basic generational garbage collector.

- Simple Copying Collector

A Simple copy collector is a copying based garbage collector in which two semispaces: from space and to space are used. This collector is non-generational - all objects exist in the same semispace. We build this collector by disabling promotion.

- Copying Compacting Collector
There is only one heap space in the old generation collector. Similar to the new
generation collector, it moves objects within the space and compacts the objects
that are still in use to one end of the heap space.

- Mark-and-Sweep Collector

  The performance of the original mark-and-sweep garbage collector which comes
  with the language has been measured for comparison with our copy-based collectors.

- Generation Collector

  Our generation collector consists of two parts: the new generation collector is the
  simple copying collector, and the old generation collector is the copying compacting
  collector described above.

1.6 Thesis Contents

This thesis consists of 6 chapters. Following this introduction, we give an overview of
the work that has been done on garbage collection. In Chapter 3, our motivations for
carrying out this work are explained. Chapter 4 describes the design considerations in
implementing the generational garbage collector. Following that, Chapter 5 presents the
performance reports of the experiments of our four garbage collectors and explains the
results. We conclude the thesis with some discussion and conclusions and also the future
direction of this work is presented.
Chapter 2

Overview of Garbage Collection Algorithms

Since the garbage collection issue was brought up by McCarthy [McC60] in the 1960s, it has been explored and implemented for many systems. Several algorithms and their variations have been proposed and studied in the literature, among which are reference counting, mark-and-sweep, copying-based garbage collection, and generation-based garbage collection. Based on these four basic algorithms there are two other groups of garbage collection algorithms: incremental garbage collection and distributed garbage collection. Incremental garbage collection algorithms are proposed to reduce long pause time caused by garbage collection. Distributed garbage collection are designed to collect garbage in a distributed computing environment. In this chapter, we present an overview of these algorithms.

2.1 Reference Counting

2.1.1 Reference Counting Algorithm

The reference counting algorithm was invented in 1960s [Col60]. It has undergone many refinements [Knu73], [Bev87]. The basic idea of the reference counting algorithm is to keep a reference count for each object, which indicates the number of references to this object. When the reference count reaches zero, it shows that the object is no longer in use, and the space it occupies can be reclaimed.

In general, a reference counting algorithm is implemented by having a reference field
in each object in which the reference count is maintained. When a pointer is assigned to point to an object, its reference count is incremented by one, and when a pointer is assigned to point to some other object, the reference count is decremented by one. Figure 2.1 shows the memory organization of a reference counting system.

When an object is reclaimed, the reference counts of all other objects for which the reclaimed object holds a pointer are decremented. This decrement can cause the reference counts of some other objects to hit zero, so that more objects can be reclaimed due to the reclamation of one object. The reference counting algorithm can be seen as an incremental garbage collection by the fact that the operations of the user program are interleaved with the operations of the garbage collection — one memory operation can cause several garbage objects to be collected.
2.1.2 Variants of Reference Counting Algorithm

There are two important variants of the reference counting algorithm: immediate reference counting and deferred reference counting.

- Immediate Reference Counting

In immediate reference counting garbage collection, reference counts are adjusted on every store operation. Therefore, when a pointer is created or destroyed, the reference count of the object in question must be adjusted accordingly. Furthermore, when the reference count of an object hits zero, the object is collected immediately.

- Deferred Reference Counting

Rather than adjusting reference counts and reclaiming objects whose counts become zero on every store operation, references from local variables are not included in this bookkeeping most of the time. These uncounted references preclude reclamation during program execution. To free dead objects, the system periodically stops, reconciles the counts with the uncounted references for local objects, and reclams all objects whose adjusted counts are zero.

2.1.3 Analysis of Reference Counting Algorithm

The advantage of reference counting collection is that it collects garbage in real-time — garbage collection work is interleaved closely with the running program. Therefore, it can be easily applied to the real-time applications. However, there are two major problems with the reference counting algorithm.

First, it fails to collect garbage in circular structures [BD76]. If the pointers in a group of objects create a cycle, the objects reference counts are never reduced to zero, even if there is no path to the objects from the root set (Figure 2.2 illustrates this problem). In
the early study of reference counting collection, circular structures did not often appear in typical programs. And where they did appear, special efforts were made to avoid the creation of cyclic garbage, or to break circles before they become a nuisance. More recently, however, a study on programming systems has shown a large number of circular structures in them. Modern systems have been forced to supplement reference counting with some other garbage collection schemes to handle cycles. It turns out that systems that use reference counting collection are not effective any more.

The second problem with reference counting collection is efficiency: it takes time proportional to the amount of work done by the running program. There are two major costs of reference counting. Firstly, when a pointer is created or destroyed, the reference count of the referenced object must be adjusted. Secondly, if a variable value is changed from one pointer to another, two objects’ reference counts must be adjusted. Some pointers die shortly after they appear and their reference counts have to be first incremented, then very soon afterwards decremented back.
The deferred reference counting algorithm aims to fix the second problem by avoiding adjusting reference counts for most short-lived pointers from the stack, and thereby greatly reduces the overhead of reference counting. However, the cost of reference counting is still roughly proportional to the amount of work done by a running program in most systems.

2.2 Mark-and-Sweep

The mark-and-sweep garbage collection algorithm was first defined by Knuth [Knu73]. Since then, many variants of this algorithm have been proposed [Ben84], [MDLSS78], [KS77], [CH84], [CH84], [YF69], [Zor90]. A great amount of effort has been made to improve the efficiency of the algorithm and make the correctness proof of the algorithm simple. In the 1970s, with the appearance of multiprocessor computers, the parallelized algorithms of mark-and-sweep were explored, for example, the algorithms for on-the-fly garbage collection [Ben84], in which the user process and the garbage collection process are separated and proceeded in parallel.

2.2.1 Mark-and-Sweep Algorithm

The mark-and-sweep algorithm involves two phases. In the marking phase, the live objects are distinguished from the garbage. This is done by tracing all the live objects from a set of roots in either a depth-first or breadth-first traversal. The objects that are reachable from the root set are marked in some way. After the marking phase is done, the sweeping phase goes through the whole space and collects these objects that were not marked during the marking phase. A free list is maintained to keep all the free space in a system. An optional phase, called the compaction phase, can be added to the mark-and-sweep algorithm which relocates objects so they are packed more closely.
Chapter 2. Overview of Garbage Collection Algorithms

2.2.2 Analysis of the Mark-and-Sweep Algorithm

There are three problems with mark-and-sweep garbage collection: efficiency, fragmentation, and locality.

The performance of the mark-and-sweep algorithm depends on the size of the heap space since the collecting phase goes through the entire heap space to reclaim all unmarked objects onto a free list. The mark-and-sweep collection algorithm was originally designed for machines containing small physical memories. At the time, the execution cost of scanning the entire memory was negligible. Of more direct importance was the need to preserve space in the memory for the mark stack, which in principle requires space proportional to the size of the entire memory. As memory sizes increase, garbage collection overhead increases proportionally. This lack of inefficiency makes mark-and-sweep less well suited for modern environments.

The second problem with mark-and-sweep collection is fragmentation. After a certain number of garbage collections, it may be very difficult to obtain memory to satisfy a big allocation request since all the free memory may be broken into small pieces which may not leave a large contiguous space. The system has to decide how to balance dividing a big fragment up to meet small object allocation requests against the increased utilization gained by the risk of not having enough big fragments to meet big object allocation requests.

Finally, the mark-and-sweep collection has poor locality of reference. Since the objects are never moved, the live objects after a collection are scattered among the new objects that have just been allocated. This results in objects of varying ages being interleaved in the memory, causing mark-and-sweep collection to be considered unsuitable for most virtual memory environments.
Some work has been done to make up the deficiencies of mark-and-sweep collection. For example, a bitmap can be used for mark bits, allowing 32 bits to be checked at once with a 32-bit integer ALU operation and conditional branch. This reduces the sweep time a great deal. Also, the tendency of objects to survive in clusters may mitigate the locality problems of reallocating space amid live objects.

Other work on the mark-and-sweep algorithm has mainly focused on the correctness proof and the analytical study of the algorithms. [Srn74] establishes difference equations which express storage availability between successive collections. These difference equations determine the best size for storage after each collection. In Larson's work [Lar77], the overhead in terms of storage availability is studied. In addition to the work that has been done on the simple mark-and-sweep algorithm, [Wad76] presents two analyses of algorithms for real-time garbage collection.

