OF / DE

METED

**National Library
of Canada**

**Bibliothèque nationale
du Canada**

Canadian Theses Service

Service des thèses canadiennes

**Ottawa, Canada
K1A 0N4**

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Creepy: An Incremental Secondary Storage Garbage Collector

by

Thong T. Nguyen

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment of
the requirements for the degree of
Master of Computer Science
School of Computer Science

Carleton University
Ottawa, Ontario
December 21, 1989

© Copyright
1989, Thong Nguyen

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-59020-3

The undersigned recommend to
The Faculty of Graduate Studies and Research
acceptance of the thesis
Creepy: An Incremental Secondary Storage Garbage Collection
submitted by
Thong Nguyen
in partial fulfilment of the requirements for
the degree of Master of Computer Science

Thesis Supervisor

School of Computer Science

Carleton University
December 21, 1989

Abstract

This thesis presents a new incremental method for garbage collecting objects in secondary storage. A garbage collector, called Creepy, can incrementally reclaim and compact objects in a persistent object system. Creepy operates in two modes: on-line and off-line. In the on-line mode, Creepy examines pages in main memory locating inaccessible objects and reclaiming their space. In the off-line mode, Creepy examines a window of pages, resolving forwarding pointers, compacting storage to increase locality and reclaiming circular structures existing in this window of pages. A prototype of the Creepy garbage collector has been implemented for a Sticky system to demonstrate the simplicity and the efficiency of the algorithm.

Acknowledgements

First, I would like to thank my two advisors Ekow Otoo and Dave Thomas for their guidance, patience and understanding during the development of this thesis. Most of all I would like to thank Dave Thomas for his understanding and invaluable advice that helped me overcome my problems and complete this thesis.

This research was supported by NSERC, Object Technology International (OTI) and the Rockwell International Science Center (RISC). In particular, I would like to thank Dr. David Morley at RISC, John Duimovich and Greg Adams at OTI for their technical advice. Greg Adams, even in his busiest time, was always available for discussion.

I would also like to thank all of my proof readers, Janice Duffney and Tina Groves for their organizational and proofreading skills. My friends and colleagues, Brian Fultz, John Glenn, Joe May and Jeff McAffer have shared with me the joys and frustrations of student life.

My family has been an important source of encouragement during my years. In particular, I wish to convey my deepest appreciation to my aunt Thanh Nguyen. Her confidence and generosity have always helped me realize my goals.

Contents

1	Introduction	1
1.1	Generic garbage collectors	2
1.2	Garbage collection requirements	3
1.3	Garbage collection advantages and disadvantages	4
1.4	Garbage collection reliability	6
1.5	Differences between main memory and secondary storage collectors	6
1.6	Motivation	8
1.7	Related Work	9
1.8	Thesis Overview	10
2	Main memory garbage collection	11
2.1	Garbage collector overview	11
2.2	Mark/sweep	13
2.3	Reference counts	14
2.4	Deferred reference counts	16
2.5	Copying/moving collectors	17
2.5.1	Baker's real time semi-space	18
2.5.2	Lieberman-Hewitt algorithm	20
2.5.3	Ungar Generation Scavenger	22
2.6	Secondary storage compaction	24

2.6.1	Disk I/O	24
2.6.2	Object clustering	26
3	Secondary Storage Garbage Collectors	28
3.1	Bjornerstedt system	28
3.2	Campin's Garbo	32
3.3	ORSLA garbage collector	34
3.3.1	Area	35
3.3.2	Inter-area links	35
3.3.3	Cables	37
3.3.4	Area header	37
3.3.5	ORSLA garbage collector	38
3.3.6	ORSLA contribution	38
3.4	Summary	39
4	Creepy	41
4.1	Overview	41
4.2	Creepy algorithm	43
4.2.1	On-line operation	45
4.2.2	Off-line operation	48
4.3	Reference count maintenance	49
4.3.1	Loading	51
4.3.2	Unloading	53
4.4	Correctness of the Creepy algorithm	57
4.4.1	Inductive proof	57
4.5	Large RefCountTable	58
4.6	Advantages and disadvantages	59
4.7	Multi-user secondary storage system	61

5	Prototype Implementation	63
5.1	Hardware Configuration	63
5.2	Sticky	64
5.2.1	Transaction	65
5.2.2	Sticky Objects	66
5.2.3	Repository	67
5.3	Restrictions and Limitations	69
5.4	Implementation Problems	70
6	Conclusion	71
6.1	Future work	72
6.2	Summary	73
6.3	Postscript	73
A	Glossary	78