2.3 Copying-Based Garbage Collection Algorithms

2.3.1 Algorithm Description

Modern hardware trends towards large memories cause the mark-and-sweep algorithm to be inadequate. Consequently, copying based collection was proposed in [YF69], [Bak78], to overcome the deficiency of mark-and-sweep collections.

Unlike the mark-and-sweep algorithm, copying based garbage collection does not collect the garbage, instead, it moves live objects into one contiguous area. After all live objects have been found and moved, the space that no longer contains live objects can be reused. In order to find live objects, the garbage collector starts from a set of roots and traverses all the objects that are accessible using a breadth-first or depth-first scheme.

The most often used technique for copying based collectors is the semispaces scheme as shown in Figure 2.3. In this scheme, the heap is divided into two spaces of identical
Chapter 2. Overview of Garbage Collection Algorithms

Figure 2.3: A Simple Semispace Garbage Collector Before a Garbage Collection

size: fromspace and tospace. Objects are created from successive memory locations in fromspace. The garbage collection process traces accessible objects, incrementally evacuating objects, that is moving them from fromspace to tospace. When no more accessible objects remain in fromspace, the entire fromspace can be reused. An operation called flip occurs, where the tospace becomes the fromspace and vice versa. In doing the moving, some mechanism is needed to ensure that objects that are reached by multiple paths are not copied to tospace multiple times. When an object is moved, a forwarding pointer is left at its old address. The forwarding pointer indicates where the new address of this object is, so an object that contains a reference to this object can find the new address. Figure 2.4 shows the memory organization after a collection.

There are two schemes for determining when the garbage collector should be invoked: the fixed collection space scheme and the fixed collection threshold scheme. In the fixed collection space scheme, a garbage collection is called to reclaim space when an allocation request from the running process does not fit in the unused space. There is a potential thrashing problem using this strategy. After the available space reduces down to a certain
value, the time required to fill up this space is not sufficient to allow enough new objects to die. Therefore, a garbage collection is invoked to collect very little available space for reuse. Consequently, garbage collections are invoked frequently and collect less and less garbage. The fixed collection threshold scheme avoids the thrashing problem by invoking a garbage collection when a fixed amount of memory has been allocated.

There are many variations on copying based algorithms [LH93], [Pix88]. The most simple one is the simple-stop-and-copy algorithm, in which the user program and garbage collector work sequentially. In other words, user program stops every time the collector starts collecting garbage. Therefore, the garbage collection does not cause any overhead to the user program. This algorithm is efficient for applications that have a large memory space, however it is insufficient for real-time or interactive applications in which the pause time caused by garbage collection is critical.

Figure 2.4: A Simple Semispace Garbage Collector After a Garbage Collection
2.3.2 Analysis of Copying Based Collection Algorithm

Unlike mark-and-sweep garbage collection, the cost of copying based collection is proportional to the number of live objects at the time collection is done. So, a copying based garbage collector can be made efficient, if sufficient memory is available [Lar77], [App89]. For example, assuming that the number of live objects are approximately the same at each collection, only the number of collections and not the size of the heap will affect the total collection cost. A simple way to decrease the number of collections is to make the interval between successive collections longer. The increase of the available space will decrease the frequency of the garbage collections simply because it takes longer to fill up the available space. The copying based collector can be made very efficient with a sufficiently large amount of memory, but the cost of extremely large memories is too high. In particular, copying based collection in a very large memory is unsuitable for virtual memory applications due to its poor memory locality. Therefore, in general, we can not achieve the potential efficiency of the simple copying based collection scheme. In addition, for real-time applications, it can be disruptive if the collection pause time is fairly long. In such systems, having a large available memory does not reduce this long pause time. Generational based garbage collection appears to be necessary to fix this problem.

2.4 Generational Garbage Collection Algorithms

Generational garbage collection is a variant of copying based garbage collection. Since the algorithm was first proposed by Lieberman [HL83], it has been implemented for many language systems, including Smalltalk [Ung84], PML, and LISP [Zor89]. It performs more efficiently than simple copying algorithms for these systems. In the following sections, we describe the generational garbage collection scheme and discuss its advantage over
Chapter 2. Overview of Garbage Collection Algorithms

mark-and-sweep and simple copying based algorithms.

2.4.1 Algorithm Description

Rather than traversing through the whole heap space, generational garbage collection focuses only on collecting the objects that are newly created. In fact, the memory is divided into several small regions, each of which keeps objects with different ages. The objects that are in the same region are of approximately the same age. Garbage collection is performed on different regions at different frequencies. The newly created objects are placed in the new generation and therefore collected more frequently. The older regions contain objects promoted from the younger regions after these objects have experienced a certain number of collections in that younger region. In general, more than two generations can be used for generational garbage collection, it however increases the complexity of the algorithms. In order to keep the algorithm simple, two-generations scheme is commonly used in the literature, where the heap space is divided into two regions: the new generation and the old generation. In generational collection, the collection starts out with a set of root pointers which include global variables, run-time stacks, registers, and the records that contain the pointers from other generations into the one being collected. Figure 2.5 shows the object distribution before a garbage collection.

The efficiency of the generational collection algorithm is determined by two factors: the size of the remember set and the object life-time distribution.

- Remember Set

In order to perform generational collection correctly, a collection of remember sets that records all the pointers from the older generations to the newer generation must be maintained. Each store operation has to be checked to see if it makes a reference from an older generation to a newer generation. If it does, a trap occurs so that this
Figure 2.5: A Generational Collector Before a Garbage Collection
Figure 2.6: A Generational Collector After a Garbage Collection
reference can be recorded in the appropriate remember set. Figure 2.6 illustrates the object distribution after a garbage collection. Each remember set contains the locations of the pointers that point from an older generation to a newer generation so that they can be used as roots when the newer generation is scavenged to find all reachable objects. The cost of recording intergenerational pointers (maintaining the remember sets) is proportional to the total number of store operations in a system as well as the number of intergenerational store operations.

• Object Life-time Distribution

In each collection cycle, the dead objects are distributed among different regions. If it is the case that most objects die while in the new generation, the collection can be concentrated on this region and does not need to be concerned with examining the older (less likely to die) objects.

2.4.2 Analysis of Generational Collection Algorithm

Generational garbage collection is claimed to have good performance [HL83], [Ung88] for two reasons. Firstly, it takes into consideration the lifetime of objects. The objects that have lived long enough are considered old and are expected to live for a long time and therefore are moved out of the new generation. This avoids repeatedly copying these old objects in the new generation. The objects with short lifetime will be collected shortly after they become inaccessible in the new generation. Secondly, it takes advantage of the fact that the objects that have lived long enough tend to be permanent, so that after these objects are moved into the old generation, they do not become garbage immediately. Since the old generation is collected less frequently than the new generation, promoting the objects that will live relatively long reduces the amount of space occupied by garbage objects in the old generation.
However, there are several questions associated with generational collection that must be handled properly to obtain a good performance, namely the promotion rate (copy count), the space dividing, and the collection threshold.

In generational garbage collection, each object is associated with an age. An age is assigned to each object at the time the object is created. The objects that are allocated during two successive collections are of the same age (age 1) and their ages are increased by one after each garbage collection. The promotion rate (copy count) determines the age of objects that are promoted into the old generation and therefore the speed at which the old generation is filled up. A high promotion rate releases the burden of the collection of the new generation. However, it causes a large number of old generation collections.

In [Zor89], a wide range of promotion rates are tested. Promoting all live objects that survive a garbage collection en masse results in an increased promotion rate, promoting more than 20% more objects than copy count promotion with small thresholds, and up to 100% more objects than with larger thresholds.

Space dividing is another important issue that affects the cost of generational garbage collection. In a collection environment, in which the heap space is divided into two generations, the boundary between the new generation and the old generation must be determined properly so that good performance can be obtained. Usually, increasing of the size of the new generation causes the cost of collecting the new generation to go down, but the corresponding decrease in the size of the old generation makes the cost of collection of the old generation go up. Therefore, the system must decide the balance between the cost of the garbage collection of the new generation and the cost of the garbage collection of the old generation.
2.5 Incremental Garbage Collection Algorithms

When a garbage collection is initiated, the execution of a user program is suspended and its execution can not be resumed until the garbage collection is finished. The main drawback in this simple "stop-the-world" garbage collection approach is that it causes large pause times due to the large garbage collection overhead. Consequently, incremental mark-and-sweep was proposed to reduce the pause time [Bak78], [Coh81], [WW87], [LAE88], [App89], [Yip91].

In order to meet the critical time requirements for real-time applications, the length of time that a user program is interrupted must be limited. This time requirement naturally limits the pause times that garbage collection can have on a user program. Two proposals have been made to reduce the length of these pause times. One approach is to perform garbage collection gradually by doing a portion of the work required by an on-time garbage collection within a tolerable time bounds. Another way is to perform the garbage collection incrementally, where, the garbage collector scans only a bounded number of "live" objects at a time. There are two programs involved in the collection; we call these the collector and the mutator.

The responsibilities of the collector and the mutator are determined by the garbage collection algorithms that are used. In mark-and-sweep algorithms, the collector marks the used objects and collects the unmarked ones into the free list. The mutator executes the user program and also cooperates with the collector, performing tasks such as marking the objects that get modified by the mutator during the collection. Like in the mark-and-sweep algorithms, in copying based algorithms, the mutator is responsible for executing the user program. The collector gets triggered by the mutator when a root object gets accessed. The collector moves the objects that can be reached by the root object to another location.
2.6 Distributed Garbage Collection

With the appearance of distributed systems, distributed garbage collection algorithms were designed to collect garbage in a distributed environment [Bis77] [HK82] [EL87] [SDP92]. A distributed environment consists of a collection of machines, each with its own local memory, connected by a network such that pointers on one machine may directly address memory locations on another machine. In order to collect garbage in a distributed system correctly and efficiently, distributed garbage collection algorithms must take into account the property that objects on one machine may have links across machine boundaries.

Distributed garbage collection algorithms are very implementation-dependent, which causes a different algorithm to be designed in a different language implementation. Viewing the work that has been done in this field, based on the basic algorithms (reference counting, mark-and-sweep) they use, distributed garbage collection algorithms fall into several categories. In [Bis77], the algorithm requires each machine to keep track of the objects which have pointers to live objects of this machine's address space.

The algorithm that was used in [HK82] were designed for a functional programming language. The algorithm takes advantage of the fact that the results of a functional programming language form a marking tree. The algorithm which is based on the mark-and-sweep algorithm is spawned on the root of the tree and is recursively spawned on each subtree. When marking reaches the leaves of the tree, they are colored and then the direct ancestors are recursively colored until the root is colored. When the root of the tree is colored, then the sweep phase can safely begin.

The algorithm in [EL87] is based on reference counting algorithm. In this algorithm, the pointers to objects are divided into two classes: defining and borrowed. A defining reference is the first, and often the only, reference to an object. If a copy of the pointer is
made then the copy is a borrowed reference since a portion of the object is being borrowed. The primary problem is recognizing when a borrowed reference should become a defining one, and to not continue collection beyond that point in the object when such a change is made.

[SDP92] proposed a more general, programming language-independent, distributed garbage collection algorithm. Unlike existing schemes in which local objects are distinguished and treated differently, in [SDP92], all references are modelled uniformly to provide a location-transparent reference mechanism.

There are two types of objects in Emerald distributed programs: local objects and remote-referenced objects. The local objects can be reached by a set of local roots that reside on a local machine and are garbage collected locally. The remote-referenced objects are referenced by objects that cross the machine boundaries. In order not to collect the remote-referenced objects during local garbage collection, the remote objects are treated as part of the root in local garbage collection and therefore are not collected. A global garbage collector can be invoked when necessary to collect all remote-referenced objects in a distributed system. In this thesis, we only consider local garbage collection in Emerald. Global garbage collection is the topic of ongoing work.
Chapter 3

Simulations and Cost Metric

The work on garbage collection has focussed on generational techniques [Zor89], [Ung84], [Bar89], and if we believe what these researchers tell us then we are forced to conclude that generational collections offer the most hope for high performance storage reclamation. However, this recent work has been done primarily in the context of functional programming languages, and it is not clear how well it extends to languages based on other paradigms - in particular the object-oriented language Emerald.

The work that was done in an object-oriented programming language can be found in [Bar89], [Yip91], [Ono93]. In [Bar89] and [Ono93]'s work, it was mainly focussed on implementing a generational garbage collector for a C-based object-oriented language (C++). While [Bar89] improved the algorithm that was used for garbage collection and reduced the time the garbage collection took, [Ono93] compared the generational garbage collection technique with the explicit storage management technique. However, these two pieces work did not address the question of, compared to functional programming languages, how well generational garbage collection can perform on object-oriented programming languages. Furthermore, they did not investigate the factors that cause generational garbage collection to obtain good performance. Thirdly, there is no evidence that generational garbage collection technique outperforms other traditional garbage collection techniques, such as single space, copying based garbage collection.

In this chapter, we first introduce three common techniques that can be used to measure the performance cost of garbage collection algorithms. Then we report the
simulation results that were obtained by doing simulation on Emerald programs. Since the behavior of both mark-and-sweep and simple copy algorithms is straightforward, in this chapter, we mainly focus on estimating the cost of the generational garbage collection of a typical Emerald program: the compiler.

First of all, we describe the simulations we have done for Emerald. We then define a cost metric to analyze the cost for generational garbage collection. Following that, the collection cost is estimated based on our cost metric. Finally, we explain our motivations for continuing our work and implementing a generational garbage collector for the Emerald language so that a thorough understanding of effect of the garbage collection algorithms on object-oriented programming languages can be obtained.

3.1 Evaluation Techniques

The evaluation studies of garbage collection fall roughly into three categories: implementation reports, analytic studies, and simulation-based evaluation. In the following sections, we explain each of them in detail.

3.1.1 Implementation Reports

This category is the most common form of garbage collection algorithm evaluation. A particular algorithm is implemented in the context of particular hardware and software, and a report is written to confirm that the implementation of a proposed algorithm was successful.

There is one big advantage of doing an implementation report. Implementation reports measure the real performance of an actual implementation. A set of programs can be executed and timed and the results are unequivocal.

One of the disadvantages of doing an implementation is that it is highly costly and
time-consuming. Another inherent limitation of implementation reports is the restricted range of system parameters that may be explored. Because the implementation is specific to a particular hardware and software configuration, exploration of parameters outside the scope of the particular system is impossible.

3.1.2 Analytic Evaluations

Another technique for performance evaluation of garbage collection uses analytic models of the algorithms to predict performance. Instead of writing detailed implementations of the algorithms, evaluators construct high-level mathematical models of the operations performed during garbage collection. It is usually used to determine the characteristics of the algorithms.

A problem with analytic performance evaluation is the number of simplifying assumptions that must be made. Typical assumptions include a constant rate of allocation, a constant rate of deallocation, a constant marking rate, etc. In practice, some of these assumptions are simply not true.

3.1.3 Simulations

Another approach that can be adopted to evaluate garbage collection techniques is through simulation [Par68]. Simulation of garbage collection techniques has been used for many years. For example, Newman and Woodward described a simulation of Lamport’s algorithm for on-the-fly multiprocessor garbage collection [NW].

Simulation usually provides believable results without the high cost of an implementation. In addition, simulation allows flexible parameterization of the execution environment to allow wide exploration of the space of design parameters. Thus, simulation appears to be an effective tool for performance evaluation of garbage collection algorithms.
One drawback of the simulation technique is that it only provides estimates of performance measurements which may not conform with the real performance measurements. And also, some application-dependent parameters may not be measured accurately by simulations.

3.2 Emerald Experiments

Due to the difficulties of different language implementations and hardware configurations, previous work on performance analysis of garbage collection has focused on cost estimates by doing simulation of systems. In fact, the cost estimate was only performed on functional languages, for instance, ML and LISP. There has been little research done on object-oriented programming languages, in which one would expect the behavior of the program to be rather different than that of traditional functional programming languages.

In order to understand generational garbage collection of object-oriented programming languages and be consistent with the work that has been done, we started our work doing simulation on object-oriented programming language - Emerald and gathered all the parameters that affect generational garbage collection.

3.2.1 Brief Introduction of the Emerald Language

The Emerald programming language is a strongly-typed programming language that supports the object-oriented programming paradigm. It provides object constructors for run-time creation of objects. The basic unit of the language is the object which is an abstraction for the notions of data and type. Objects are created whenever they are needed as a program executes. Unlike Smalltalk and ML which allocate activation records in the heap, Emerald is stack based, which reduces somewhat the demands placed
on the garbage collector.

3.2.2 The Existing Emerald Garbage Collector

Our experiments begin with an implementation of Emerald on a SparcStation using a mark-and-sweep based garbage collector which was distributed by Xerox PARC [BW88] and was modified to support Emerald. This collector is a conservative garbage collector in that it may fail to reclaim some inaccessible objects. Furthermore, the collector assumes that every accessible object is referenced by a pointer falling into one of the following places:

- the stack segment
- the registers
- an accessible block

3.2.3 Experiment Program

Since simulation results are affected by the benchmark programs from which the parameters are gathered, choosing typical programs as benchmark is an important issue in doing a simulation.