List of Figures

1.1	An illustration of a common mistake with hand-coded memory management.	5
1.2	An example of reference count updates.	7
2.1	An illustration of Baker's algorithm.	19
2.2	An illustration of a forward chain in Lieberman and Hewitt's algorithm.	22
3.1	An illustration of Bjornerstedt's algorithm.	31
3.2	ORSLA storage organization.	36
4.1	A snapshot of a running system.	44
4.2	Creepy algorithm in on-line mode.	46
4.3	An example of Creepy working in on-line mode.	47
4.4	Creepy algorithm in off-line mode.	48
4.5	An example of Creepy working in off-line mode.	50
4.6	Load algorithm.	51
4.7	An example of object loading.	52
4.8	Unload algorithm.	54
4.9	An example of object unloading.	55
4.10	An example of negative value in the RefCountTable.	56

Chapter 1

Introduction

Most persistent object systems provide either no garbage collector or a very inefficient off-line one. Many researchers in the object-oriented and database community have questioned the feasibility of a persistent object garbage collector. This thesis solves the garbage collecting problem by presenting Creepy¹, an incremental secondary storage garbage collector which operates efficiently in near real time. This algorithm is simple and straightforward to implement as a reliable component of a persistent object system.

The avoidance of disk I/O operations is an important aspect in garbage collecting secondary storage. Creepy minimizes disk I/O operations by reclaiming all unused space in a single disk scan.

The overhead involved with existing garbage collection algorithms [6] [9] is very high. Creepy's unique method for maintaining the information necessary for garbage collection has virtually no overhead when an object resides in memory and very low overhead otherwise.

The reference count method utilized to implement the garbage collection algorithm has traditionally been overlooked due to its inability to reclaim circular

¹The name Creepy is chosen since the garbage collector is constantly "creeping" the secondary storage, reclaiming unused space much like the popular "creepy-crawler" swimming pool cleaner.

structures. However, this obstacle is overcome by providing the ability to reclaim these structures during the off-line operation.

In addition, Creepy compacts secondary storage to reduce storage fragmentation and enhance object clustering. Creepy accomplishes the aforementioned tasks without compromising the execution time of currently active applications.

1.1 Generic garbage collectors

A garbage collector in languages such as Lisp, Scheme and Smalltalk provides a mechanism to automatically reclaim storage of unreachable objects. In these languages, attention to details of allocation and explicit deletion of storage are no longer necessary.

A difficult aspect of garbage collection is to determine which objects are inaccessible. This can only be achieved by forming a *root set* consisting of immediately accessible objects in the current computational frame. Any object reachable from this set is accessible; otherwise, it is inaccessible by definition. Inaccessible objects can be removed and their storage reclaimed.

Most existing garbage collection algorithms can be summarized in three steps:

1. Trace through all accessible objects, thus locating those which are inaccessible.
2. Reclaim the storage occupied by the inaccessible objects.
3. Compact storage and update object pointers to reflect the object movements during the compaction process.

As a bare minimum, all garbage collectors must perform steps 1 and 2. Step 1 can be time consuming if the entire memory must be scanned. In the infancy of garbage collection, much research had been focussed on reducing the amount of work involved in this step. Step 2 is the easiest of the three steps, and simply consists of returning storage of inaccessible objects to the free memory list as discussed by Lang and Dupont [31]. Step 3 is done as part of step 1 in some algorithms [3] [33] [21]

[41]. Otherwise, this step is very difficult to perform since the compacting problem is NP-complete². Consequently, compaction is usually achieved by a brute-force technique which slides objects to one end of storage, thereby eliminating the storage fragmentation. This technique is time consuming since the entire memory must be scanned to update the referents of moved objects.

1.2 Garbage collection requirements

Garbage collection³ of inaccessible objects is only possible if the accessibility of an object can be determined without the programmer's assistance. An object is accessed via an object reference. This reference can be in the form of machine address pointers, as in Smalltalk, or of *persistent identifiers*, as in persistent object servers [7] [23] [24], or "symbolic" names [10] as in relational database.

In a system where object references can be constructed, the entire storage space is always addressable; hence, all objects are accessible. Programmer assistance is required to determine whether an object is no longer in use. Therefore, garbage collection in this system is not possible.

On the other hand, if object references cannot be constructed, an object without a reference is inaccessible. Creation and assignment of the object references must be performed by a mechanism hidden from the programmer. Furthermore, operations on an object are permissible only if its reference is available. Therefore, if all of the references to an object have either been destroyed or stored in other inaccessible objects, this object can never be used again. In such a system, it is impossible to reuse the storage associated with an inaccessible object since no operation can be performed on the object, including delete and reclaim. Thus, the physical storage used by inaccessible objects is wasted. Without a garbage collector, the programmer is forced to ensure that no objects ever become inaccessible by releasing the storage of all objects before they become inaccessible.

²See proof of Theorem 1 in Chapter 2.

³Garbage collection implies automatic reclamation unless stated otherwise.

1.3 Garbage collection advantages and disadvantages

There are two opposing memory management philosophies arising from the procedural and symbolic communities [46]. The procedural community argues that memory management is so important that it must be left to the programmers. The symbolic community argues, on the other hand, that memory management is so important that it cannot be left to the programmers.