Since the behavior of the Emerald system is fairly different from that of the LISP system, the typical programs of the LISP system that are used as benchmarks, such as the quicksort program, may not be the proper benchmarks for the Emerald system. In our simulation, the Emerald compiler which is written in Emerald is used as our benchmark program due to two reasons. The compiler of a programming system, in general, well reflects the characteristics of that programming system. Another reason is that the LISP compiler program was one of the programs that were used for estimating the cost of
the generational garbage collection of the LISP system, it therefore makes it possible to compare the simulation results of the Emerald system with those of the LISP system using the Emerald compiler program as a benchmark program in our simulation.

3.2.4 Our Experimental Model of Generational Garbage Collection

In general, generational garbage collection, in order to take advantage of the fact that the objects die at a young age, is directed towards garbage collecting young generations more frequently. Therefore, rather than copying all live objects in the whole system at each garbage collection, most often, generational garbage collection only copies objects that are in the young generations. This is done by promoting objects that have experienced several collections into an older generation.

In our analysis, only two generations are considered. Furthermore, both generations use the stop-and-copy garbage collection algorithm, where an application system’s normal execution stops when a garbage collection is in progress. Therefore, there is no overlap between the user process and the garbage collection process.

In our experimental model, a garbage collection starts after a certain amount of space (collection threshold) is allocated. In order to avoid copying the objects that have lived for a long time, they are promoted into the old generation after having experienced a certain number of collections (copy count).

3.2.5 Cost of Generational Garbage Collection

There are two components to the cost of generational garbage collection: the total number of live objects that need to be copied and the cost of maintaining the remember set.

In turn, the number of live objects that need to be copied depends on the distribution of the live objects between the new and the old generations. That is, the fewer live objects that live in the new generation, the less time spent by the garbage collector
in copying them. For instance, in a system with a constant object death rate (a fixed number of objects die at certain time intervals), if most objects that have died during two successive garbage collections are mainly distributed in the old generation and therefore all the live objects live in the new generation and are collected, the time spent by the new generation collection will not be reduced by promoting objects into the old generation. In the meantime, the space that is occupied by the dead objects in the old generation is wasted since the old generation is rarely collected. Wasting space in the old generation has the side effect of requiring more frequent collections on the new generation.

The second component in the cost of generational garbage collection is the maintenance of the remember set. In order to perform generational collection correctly, we must pay the extra cost of recording all the references that point from the old generation to the new generation. Additionally, all store operations have to be checked. The cost of checking increases the cost of each store operation and therefore the total overhead of garbage collection.

### 3.2.6 Generational Garbage Collection Parameters

Based on our analysis model, several fundamental parameters that affect the performance of the generational garbage collection are characterized. The parameters can be mainly divided into two categories: the user-tuned parameters and the system-dependent parameters. The garbage collection threshold ($CT$) and the promotion rate (copy count) ($PR$) fall into the first category, while, the live object lifetime distribution ($LOD$), the number of store operations ($NS$), and the size of the remember set ($NR$) fall into the second. In the following sections, we explain the relationship between these parameters and how each of them affects the generational garbage collection.
Chapter 3. Simulations and Cost Metric

<table>
<thead>
<tr>
<th>Threshold</th>
<th>125K</th>
<th>250K</th>
<th>500K</th>
<th>1000K</th>
<th>2000K</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collection Number</td>
<td>331</td>
<td>165</td>
<td>82</td>
<td>41</td>
<td>20</td>
</tr>
</tbody>
</table>

Table 3.1: Collection Number

Garbage Collection Threshold

Garbage collection threshold is a user tuned parameter. As the user program proceeds, the usage of memory varies from time to time. It will be expensive and impractical to have a shot of memory at every memory change. Instead, we record the memory usage at a certain time interval which is determined by the collection threshold. A garbage collection is invoked whenever collection threshold is hit. Three parameters that affect the cost of the generational garbage collection, namely, the number of garbage collections, the size of the remember set, and the live object distribution, are determined by the collection threshold.

Firstly, the collection threshold indicates, with total allocation bytes, how many garbage collections need to be done during the entire system life time. Larger numbers of collections are expected with a smaller collection threshold. However, the number of live objects that need to be copied in each collection cycle will be reduced with a small collection threshold. Table 3.1 shows the number of collections as a function of collection threshold of the Emerald compiler program.

The collection threshold, together with the promotion rate, also affects the live object distribution and the size of the remember set. The effect of the collection threshold and the promotion rate on live object distribution and the size of the remember set will be explained in the following sections.
Promotion Rate

Another important issue that needs to be considered in the generational garbage collection is the promotion rate [Ung88] which determines when the objects are promoted into the old generation. Like the collection threshold, the promotion rate is also a user-tuned parameter that affects the generational garbage collection. Ideally, promoting objects from the new generation into the old generation aims to avoid copying objects in a system back and forth when they live for a long time. However, if we promote premature objects into the old generation, they will die shortly after. Since the old space is not collected frequently, the space that is occupied by the garbage will not be released — wasting space. On the other hand, if the objects that live in a system for a fairly long time are not promoted on time, they will be repeatedly copied, which in turn increases the copy cost of the garbage collection.

Therefore, choosing the right promotion rate is another very important issue of the generational garbage collection. The following sections will explain how collection threshold and promotion rate affect the live object distribution and the remember set cost.

Live Object Distribution

In our model, a garbage collection is invoked when the collection threshold is reached. At this point, the live objects in the system are distributed either in the new generation or in the old generation. If a system fits in the assumption that most objects die at a young age, the number of live objects in the new generation should be fairly small due to the fact that the new generation only contains objects that exist only for a short period of time. This distribution depends on both the collection threshold and the promotion rate (copy count).
Store Operations and Remember Set

Since the garbage collection is done only to the objects in the new generation, the references (remember references) that point from the objects in the old generation to the objects in the new generation must be maintained in a table (called the remember set) in order to find all the live objects in the new generation. This remember set is used as part of the root set to trace out all live objects in the new generation. A trap occurs to add a remember reference to the remember set. The cost of maintaining this set is proportional to the number of store operations in the whole system, and also to the number of the remember references.

3.3 Cost Metric

Since there is usually a big available space for the old generation, and the garbage collection on it is very infrequent, we ignore the garbage collection cost of the old generation in our cost analysis. In the following sections, garbage collection is limited to the new generation. In this section, the analysis of the cost that the generational garbage collection puts on an object-oriented language will be estimated.

3.3.1 Total Cost Metric

In our analysis, the cost of the generational garbage collection is broken down into two parts: the cost of maintaining the remember set, and the cost of copying live objects in each collection cycle. This cost therefore can be modeled as:

\[ TC = T_{storep} + T_{copy} \]  

(3.1)

\( TC \) represents the total cost of the generational garbage collection. Besides the cost of copying live data from one semispace into the other, which is represented as \( T_{copy} \), in
our model, additional time must be spent in checking each store operation in a system to record every reference that points from the old generation to the new generation. This cost is represented as $TC_{\text{store}}$ in our cost metric. The dominating cost for the generational garbage collection, as in the simple copying, is the copying cost ($TC_{\text{copy}}$). This includes detecting the live objects and moving the live objects from one semispace to the other.

In the following sections, we first measure the parameters that affect the above two costs, and then estimate the two costs based on the measured parameters.

### 3.3.2 Cost Metric for Copying Live Objects

The cost of copying live objects in the system dominates the entire cost of garbage collection. This cost can be modeled as:

$$T_{\text{copy}} = NC \times OC$$  \hspace{1cm} (3.2)

$T_{\text{copy}}$ is the total cost of copying live objects from one semispace to the other in the new generation. $NC$ represents the number of executions of the garbage collection with a certain collection threshold. $OC$ is the total size of objects that are copied at each garbage collection cycle. $NC$ is determined by the live data distribution, while $ON$ is determined by the collection threshold.

### 3.3.3 Cost of Store Operations

The ratios $R_{\text{store}} \ (NS/NO)$ and $R_{\text{remember}} \ (NR/NO)$ determine how costly the store operations are in the generational collection, where $NS$ is the total number of store operations in a system, $NR$ is the size of the remember set, and $NO$ is the total number of reference operations in the system.
3.4 Result Reports

3.4.1 Live Object Distribution

In our experiment, live objects in each collection are measured as a function of the collection threshold with different copy counts (promotion rates). To be consistent with Zorn's\[Zor89\] work, collection thresholds of 128kbytes, 256kbytes, 512kbytes, and 1024kbytes have been chosen. Figure 3.7 shows the live object distribution with different collection thresholds and copy counts.

3.4.2 Cost of Copying Live Objects

Based on our cost analysis model, the copying cost is determined by the ratio of the total number of objects that get copied in all collections over the total objects that are allocated. Figure 3.9 indicates that the overhead of copying live objects will be doubled when we halve the size of the collection threshold. This is due to the fact that very few objects that have survived for one collection will die during the time that the next collection occurs. This makes the cost of copying live objects in the collection twice what we would see with a halved size of the collection threshold. Figure 3.10 shows that for each \(i = 0, 1, 2, \ldots, n\), the percentage of objects which die between the \(i\)th and \((i + 1)\)th garbage collection.

3.4.3 Cost of the Store Operations

The size of the remember set is measured as a function of the collection threshold and the promotion rate. The ratio of store operations over the total memory operations \(R_{storep}\) is 5.2\%. Figure 3.8 presents the ratio of the size of the remember set over the total number of memory operations as a function of the collection threshold.
3.5 Explanations

Based on our analysis model and the experimental results, the generational garbage collection works lower efficiently in Emerald than it does in the Lisp system. There are two reasons for this less efficiency: expensive store operations and varying object longevities.

Table 3.2 illustrates the ratio of the size of the remember set to the total number of the store operations and the ratio of the total number of the store operations to the total number of the memory operations in the LISP system [Zor89]. The above table is obtained with a 512K threshold size assuming stop-and-copy garbage collection with copy count promotion after three copies.
Chapter 3. Simulations and Cost Metric

<table>
<thead>
<tr>
<th>Event Ratio</th>
<th>ACLC</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_{storep}$</td>
<td>5.6</td>
</tr>
<tr>
<td>$R_{remember}$</td>
<td>0.007</td>
</tr>
</tbody>
</table>

Table 3.2: Store Operations and Remember Set Ratios in LISP

Figure 3.8: The Ratio of Remember Set over Total Memory Operations as a Function of Threshold
Figure 3.9: Copying Overhead of the Collection in Terms of Objects Allocated
Our experiments with the store operations indicate that the ratio of memory store operations over total memory operations $R_{store}$ is about the same as the one in the LISP programming environment (5.6%). But with a 512K collection threshold and copy count of 3, the ratio of remember size over memory operations (Figure 3.8) is 10 times higher than that of the LISP system (Table 3.1). Consequently, the cost for the traps that record the references into the remember set is about 10 times higher than that of the LISP system. Given the fact that the overhead of handling the remember set makes up most of the overhead of store operations, it will make the cost of handling the store operations much higher than that of LISP operations. Therefore, the higher cost for handling store operations is expected.

By Zorn's analysis of the LISP system [Zor89], the cost of the store operations makes
up 30% of the total cost of the garbage collection. Correspondingly, our experiments with Emerald have also shown that store operations were highly costly. The combination of these expensive store operations and their large proportional make-up of the total cost of garbage collection results in a much higher total garbage collection cost compared to the Lisp system.

Our experiments with Emerald also indicated that objects die at different ages; therefore, the assumption that most objects die at a young age is violated. This assumption is required for the generational garbage collection to perform efficiently.

Though our experimental results show less efficiency of the generational garbage collection in an object-oriented programming paradigm, there are still several unsolved questions. Firstly, in estimating the cost of garbage collection, the old generation is ignored due to the difficulties of building an accurate cost model for the old generation collection. In our experimental results, however, the cost of the generational garbage collection varies dramatically with different copy count. By the fact that the copy count determines both the old generation collection frequency and the total live objects of the garbage collection, ignoring the cost of the old generation collection in estimating the total cost of the generational garbage collection therefore may not provide an accurate result.

Secondly, since the garbage collector that is used to perform our experiments is a mark-and-sweep based conservative collector, it may in turn cause the experimental results to be unaccurate. Finding the cost of the old generation collection and the real generational garbage collection motivates us to continue our work and implement a generational garbage collector for Emerald, so that a real performance measurement of the generational garbage collection in the Emerald system can be reported. In the next chapter, we report the real performance of the generational garbage collection in Emerald.
Chapter 4

Implementation

In this chapter, the algorithms that are used to implement the generational garbage collector are presented. For the mark-and-sweep collector, we used the one that already existed in Emerald without any changes. For the detailed description of the mark-and-sweep collector, please refer to Chapter 3.

As we described in Chapter 3, the heap space is divided into two parts: space for the new generation, and space for the old generation. Correspondingly, we have a new generation collector which is in charge of doing garbage collection on the new generation objects, and an old generation collector which does garbage collection on the objects in the old generation.

4.1 Emerald-related Implementation Considerations

4.1.1 Objects in Emerald

Objects in Emerald fall into four categories: process identifier objects, system objects, concrete type objects, and normal objects. In our implementation, they are distinguished and processed somewhat differently during garbage collection.

Emerald is a concurrent processing language, and therefore allows several processes to be created in Emerald programs. Each of the process has a process identifier object associated with it. Process identifier objects are permanent and self-contained, i.e., they
Chapter 4. Implementation

Figure 4.11: Emerald Object Structure

are not pointed to by any other objects, and also do not contain pointers to other objects. During garbage collection, process identifier objects are treated as part of the root pointers and therefore not collected.

The second group of permanent objects are system objects which consist of the current time and date as well as some other system information. In order to avoid collection of these objects during garbage collection, like process identifier objects, system objects are distinguished and treated as root pointers.

Another important group of objects are the concrete type objects which maintain the system description of how objects behave. A concrete type object contains the name, a description of the object, and the operations of the object. This group of objects usually live during the entire system life time and therefore are not collected.

The most common objects that appear in Emerald programs are normal objects. Every such object consists of a unique data structure which contains a pointer that points to the object’s concrete type object and several data fields which are decided by a data structure, called a template contained in its concrete type object. Figure 4.11 shows the structure of a normal Emerald object.
4.1.2 Object Allocation

As stated in the previous section, we distinguish between four types of objects in Emerald. According to the category it falls into, an object resides in a designated memory region. In our implementation, normal objects are allocated from the new generation in the simple copying collector and the generational collector, but from the old generation in the copying compaction collector. Since the concrete type objects, the system objects, and the process identifier objects are considered permanent objects in Emerald, they are allocated from the old generation in all the collectors. Consequently, less frequent or no garbage collection is needed on them. In our implementation, besides the concrete type objects, the system objects, and the process identifier objects, the old generation also contains the promoted objects from the new generation which have survived for a certain number of collections in the new generation.

4.2 New Generation Collector

A copying based, two-semispace algorithm was used to implement our new generation collector. This section discusses the issues involved in implementing the new generation collector.

4.2.1 Space Dividing and Object Allocating

The new generation is divided into two equal parts, from space and to space. The system maintains two pointers into the heap: fromSpace, used for satisfying allocation requests, and toSpace which initially points to the beginning of the to space. When an allocation request is received, the system allocates a memory with an extra word (in which the age of this object is maintained) from the fromSpace pointer and advances the fromSpace pointer according to the size of the object allocated.
4.2.2 Trace out Live Objects

The new generation collector traverses all live objects from a set of roots, which, in our case, consists of an object table, a set of system objects, all of the process identifier objects, and a remember set. The objects that are in the object table and the remember set include both normal and concrete type objects, and usually they contain nested pointers. The collector must find the template associated with each object which interprets each data field of the object. After that, the collector goes through every data field to find more objects that are pointed by this object in a breadth-first manner, and moves each object so discovered into to space.

After all live objects are moved into to space, from space and to space are flipped, consequently, fromSpace points to where toSpace pointed to and toSpace points to the beginning of the to space which, in turn, was the from space before the flipping. Figure
Figure 4.13: State of the New Generation After a Garbage Collection

4.13 shows the state of the new generation after a garbage collection.

4.2.3 Finding New Homes and Forwarding Addresses

As the new generation garbage collector traverses the data fields of the root objects and the objects that have already discovered to be alive, additional live objects are detected. When an object is found to be alive and has not yet been moved, a new home is allocated in the to space. Rather than moving this object right away to the new home, we store it in a user stack. The user stack is handled after the data fields of the root object or the live object that contains the pointer of this object has completely been moved.