The procedural community believes that “real” programmers manage their own memory. Handcrafted memory management can achieve better memory efficiency. Garbage collection is, therefore, considered unnecessary. Furthermore, garbage collection causes too many interruptions, hindering the performance of a running application.

However, garbage collection makes the job of programming much easier by simplifying an application design. For example, up to thirty percent of the code written in a language without a garbage collector is concerned with keeping track of shared data structures, releasing them when all references are gone, and reusing the space for new allocation [29].

Garbage collection removes the error-prone task of allocating and deallocating memory. In a procedural language such as C, mistakes are often made in managing memory. C programmers often forget to allocate memory before assigning a value; or release memory prematurely or never release unused memory. These mistakes illustrated in Figure 1.1, lead to clobbering memory space and causing abnormal behaviour in an application or in consumption of all the available space and causing a memory shortage.

Moreover, in languages such as Smalltalk, Scheme and Lisp, tens of thousands of objects are being created and destroyed per second [41]. Programmers are more apt to make mistakes when managing objects at this rate. Apart from just reclaiming used memory, garbage collection can improve the performance of an application by reducing memory fragmentation and increasing the locality of data references.

```

struct Node {
    int value;
    struct Node *left, *right};

main() {
    struct Node *aNode;
    aNode->value = 5};

replaceLeftNode (aNode, newNode)
    struct Node *aNode, *newNode; {
/*****
 *   Programmers usually forget to free the space of      *
 *   the left node, i.e., free (aNode->left);              *
 *****/
    aNode->left = newNode};

```

Figure 1.1: An illustration of a common mistake with hand-coded memory management.

These tasks are often overlooked when the programmers manage their own storage.

Garbage collection techniques have improved substantially. The concern that garbage collection is too expensive is no longer accurate. The Generation Scavenger [41], for example, reduces pause time to a fraction of a second, virtually imperceptible to the users. A visual cue is usually provided to show that the garbage collector is at work. Even then, the change is often so fast that only a flash is perceived.

In summary, the advantages of a garbage collector clearly outweigh its disadvantages. A garbage collector can reduce code size and eliminate errors in memory management. Furthermore, the increase in locality can enhance the performance of the user applications.

1.4 Garbage collection reliability

Given the importance of garbage collection, a reliable implementation is absolutely essential. This requirement dictates the need for a straightforward and easy to implement algorithm.

A simple garbage collection algorithm is preferable over a complex one, even if the more complex one yields better performance. Complicated algorithms not only take a long time to implement, but are also error prone. The test suite for a complex algorithm is also difficult to devise leaving many aspects in the algorithm untested. Errors introduced by garbage collectors cause running applications to fail occasionally at random locations. Reproduction of the problem is usually not achievable since it is difficult to recreate the same environment. As a consequence, complicated garbage collection algorithms [26] are typically not implemented and are usually used as a measure by which to compare those implemented.

1.5 Differences between main memory and secondary storage collectors

There are many garbage collection algorithms designed for main memory which reclaim the space occupied by inaccessible objects. The following paragraphs show the differences between main memory and secondary storage garbage collectors, highlighting the reasons they are unsuitable for secondary storage.

The primary function of main memory garbage collectors is to reclaim unused space. They are often forced to execute when the storage is exhausted. Since there are several orders of magnitude difference in the capacity of secondary storage and main memory, it is unlikely that secondary storage will be exhausted. If a storage request cannot be satisfied by secondary storage, it is likely a result of fragmentation. In addition, fragmentation leads to poor storage locality, causing a deterioration in system performance due to excessive disk I/O. Consequently, the criteria for garbage collection cannot be dictated by the space conditions alone; degradations of system

Figure 1.2: An example of reference count updates.

performance due to storage fragmentation must also be considered.

Compaction in main memory garbage collection is a process which packs objects together so that memory space is not wasted by fragmentation. In secondary storage, objects tend to be dispersed throughout, resulting in unnecessary amounts of disk I/O. Compaction of secondary storage minimizes disk fragmentation and also localizes objects so that those objects which are accessed together are physically stored together.

The access time of secondary storage is variable and much slower than that of main memory. The sectors or tracks closest to the disk arm require less time to access than those further away. Many of the algorithms developed for main memory are hopelessly inefficient when applied to disk due to thrashing. For example, in a reference count system, changing an object pointer requires three object updates. Figure 1.2 shows that when A changes its reference pointer from B to C, A's pointer as well as the reference counts of B and C must be updated. Potentially three disk seeks may be required, making this method inappropriate for secondary storage.

The very large address space of secondary storage is problematic to garbage

collection. If paging activity is ignored, the time required for garbage collection is directly proportional to the amount of accessible storage. For example, the Kodak optical disk system: 6800 can store a trillion (10^{12}) bytes of information [28]. Therefore, two weeks of operation would be required to access the entire data store at the rate of 10^6 words per second. This calculation does not include update time for objects modified since the last garbage collection and compaction. Moreover, the address space of secondary storage can expand to 10^{30} words, such storage would require three years of operation to access all objects using the same access rate.

The size of secondary storage shows the necessity for incremental garbage collection on a small segment of secondary storage. Any segment to be garbage collected must contain all external references and any other information required by the collector within the segment. Without this information, the entire address space may need to be scanned in order to determine the accessibility of objects. None of the main memory garbage collectors have this property, and are therefore too slow to use for secondary storage.

1.6 Motivation

This thesis is motivated by the Sticky project currently ongoing at Carleton University. Sticky is a distributed persistent object server which manages long-term objects by storing them on a secondary storage rather than as a snapshot of memory such as a Smalltalk image. In so doing, Sticky liberates objects from their virtual memory environment by allowing them to be distributed over a local or a wide area network.

Sticky objects can move freely between main memory and secondary storage using the load and unload operations. Unlike many existing object servers, where objects are grouped into segments of large uninterpreted strings of bytes, Sticky objects have a common representation in memory as well as in secondary storage. Hence, Sticky does not require an expensive translation when objects move between main memory and secondary storage. Each Sticky object is associated with a unique 32 bit persistent object identifier. The memory address of each machine in the

network is also a linear address space of 2^{32} bytes; the mapping of objects between secondary storage and memory does not require a look-up table.

Sticky seeks to eliminate the sharp boundary that currently exists between application programs, operating systems and secondary storage. However, since Smalltalk has no notion of versions, clustering or transactions, the introduction of such concepts in Sticky creates a boundary. To minimize the existing boundary, Sticky makes the object transition reasonably transparent to the application programmer by treating all objects (both Sticky and non-Sticky) in the same manner and eliminates the need for database query language.

Objects in Sticky are manipulated through a transaction as described by Stringham [38]. When a referenced object is not in memory, the current transaction faults the object into memory from disk. At commit time, changed and new objects are written to secondary storage as part of the transaction operation. The transaction manager handles the concurrency and recovery issues.

In Sticky, a garbage collector subsumes the traditional maintenance tasks of file system reorganization and restructuring. In order to achieve this, Sticky requires an incremental secondary storage garbage collector. Creepy has been designed to satisfy this requirement.

1.7 Related Work

Many garbage collectors designed for main memory and virtual memory are unsuitable for secondary storage due to their excessive use of disk I/O. The garbage collectors of Campin [9] and Bishop [6] are, however, tailored for secondary storage. Both Bishop and Campin demonstrate that in order for a garbage collector to work for secondary storage, it must have two fundamental features. First, the garbage collector must work incrementally to avoid shutting down the system for a long period (in the order of days). Secondly, it must have all the information required in a garbage collected area to prevent unnecessary disk scanning. Bishop divides disk space into areas, where each area maintains a list of inter-area links which allows an

area to be garbage collected separately from the whole system. Campin divides the disk space into databases and maintains a cross-reference table in each of them, enabling a database to be garbage collected piece by piece without scanning the entire disk space. The problem arises from the high cost of maintaining inter-area links and cross-references. Bishop's algorithm requires special virtual memory hardware to trap for inter-area links. The Campin algorithm requires all databases to lock while the cross-references are built.

Creepy incorporates the above features by dividing the disk space into pages and garbage collects these pages independently. The unique method of maintaining the page cross-references distinguishes Creepy from the techniques of Bishop and Campin. Creepy maintains reference counts of cross-references in a page. These reference counts are updated when objects move between secondary storage and main memory. Thus, there is no overhead for maintaining reference counts while objects are being manipulated in main memory.

1.8 Thesis Overview

This thesis is organized into an additional five chapters. Chapter 2 presents a summary of the existing main memory garbage collection techniques. The advantages and disadvantages of each technique are examined. This chapter also includes a proof showing that the storage compaction problem is NP-complete. Chapter 3 presents an in-depth study of existing secondary storage garbage collectors. The discussion provides a foundation for the design of Creepy which is described in detail in chapter 4. The implementation of a Creepy prototype is described in chapter 5 to illustrate the simplicity of the algorithm and to suggest further research. Restrictions and limitations of the prototype are also discussed. The last chapter summarizes the major contributions of this thesis and suggests further work.

Chapter 2

Main memory garbage collection

The development of garbage collectors began in the late 1950s when many artificial intelligence (AI) researchers were investigating the possibility of memory management by the system instead of programmers. This chapter provides a brief summary of three fundamental techniques for garbage collecting in main memory. In particular, the mark/sweep, reference count and copying algorithms are examined to study their advantages as well as disadvantages. In addition, this chapter provides a proof to show that the compaction problem is NP-complete.

2.1 Garbage collector overview

Before garbage collectors were invented, programmers were responsible for managing the allocation and deallocation of memory. This was adequate for systems with small memory size, but the task of memory management became a burden on programmers as the memory size became much larger. Many AI researchers realized this problem in the late 1950s and began investigating alternative memory management methods. The concept of garbage collection resulted.