A forwarding address which indicates the new home of the object is left at the old address. In order to use one word in an object for the forwarding address, we move one word of a header field of the object to its new location, so that this word can be used to indicate the forwarding address of the moved object and this word also contains a flag to indicate that the object has been moved. When an object is found to contain a pointer to an object that has been moved, the pointer to the moved object is adjusted to point to the new home of the moved object.
4.3 Old Generation (Compaction) Collector

When there is no space remaining in the old generation for the objects that will be promoted from the new generation, a garbage collection on the old generation is initiated. Since the old generation, in most cases, requires a large memory, it is not practical to use a semispace algorithm to build the old generation collector. In order to be consistent with our new generation garbage collector, we decided to build a copying based collector which works in a single space and does compaction as well.

In order to meet our design goal for the old generation collector, we designed a new algorithm which consists of three phases: marking live objects, finding the boundary between live and dead objects, and moving objects. In the marking phase, no objects are copied. The collector traverses all the live objects from a set of roots which include:

- object table
- system objects
- process identifier objects

and marks them. While marking the live objects, the collector keeps track of the total number of word consumed by live objects as well as the distribution of the sizes of the live objects. When a collection is done, all the live objects will be compacted in a single contiguous region of the old generation. However, since the data structures that guide the garbage collector (the templates contained in the concrete type objects) are in the old generation, we cannot overwrite existing live objects in order to finish the compaction. Instead, we fill in the holes between existing live objects near the front of the old generation space with live objects from near the end of that space. A live object that is in this front region before the move will remain at the same address, only the objects that are in the back region will be moved during the moving phase. So, the
question that remains is to determine the boundary. The criteria is that all the objects beyond the boundary must fit in the holes in the space before the boundary. Using the information about the total size as well as the distribution of the sizes of the live objects obtained in the marking phase, finding the boundary phase sweeps the whole heap space and fits all the live objects (the ones that will not be moved and the ones that will be moved) in the front region of the heap space. At last, the collector starts moving every object into the home that has been found in the finding the boundary phase, and leaves a forwarding address in the old address of the object.

4.4 Variants of the Collector

In this section, we explain, based on the two collectors (the new generation collector and the old generation collector) we built, how to turn them into three typical copying based garbage collectors, namely, a simple copying collector, a copying compacting collector, and a generational collector.

4.4.1 Simple Copying Collector

As we explained in the previous section, our new generation collector is a copying based semispace collector. It is designed as part of our generational collector which detects and collects the garbage in the new generation.

In our implementation, there are two heap regions: the new generation region and the old generation region in the system. For the simple copying collector, the new generation contains normal objects that are allocated during the system execution. When it is filled up, a new generation garbage collection is called. In general, the objects in the old generation come from two different sources: allocating and promoting. The concrete type objects are allocated from the old generation during system initialization, and are almost
permanent. The other portion of the objects in the old generation come via promotion from the new generation.

In order to measure the performance of the simple copying collector, we shut off our old generation collector by not promoting any objects into the old generation, so that no old generation garbage collection is ever needed, which effectively turns off the old generation collector. To avoid promoting any object into the old generation, a very large copy count is given so that no object can actually go through that many garbage collections. Consequently, no object is actually promoted into the old generation.

4.4.2 Copying Compacting Collector

As we described in Section 4.3, the algorithm we used to implement the old generation collector is a copying-based compacting one, and the old generation collector is invoked when the old generation is full. Unlike the simple copying collector, all objects are allocated from the old generation. Since there are no memory allocation requests happening in the new generation, the new generation collector, though it is called when the old generation collector is called, does not add any measurable cost to garbage collection.

4.4.3 Generational Collector

Again, the heap space is divided up into two regions: the new generation and the old generation. Objects in the new generation are allocated upon request during the system execution. Objects in the old generation are both the concrete type objects that are created at system initialization and the objects that are promoted from the new generation.

In the generational collection, the new generation collector and the old generation collector work separately. There are two approaches which can be used to determine when a new generation collection is invoked: fixed collection threshold and fixed allocation
threshold. These options are explained in the next section. Determining when to invoke the old generation collector is simple: it is called when the old generation fills up.

4.5 Parameters in the Garbage collectors

We have described four different garbage collector implementations: simple copying (new generation collector), copying and compacting (old generation collector), generational collector (new generation and old generation collector), and mark-and-sweep collector. In this section, we describe the parameters that affect the performance of each collector described above.

4.5.1 Heap Space

For the simple copying collector and the generational collector, the heap space is divided into two parts: the old generation and the new generation. In the simple copying collector, the old generation is only used to allocate the concrete type objects which are permanent in the system and are used, during a collection, as a part of our root pointers to trace out all live objects. The old generation in generational collector contains, in addition, objects promoted from the new generation. The heap space is managed as a free list which contains the free space with different sizes for the mark-and-sweep collector. In the copying compacting collector, all objects are allocated from the old generation.

Theoretically, copying based collection can be made very efficient with an arbitrarily large heap space. Therefore, the difference between different collectors will be blurred by a heap space that is too large. From experience with the Emerald compiler system, we observed that the maximum live objects in the system at any given time is roughly 2MBytes. So in order to observe the fine variations between each collector, 3MBytes,
4MBytes, 5MBytes and 6MBytes are chosen as the heap sizes for measuring the performance of each collector.

4.5.2 Collection Threshold in the Generational Collector

Two options exist for the collection threshold policy for the new generation collector in the generational collection: fixed collection threshold and fixed allocation threshold.

In the fixed collection threshold policy, a collection is invoked when a semispace of the new generation fills. The problem with this policy is that it can give rise to thrashing. For example, if copied objects fill 90% of the to space after collection, then another collection will be invoked after allocating from the remaining 10% of the semispace. If 90% of to space is filled after each of many collections, then frequent garbage collections will consume a large fraction of the CPU.

The fixed allocation threshold policy avoids the thrashing problem by invoking a collection when it is most appropriate to begin the collection. In this scheme, the collector is triggered when fixed amount of memory has been allocated. If the allocation rate of a system is constant, the collection is invoked at regular intervals. Thresholds of 128KBytes, 256KBytes, 512KBytes, and 1024KBytes are used for our experiments to test out the performance of the fixed allocation threshold policy.

4.5.3 Space Dividing in the Generational Collector

Since there are two heap regions in the generational garbage collection: the new generation space and the old generation space, the space dividing issue arises in a static garbage collection environment. In a static garbage collection environment, heap space will not be expanded during program execution.

Usually, an old generation collection is more expensive than a new generation collection. It will reduce the CPU overhead of garbage collection if less old generation
collections need to take place. There are two approaches which can make old generation collections less frequent: increasing the size of the old generation and decreasing the promotion rate. Both increasing the size of the old generation and decreasing the promotion rate extend the time that it takes to fill up the old generation, so that less collections need to be invoked.

On the other hand, increasing the size of the old generation leaves a smaller space for the new generation, assuming a fixed amount of total heap space. The decrease in the size of the new generation will increase the total number of collections that need to be done in the new generation. Even though a new generation collection is less expensive than an old generation collection, increasing the number of collections that take place in the new generation will also increase the cost of garbage collection in the new generation. So, determining the heap space division is an important factor in the generational garbage collection.

4.5.4 Promotion Policy in the Generational Collector

Policies for promotion interact closely with space dividing policies. This section discusses policies intended to prevent premature promotion, what can be done if premature promotion occurs, and how promotion policies interact with generational based collection.

A collection algorithm benefits from promoting a long-lived object as soon as possible so that the object is not copied back and forth many times in the new generation. In our implementation, an age is maintained with the objects which indicates how many garbage collections they have gone through. It is assigned to every object when the object is allocated. The objects that are allocated after a collection and before the next garbage collection have the same age, and the objects that are allocated before a garbage collection are one "year" older than the objects that are allocated after this garbage collection. To facilitate the implementation, we actually keep track of the birth-date of
objects rather than their age in our implementation. By subtracting the birth-date of an object from the current date, we can obtain the age of an object.

A copyCount is defined to control when objects are promoted. Before an object is moved during a new generation collection, the age of the object is checked. It is promoted into the old generation when its age exceeds the copyCount. Otherwise, it is moved into to space in the new generation.

4.5.5 Remember Set in the Generational Collector

A table is maintained in our implementation to keep track of the remember set. Each item in the table is a pointer to an object in the old generation that may have pointers to objects in the new generation. The pointers in the remember set table come from two sources: store operation checking and object promotion.