Garbage collectors were originally designed to reclaim storage occupied by inaccessible objects. However, they have evolved greatly since their initial development. Their role has been extended to include such tasks as reducing storage fragmentation as well as increasing locality by aggregating related objects.

A variety of algorithms have been developed, many of which have been summa-

alized by Cohen [13] who surveyed the work done before 1981. Recently, Duimovich's dissertation [19] reviewed many techniques developed between 1981 and 1988.

McCarthy [34] published the first garbage collector called *mark/sweep* in early 1960. In the same period, Collins [14] invented a reference count technique for garbage collecting. These two techniques, even though they have been shown to be inefficient by Ungar [41] and are not widely used any more, still provide a building block for many new garbage collection algorithms, including Creepy. In 1978, Baker [3] published an incremental copying garbage collector which was derived from Fenichel and Yochelson's algorithm [21]. The Baker garbage collector is currently one of the most widely-used collectors due to its simplicity and efficiency. Baker's algorithm, along with the mark/sweep and reference count techniques, are considered classical examples of uni-processor main memory garbage collectors.

In the 1960's, memory size was small. Thus, many variations of the mark/sweep algorithm were invented to minimize the use of new space during garbage collection. Knuth [26] presents the most comprehensive discussion for these techniques and remains a standard reference for many algorithms before the seventies. But the invention of virtual memory in the late sixties had an impact on the design of garbage collection algorithms in two significant areas. First, the use of additional storage for a stack was now feasible, since virtual memory was unlimited. Avoiding page faults and thrashing, on the other hand, becomes a critical factor in improving the efficiency of garbage collection. Garbage collectors can improve placement for locality since objects are moved into the virtual address space while reclaiming storage. An application can greatly benefit from garbage collectors as they cluster a working set of objects in memory, reducing the occurrence of page faults. Second, garbage collectors in a virtual memory environment are usually invoked whenever page faulting significantly degrades performance of an application [21].

2.2 Mark/sweep

Mark/sweep [34] is one of the classical garbage collection algorithms. It gained much of its popularity due to its simplicity. Mark/sweep consists of three phases: the first phase, the mark phase, performs an exhaustive search starting from the root set and marks all objects reachable from this set. The sweep phase follows, linearly scanning the whole memory unmarking all marked objects and returning unmarked objects¹ to the free memory pool. As memory is reclaimed on a per object basis, memory fragmentation can occur. When enough memory is available but is unable to satisfy a memory request due to memory fragmentation, the third phase, compaction, begins. This phase slides all accessible objects to one end of storage and inaccessible ones to the other end. A separate memory sweep is required to update addresses and copy the objects when compacting memory.

One advantage of mark/sweep is its simplicity. A more important advantage is that mark/sweep examines every object in memory which allows it to reclaim the storage of all unreachable objects. These two advantages highlight reasons for its popularity in the early 1960s.

The major disadvantage of mark/sweep is its inefficiency. The execution cost is proportional to the size of available memory, rather than to the accessible objects (as in the case of copying collectors). The overhead for garbage collecting becomes excessive when only a small part of memory is reachable. Ungar [41] showed that mark/sweep can be very inefficient for some Lisp programs which spend as much as 25% to 40% of their time garbage collecting.

Another drawback associated with mark/sweep collectors is memory fragmentation. Memory is partitioned into small chunks making it difficult to satisfy a memory request. Furthermore, to perform memory compaction efficiently is an NP-complete problem². Furthermore, memory compaction requires an additional memory scan to update pointers of relocated objects.

¹These objects are not reachable; otherwise they would have been marked by the first phase

²See proof of Theorem 1 in Chapter 2.

Mark/sweep collectors perform poorly in a virtual memory system since every accessible object must be inspected during the mark phase thereby causing unnecessary paging activity. In addition, the sweep phase requires a complete scan of the whole virtual space to find unreachable objects, thus forcing more page swapping.

Mark/sweep collectors require additional memory for marking bits and a tracing stack to locate live objects. Knuth [26] suggested various methods to minimize the space required for garbage collection. One such technique is the reverse pointer technique which temporarily destroys an object structure in the mark phase and restores it in the sweep phase. However, these algorithms are complicated and require a considerable amount of execution time to maintain the additional information.

2.3 Reference counts

At the same time when the mark/sweep technique was published by McCarthy, Collins [14] published a different approach to garbage collection using reference counts. In a reference counting system, every object contains an extra field (the reference count) representing the number of objects pointing to it. The reference count field of an object, A, is updated each time a pointer to A is created or destroyed. The reference count is incremented by one when a pointer to A is created. Similarly, when a pointer to A is destroyed, the reference count of A is decremented by one. As illustrated in Figure 1.2, when A changes its reference pointer from B to C, B's reference count is decremented by one while C's is incremented by one.

When the reference count becomes zero, A can be reclaimed since it is not reachable from any objects in the system. Before A's space can be reclaimed, the reference counts of its immediate reference objects are decremented by one, and are also reclaimed if zero. This recursive process continues until all zero reference count objects are reclaimed.

Due to the recursive freeing of objects, a processor can waste a large amount of time reclaiming unused objects. However, if a freed object pool is kept instead of a free memory pool, the newly freed object can be added to this pool without updating

its immediate references. When the space of a freed object is reused, the object is removed from the freed object pool, and the reference count of its immediate objects are decremented. If the reference count of these objects is zero, they are added into the pool. In this way, the work required to reclaim objects is bounded by each allocation.

In theory, the reference count value may be as large as the number of objects in memory. However, in practice, an object usually has a small set of references. An object in Smalltalk, for example, has eight references on average [27]. Hence, the reference count size can be smaller. When the reference count of an object exceeds the maximum size, the count is not decremented when its referents are destroyed and the object becomes permanent. These objects are reclaimed by another technique such as mark/sweep or copying collectors.

The reclamation overhead of a reference count technique depends on the amount of garbage in the system since inaccessible objects are traced. This technique is therefore more useful in systems with large acyclic objects and low frequency of reference creation and destruction. Furthermore, the reference count technique can reclaim objects incrementally without causing significant interference with normal computation. Thus, it is also suited for real time applications and interactive environments.

Reference count techniques can be adapted easily to work in distributed systems since garbage objects can be collected independently from other computations. Furthermore, Bjornerstedt [7] observed that the overhead of garbage collection is distributed (not necessarily evenly) over the running applications in the system.

Unfortunately, reference count collectors are unable to reclaim circular structures. This problem can be solved by using another technique, such as mark/sweep, to garbage collect the entire system when the reference count technique fails. Alternatively, the programmers are responsible for explicitly unlinking circular references when these objects become obsolete.

The performance of reference counting collectors is poor due to the high accounting costs in maintaining the reference count. As illustrated in Figure 1.2, in order

to change a reference of one object, three objects must be updated. Ungar observed that reference count techniques consume as much as 15% of the processor time to maintain the reference counts and another 5% to reclaim storage of unreachable objects.

2.4 Deferred reference counts

Deutsch and Bobrow [18] developed a hybrid collector for a virtual memory environment which defers the reference count updates until an object is brought in memory. This algorithm is based on the statistical observation that in most Lisp programs, only two to ten percent of the Lisp cells (objects) have more than one reference [11] [12].

The algorithm maintains three hash tables: a multiple reference table (MRT), a zero count table (ZCT) and a variable reference table (VRT). The MRT, indexed by the object address, contains the reference counts of objects which have more than one reference. The ZCT holds the addresses of objects whose reference counts are zero. These objects are either referred to only by the variables of a program (still active) or are truly unreferenced and can be reclaimed. The reference count of objects which are in neither the MRT and ZCT is one. The VRT contains the addresses of objects referenced from the program variables. This table is updated when an object is referenced by a program variable so that main memory traversal can be avoided when garbage collecting.

When a new object is allocated, its address is added to the ZCT (it is not yet referenced by any objects in the system). When a new pointer is created, if it refers to an object in the MRT, the corresponding reference count value is incremented by one. If the new pointer refers to an object in the ZCT, the object is removed from this table since its count becomes one. If a new pointer is in neither MRT nor ZCT, the pointer refers to an object having a reference count of one. Hence, a reference count of two, keyed by this object, is placed in the MRT.

When a pointer referring to an object in the MRT is destroyed, the corresponding

object reference count value is decreased by one. If the new reference count value is one, the object is removed from the MRT. If, however, the pointer does not refer to an object in the MRT, (the object's reference count is one by default), the reference count is decremented to zero and the object is placed in the ZCT.

Any objects which are in the ZCT but are not in the VRT, are reclaimable since they are not reachable by other objects in the system. The reclaimed object space is linked to a free space list and the reference counts of its immediate references are decremented.

Deutsch and Bobrow realized that in a transaction, object accessibility is changed by the allocation of new objects, or the creation and destruction of pointers. Instead of updating the hash tables as the transactions occur, Deutsch and Bobrow proposed storing them in a sequential file. The transactions are examined at suitable time intervals to update the above tables. This scheme has the advantage of minimizing paging overhead since the allocation of an object is usually followed by the creation of a pointer to it; consequently, no updates to the ZCT are required.

The performance of the deferred reference count technique is an improvement over Collins' reference count algorithm but a large amount of processor time is still consumed. Ungar observes that this technique only consumes approximately 3% of the processor time to reclaim storage, 3% for periodic reconciliation and 5% for recursive freeing. This algorithm also suffers the same drawback as in the Collins' reference count as it is unable to reclaim circular objects.

2.5 Copying/moving collectors

Fenichel and Yochelson [21] introduced the idea of the copying collector in the late 60's, but the technique did not gain popularity until Baker [3] published his algorithm in the mid 70's. In this section, three popular copying collectors are discussed to show their superiority over the two previous methods.