In our implementation, a check is performed on every store operation. We insert a pointer to an object into the remember set table if a store operation assigns a pointer to an object in the new generation to some field of an object that lives in the old generation. In addition, a pointer to each object that is about to be promoted into the old generation is added to the remember set table.

An object is removed from the remember set table if it is discovered to no longer contain any pointers to objects in the new generation. This checking happens during each garbage collection in the new generation.
Chapter 5

Performance

In this chapter, we report the performance of each collector described in Chapter 4. Firstly, we present the experimental results that are obtained from running the collectors on the Emerald system. Following that, we give our explanations of the results. Finally, we offer some conclusions.

5.1 Sample Program

For the same reasons as were explained in Chapter 3, all the performance measurements are gathered from the Emerald compiler system. In order to understand the behavior of the garbage collectors, four types of general information of the Emerald compiler program need to be considered. This section presents the detailed data about these four types of information about the Emerald compiler system.

Firstly, the average size of the live objects in the compiler system indicates how much heap space is required for the collectors to perform properly. The average size of the live objects in a program is extremely application-dependent. We measure the average size of the live objects with different heap sizes, which determine the instants in time at which garbage collections are invoked. Figure 5.14 presents the average size of the live objects of the Emerald compiler system with different heap space sizes.

Figure 5.14 demonstrates that the average size of the live objects of the Emerald compiler system do not vary very much with different heap sizes. This is due to the fact that allocation and deallocation are roughly in a steady state (i.e. equal), which means
Chapter 5. Performance

Figure 5.14: Average Live Objects of the Compiler

the net allocation rate is zero. In any long running application the net allocation rate must be zero. If the net allocation rate in a system is not zero, either the space explosion will occur and the system will crash or the heap space will go down and the system will not have any live objects after certain time.

The second system-related data is the maximum size of live objects which indicates the minimum heap size that is required for system execution with garbage collection. In order for a system to perform properly, enough heap space must be allocated to guarantee that the heap can contain all live objects at any given time during the system execution. Therefore, as much as the size of the maximum live objects must be acquired for a system to perform properly. This number varies somewhat from garbage collector to garbage collector, but during our experiments the maximum required by any collector was 3.2Mbytes.
Another system-dependent piece of information we want to address is the total bytes allocated during the compiler system execution. This total allocation bytes, together with some garbage collection related parameters, determines the number of garbage collections that need to take place. In the Emerald compiler system, 40MBytes in total are allocated.

Finally, the total system execution time is another important parameter in understanding the performance of the garbage collectors. The total execution time of 360s is measured for the Emerald compiler system.

5.2 Performance of the Collectors

5.2.1 Mark-and-Sweep Collector

As indicated in Figure 5.14, the average bytes of live objects at any given time of Emerald compiler system is roughly 2MBytes. Therefore, in theory, the program should be able to execute in a 3MByte heap space. In order to observe the fine difference of the performance between each collector, heap spaces of 3MBytes, 4MBytes, 5MBytes, and 6MBytes are chosen for the mark-and-sweep collector. Figure 5.15 illustrates the overhead of the mark-and-sweep collector with different heap spaces.

Looking at the CPU overhead of the mark-and-sweep collector, the behavior of the system becomes apparent. The space expands to 6MBytes with heap spaces of 3MBytes, 4MBytes, and 5MBytes and to 7MBytes with a heap space of 6MBytes. There are two reasons that cause this space expansion in the mark-and-sweep collector. The first one is that our mark-and-sweep collector is a conservative collector which does not guarantee to free all the space that is occupied by garbage objects after a garbage collection. This, in turn, causes the system to use more space than is needed. Secondly, the mark-and-sweep collector does not perform compaction. Consequently, the space gets fragmented after several garbage collections. When a large memory allocation request that can not be
satisfied occurs the system must require more space from OS even there are many small pieces of memory fragmented in the free list.

As also indicated in Figure 5.15, the overhead shows a decrease with a bigger initial heap space even the collector ends up with using the same amount of memory (6MBytes). With small heap sizes, the space is filled up quickly, and garbage collection is therefore invoked frequently. Frequent garbage collection makes the CPU overhead fairly high. In addition, the time that takes to fill up the heap space is less with a small heap space compared to a large heap space, so that less objects die during the time before next garbage collection is invoked. Accordingly, the free space is less for the allocation before the garbage collection for the next round, which makes the next garbage collection happen soon. This phenomenon accelerates the rate at which garbage collection takes place.

From Figure 5.15, we can conclude that the mark-and-sweep collector must live in a
heap with three times as much space as the average size of the live objects. And, with three times as much initial heap space as the system needed (6MBytes), good performance can be obtained.

5.2.2 Simple Copying Collector

We measure the same heap sizes: 3MBytes, 4MBytes, 5MBytes and 6MBytes for the simple copying collector. However, rather than having only one collection space, the heap space is divided into two parts. One part of the space contains only concrete type objects which are part of the roots for garbage collection, so that this space will never be collected. The other part, so-called collection space is the space from which the objects are allocated and where garbage collection happens. In the Emerald compiler system context, concrete type objects take up 1.5MBytes of heap space, leaving 1.5MBytes, 2.5MBytes, 3.5MBytes, and 4.5MBytes heap spaces for the collection space with total heap sizes of 3MBytes, 4MBytes, 5MBytes, and 6MBytes respectively.

Figure 5.16 presents the CPU overhead of the simple copying collector with different heap spaces. From Figure 5.16, we can observe that the program can not perform with total heap spaces of 3MBytes and 4MBytes. This observation can be explained by looking at the available heap space in allocation space and the maximum size of the live objects in the system. An allocation space is defined to contain both the space for concrete type objects and the half of the collection space. As explained previously, our new generation collector is a copying based, two-semispaces collector, therefore, with heap spaces of 3MBytes and 4MBytes, each semispace is of size of 0.75MBytes and 1.25MBytes respectively. Adding in the 1.5MBytes free space for concrete type objects, the total available allocation spaces are 2.25MBytes and 2.75MBytes. Although the average size of the live objects in the system is 2MBytes, the maximum is somewhat larger. The garbage collection fails at the first instant where the size of the live objects in the system
Figure 5.16: CPU Overhead of Simple Copying Collector

exceeds 2.25MBytes for a heap space of 3MBytes and 2.75MBytes for a heap space of 4MBytes.

Figure 5.16 also shows very high CPU overhead with a heap space of 5MBytes. With a heap space of 5MBytes (1.75M semispaces for allocation), a large number of garbage collections happens. This can be explained by looking at the size of the semispaces, total allocation bytes, and the size of the average live objects. With 1.75MBytes semispaces for object allocation, 1.5MBytes for concrete type objects, and 2MBytes average live objects (including concrete type objects), there is 0.8MBytes free space for allocation between two successive garbage collections. By the fact that the total amount of allocation of the Emerald compiler system is roughly 40MBytes, there are 76 collections in total, which, not surprisingly, causes high CPU overhead. Better performance is obtained with a bigger heap space as expected due to the small number of collections that are needed.
5.2.3 Copying Compacting Collector

There is only one heap space in the copying compacting collection. Similarly, the heap spaces are set to be 3MBytes, 4MBytes, 5MBytes, and 6MBytes. Since concrete type objects are usually allocated at the beginning of the program execution, the head part of the heap space is occupied by the concrete type objects and these objects are not moved at all during the garbage collection by our algorithm.

Figure 5.17 demonstrates the performance of the copying compacting collector. Unlike the other collectors discussed so far, Figure 5.17 shows that the copying compacting collector is capable of living in a small heap space (as small as 3MBytes). This is possible because there is only one allocation space in the copying compacting collector. It allows the system to allocate as many objects as possible until the heap space boundary is hit, at which time a collection is invoked. Since the average size of the live objects in the Emerald compiler system is roughly 2MBytes, a 3MBytes heap space provides an average of 1MBytes free space for allocation after each collection.

Figure 5.17 also indicates that, with heap space of 4MBytes, the CPU overhead is approximately half of that obtained with a heap space of 3MBytes. In fact, there are two factors that determine the cost of the copying compacting collection: the number of collections that are needed and the number of live objects that are moved in each collection. From our experiments, we observe that the number of objects that are moved at each collection does not change very much with different heap spaces, and therefore, the number of garbage collections dominants the total garbage collection cost. Due to the fact that there are roughly 2MBytes average live objects, with a heap space of 3MBytes, there is 1MBytes free space after each collection, while, the free space is 2MBytes with a heap space of 4MBytes. Therefore, twice as many as collections are needed with 3MBytes heap space, which makes its CPU overhead twice as high.
5.2.4 Generational Collector

Since the new generation collector is a two-semispaces one, unlike the copying compacting collector, the smallest heap space that the generational collector can perform correctly is 4MBytes. The generational collector has many parameters that can be used to tune its performance. First is the total size of the heap, which, like the other collectors, includes 3MBytes, 4MBytes, 5MBytes, and 6MBytes. Second is the copyCount, which controls how quickly objects are promoted from the new to the old generation. In our experiments, copyCounts of 1, 2, 3, 4, 5 are measured. Third is the ratio of the heap size of the new generation and the old generation. Finally, we can choose either the fixed collection threshold or the fixed allocation threshold scheme to trigger garbage collections.