2.5.1 Baker's real time semi-space

The Baker algorithm [3] is based on the Minsky garbage collector used by Fenichel and Yochelson in an early Multics Lisp. This method divides the working memory into two semi-spaces called ToSpace and FromSpace. The ToSpace is further divided into two areas, the creation area and evacuation area. During the execution of the user program, all newly created objects are located in the creation area of the ToSpace.

Garbage collection begins by incrementally evacuating objects directly referenced by a given root set in the FromSpace to the evacuation area of the ToSpace. The space of the evacuated objects in the FromSpace is replaced by forwarding addresses pointing to the corresponding new location in the ToSpace. Whenever a FromSpace object is referenced, the forwarding pointer is followed and the reference is updated to point to the ToSpace object. As illustrated in Figure 2.1 (a) and (b), when A is evacuated to the ToSpace, a forwarder (the shaded rectangle) is placed in its old address.

A scavenger process begins when all objects directly referenced by the root set are copied to the evacuation area. A scavenger linearly scans for references in the evacuation area pointing to the FromSpace. When these references are found, the scavenger evacuates the referenced objects to the evacuation area and updates the referents. The scavenger continues until all accessible objects in the FromSpace are moved to the evacuation area. In Figure 2.1(b), since A contains a reference to B in the FromSpace, the scavenger must evacuate B to the ToSpace. Figure 2.1(c) shows the result after B is evacuated.

The scavenger can be performed in a stop and copy manner or by interleaving with object allocation. In a stop and copy approach, the mutator is halted while the garbage collector executes and regains control when garbage collecting is finished. An interleaving mechanism copies a few FromSpace objects every time a new object is created. This interleaved collection/creation strategy can be made real time by bounding the time allowed for object copying.

When all reachable objects have been evacuated to the ToSpace, the memory

(a) Before garbage collection.

(b) After A is evacuated to ToSpace.

(c) After B is evacuated to ToSpace.

(d) ToSpace and FromSpace interchanged, object C is reclaimed.

Figure 2.1: An illustration of Baker's algorithm.

occupied by the FromSpace can be reused. An operation, called a *flip*, interchanges the ToSpace and the FromSpace. In Figure 2.1(d), shows the result of a flip.

Unlike the reference count technique, the Baker algorithm traces accessible objects instead of inaccessible objects. Therefore, Baker's algorithm is applicable to an environment where many objects become unreachable quickly after creation.

Baker's algorithm is often credited as being simple and elegant. In one pass, the space occupied by unreachable objects, including circular structures, is reclaimed and the memory space is compacted. These same tasks require at least three passes in the mark/sweep technique.

This algorithm evacuates objects in the breadth-first order using the evacuation area of the ToSpace as a queue to store the next object to scan. Hence, it does not require a collector stack like the mark/sweep technique.

A major drawback of the Baker algorithm is its inefficient use of storage space since the FromSpace is reserved as storage for objects to be evacuated. Effectively, only half of the available memory space is being used for computation at any time. Another drawback is the unnecessary copying back and forth of objects which survive a long period of time. These drawbacks are addressed by Lieberman-Hewitt [33] and Ungar [41] which are discussed below.

2.5.2 Lieberman-Hewitt algorithm

Lieberman-Hewitt's garbage collector is also known as the Generation Garbage Collector. This algorithm, a variation of the Baker algorithm, exploits the empirical observation that many newly created objects become unreachable quickly after their creation. In addition, this algorithm uses statistical knowledge of the Lisp environment (e.g. destructive operations are rare) to optimize the use of space.

This algorithm divides the memory space into regions based on age. The rate of garbage collection for each region depends on its age. The youngest region, containing the newly created objects, is scavenged most often since objects tend to become inaccessible soon after their creation. The older regions contain relatively permanent data; they will require garbage collection less frequently.

Garbage collecting a particular region is initiated by condemning it. A new region is created to hold accessible objects evacuated out of the condemned region. This new region inherits the age of the condemned region but has a higher version number. This number indicates the number of times a region has been garbage collected. The scavenger, as in the Baker algorithm, evacuates reachable objects and ensures that no pointers outside the condemned region exist before the space of a condemned region is reclaimed. Since condemned regions occur more often than flips in Baker's algorithm, the efficiency of the scavenger in this algorithm is very important. Otherwise, the garbage collector will monopolize all of the processor time.

Lieberman and Hewitt capitalize on an empirically observed property that most pointers in Lisp point backward in time (since destructive operations are not often used in Lisp). This observation leads to the restriction that objects are only allowed to point forward one generation, but backward any number of generations. Therefore, only the immediately previous region contains references pointing to the condemned region. Consequently, the scavenger is only required to search and update references in the region pointing the condemned region.