Comparing the performance figures obtained using the fixed collection threshold policy to the fixed allocation threshold policy, we observe, in contrast to the argument
Chapter 5. Performance

Figure 5.18: CPU Overhead of the Generational Collector

presented in [Zor89], the fixed collection threshold policy offers better performance. In [Zor89], he claims that thrashing in the new generation would occur when using the fixed collection threshold policy, which causes a high garbage collection cost. From our experiments, we observe that there is certain amount of free space in the new generation for allocation after each collection due to the promotion of objects to the old generation. Therefore, the thrashing is in fact avoided in the fixed collection threshold in our implementation.

Looking at the performance figures (Figure 5.19, Figure 5.20, and Figure 5.21) obtained for our generational collector with different sizes of the old generation, unlike what is claimed in Zorn’s work [Zor89], the best performance is obtained with a fairly small old generation space (2MBytes). Figure 5.18 presents the CPU overhead with 2MBytes as the old generation space at heap spaces of 4MBytes, 5MBytes, and 6MBytes. Larger
heap space (larger new generation space) shows better performance as expected due to the smaller number of collections.

Figure 5.18 also indicates that the copy count is not a major factor in determining the cost of the generational collection. Furthermore, the generational collector can not perform correctly with copyCount of 1 with heap spaces of 4MBytes, or 5MBytes due to running out of space in the old generation. Since a static allocation scheme is used in our generational garbage collector, the heap space is not allowed to be expanded during the system execution. The system crashes when there is not enough free space in the old generation after an old generation collection for the large number of objects that are about to be promoted into it with a copyCount of 1.

Another conclusion we can draw is that the old generation collection cost makes a big portion of the total cost of the generational garbage collection (higher than the new
5. Performance

5.2.5 Performance Comparison of the Collectors

Figure 5.22, Figure 5.23, and Figure 5.24 illustrate the performance of each collector with heap spaces of 4MBytes, 5MBytes, and 6MBytes respectively. These figures indicate that the generational collector outperforms the other collectors. However, the two assumptions that have been made for generational collection to be able to obtain a good performance do not appear to be the real reasons in our experiments.

From the experiments, we observe that the total number of garbage collections is the key factor that determines the garbage collection cost. While the generational collection frees the space that is occupied by garbage objects, it also promotes objects into the
Figure 5.21: CPU Overhead of the Generational Collector (6MBytes Heap Space)

Figure 5.22: CPU Overhead of All the Collector with 4M Heap Space
old generation, which in turn guarantees a certain amount of available memory in the new generation after each collection. Frequent garbage collection is effectively avoided and therefore good performance is obtained — even though the lifetime distribution of Emerald objects does not match that expected to give good generational garbage collection performance and that store operations are frequent.
Figure 5.24: CPU Overhead of All the Collectors with 6M Heap Space
Chapter 6

Conclusion

This thesis studies the behavior of an object-oriented programming language and how it affects automatic storage reclamation - garbage collection. Two important techniques: simulation and measurement of implementation on actual garbage collectors are investigated by using Emerald - an object-oriented programming language as an example language.

In doing the simulations, firstly, several parameters that affect storage reclamation are analized and measured and then reported. The simulation provides us with a cost model which allows us to estimate the performance of generational garbage collection in Emerald.

The second part of the thesis involves the implementation of a real generational garbage collector for Emerald. Implementation reports prove our simulation results, and allow us to compare the performance of several different garbage collection algorithms, which helps us better understand garbage collection in an object-oriented programming paradigm.

6.1 Simulation

Our simulation of the generational garbage collection in Emerald is based on the mark-and-sweep garbage collector which was implemented as part of the original Emerald system. In order to measure the parameters that are specified in the generational garbage collection algorithms (collection threshold, remember set, copy count, etc) the original
Chapter 6. Conclusion

collector was modified to record the birth time of each object when it is created.

Since the mark-and-sweep collector that we used to generate our simulation results is a conservative garbage collector, which does not guarantee to collect all the garbage a system produces in a garbage collection, the simulation results may not be completely accurate. Nevertheless, the simulation provides us valuable data to guide the real implementation done in the second part of work of the thesis. The simulation also indicates that Emerald's behavior with respect to garbage collection is somewhat different from that in LISP.

A difficulty with simulations is that they can produce a large amount of data; it is easy to get lost in the sea of data collection. This thesis therefore abstracts the system parameters that affect garbage collection performance into a cost analysis model.

6.2 Generational Garbage Collection and its Cost Analysis

The first part of the thesis has focussed on the performance analysis of the generational garbage collection. Generational garbage collection has been proved an efficient garbage collection algorithm in many functional languages, such as, LISP, ML, FL, etc for two reasons: the vast majority of objects distributed at a very young age and store operations are quite rare.

In this thesis, cost estimates indicate that the garbage collection cost of generational algorithm is higher in the Emerald system than in functional systems due to the violation of these two assumptions that are observed in functional languages (high young object death rate and low frequency of store operations). Our simulation observes 53% average death object rate at age 1 with a 512Kbytes collection threshold. This indicates that almost half of the objects that are created between two garbage collections are still alive after the first garbage collection.
Chapter 6. Conclusion

Our simulation shows that those live objects that survive their first garbage collection are likely to live for a fairly long time. Only a few objects which have experienced their first garbage collection die shortly after. This object lifetime distribution indicates that a big performance gain can be obtained by promoting the objects that have experienced one garbage collection into the older generation to avoid copying them back and forth in the new generation.

The number of store operations that assign a pointer from the old into the new generation is larger than in functional languages, which causes a higher cost of store operations in Emerald (about 10 times higher than in Zorn’s LISP program).

6.3 Implementation

In the implementation part of the thesis, a generational garbage collector is implemented for Emerald. We can create two other collectors by turning the generational one appropriately. We obtain a simple copying collector by setting copy count to infinite so that the old generation collector does not need to be called at all during the garbage collection. Similarly, by setting the size of the new generation to zero and allocating objects from the old generation we generate our copying compacting garbage collector. The mark-and-sweep collector we used to measure the performance is the one that we used to produce all simulation results.

After having all the garbage collectors, we measured their real CPU cost of garbage collection for the Emerald compiler program with different collection thresholds. Since any garbage collector can be made efficient by using large amount of heap space, we can not observe the difference between each collector with a very large heap space. Therefore, we measured the performance of each collector with a set of reasonably small heap spaces: 3Mbytes, 4Mbytes, 5Mbytes, and 6Mbytes.
From the implementation results, we observe that the generational garbage collection slightly outperforms the mark-and-sweep, the simple copying, and the copying compacting garbage collection algorithms. Promotion has been proved to be the cause of the better performance of the generational algorithm. In the single space copying based garbage collectors, after the available space goes down to a certain value, garbage collection occurs frequently, which results in a high garbage collection cost. In a generational collector, a certain number of objects are promoted into the old generation at each collection which guarantees a fairly large available space in the new generation after each garbage collection, which in turn avoids frequent garbage collections.

In contrast to the arguments given in [Zor89], our implementation shows that generational garbage collection obtains better performance with a small old generation space than with a big old generation space. This is due first to the high cost of the old generation collection in a large old generation space, and second to the frequent garbage collection of the new generation which is caused by the small new generation space.

6.4 Future Work

This thesis considered the generational garbage collection with at most two generations. Our simulation results, notably the rather even distribution of the object life time, indicates that a generational collector having more than two generations may achieve significant performance improvement.

And also, in our old generation collector, the allocation strategy could be improved. In this thesis, the space for the old generation is all allocated at the beginning of program execution and it is not allowed to expand during a program’s execution. This results in system failure when there is insufficient space in the old generation after a garbage collection for the objects that are to be promoted into it. A dynamic memory allocation
strategy is needed to avoid the failure, in which a system starts with a certain amount of memory in the old generation and expands that memory (and possibly contracts when unwanted).
Bibliography