In optimizing the space utility, Lieberman and Hewitt's algorithm introduces several drawbacks. First, the restriction on forward pointers is only applicable in a functional language such as Lisp where the destructive operations are uncommon. In languages such as Smalltalk and Flavors, where destructive operations are pervasive, the Lieberman and Hewitt algorithm becomes impractical. Second, this algorithm tends to create chains of forward pointers to permit objects in older regions to point to those in much younger ones. As illustrated in Figure 2.2, for an object, A, originally in generation 1960, to point to an object B in the 1980 generation, A must be evacuated to the same region as B. This leaves behind a chain of forwarding pointers to conform to the forwarding pointer rule. Moreover, object evacuation is expensive as it is performed one generation at a time until the destination generation is reached. As in Figure 2.2, A is evacuated into the 1970 generation before moving to the 1980 generation. Furthermore, the pointer chasing incurs hidden cost when

Figure 2.2: An illustration of a forward chain in Lieberman and Hewitt's algorithm.

referencing a forwarded object.

2.5.3 Ungar Generation Scavenger

The Ungar algorithm, called Generation Scavenger, is another refinement of Baker's garbage collection technique. It exploits the knowledge that most objects either become unreachable soon after being created or linger for a very long time. Ungar introduced the notion of *tenure* to avoid repeated copying of long lived objects, hence, the performance of Generation Scavenger is better than that of Baker's algorithm.

Generation Scavenger divides memory space into NewArea and OldArea. The OldArea is a static area which contains tenured objects, and hence is garbage collected off-line. The NewArea is further divided into three areas: NewSpace, PastSurvivorSpace and FutureSurvivorSpace to conserve memory space. The NewSpace, relatively larger than the PastSurvivorSpace and FutureSurvivorSpace, contains newly created objects. The PastSurvivorSpace holds reachable objects that have been evacuated from previous scavenges. The FutureSurvivorSpace, initially empty, is used by the scavenger to evacuate reachable objects from NewSpace and PastSur-

vivorSpace.

A *remembered set* contains objects in the OldArea that have references to objects in the NewArea. It is used as a root set by the scavenger to evacuate new reachable objects in the NewSpace and PastSurvivorSpace to the FutureSurvivorSpace. When all reachable objects are copied to the FutureSurvivorSpace, the PastSurvivorSpace and FutureSurvivorSpace interchange while the NewSpace is reused. The scavenger removes objects in the *remembered set* which no longer refer to objects in the NewArea. When an object survives "enough" scavenges, it is promoted to the OldArea and is no longer subjected to on-line automatic reclamation.

Generation Scavenger is one of the most efficient techniques as it uses at most 3% of the processor time for garbage collection when objects are tenured at a "proper" rate. This algorithm also conserves main memory by dividing the NewArea into three spaces instead of two as in the Baker algorithm. The scavenger runs in a stop and copy fashion since the pauses it introduces are small enough to go unnoticed in normal interactive sessions.

Maintaining the *remembered set* introduces a hidden cost. When an object is stored in the OldArea, the *remembered set* must be updated if this object contains references to objects in the NewArea. Efforts have been made to reduce the maintenance cost by creating execution contexts in the NewArea so that local storage does not require testing.

A weakness of Generation Scavenger is the difficulty in deciding the proper time to tenure an object to the OldSpace (i.e., how many flips must an object survive before it gets promoted?). If an object is tenured too soon, it may become unreachable while in the OldArea, wasting memory. Whereas, if objects are tenured too slowly, long lived objects are repeatedly copied by the scavenger.

Many researchers are concentrating on this problem to provide a rule for tenuring objects. Several papers [36] [15] [44] suggest a stochastic method for determining object tenuring. All of these algorithms require a training period to determine a policy for tenuring objects by analyzing how objects are accessed, created and destroyed.

2.6 Secondary storage compaction

Compaction is a process which assigns objects to new locations to eliminate storage fragmentation. Compaction in secondary storage, however, is more complicated than in main memory since object relocation is much more expensive. This section examines the problems associated with secondary storage compaction by focusing on two issues:

- disk I/O
- object clustering

Ideally, disk I/O should be minimized to increase the performance of a compactor. As a side-effect of the object relocation performed during compaction, object clustering can be increased. By grouping related objects on the same physical page, disk I/O can be greatly reduced. These issues reinforce the difficulty of designing an efficient garbage collector for secondary storage.

2.6.1 Disk I/O

Unlike main memory, secondary storage access time is dependent on the disk arm position. Every secondary storage access requires a seek of the disk arm to a specified track and the transfer of data to main memory. Since these operations are slow, disk I/O should be minimized in order to obtain an efficient compactor.

Ideally, a compactor must eliminate storage fragmentation using the minimum number of disk I/O operations. This can only be achieved if storage fragmentation can be eliminated in the least number of object relocations. Unfortunately, this problem is NP-complete as shown in Theorem 1 below.

Fortunately, since secondary storage access is usually done by transferring pages (blocks of fixed size data), the compaction problem can be divided into two categories: global and page. Global disk compaction involves reorganizing pages to eliminate fragmentation. Since the page size is fixed, this reorganization is trivial.

The fragmentation size is always a multiple of the page size. Therefore, the global disk compaction can be achieved with the least number of page relocations.

Pages must be compacted to eliminate object fragmentation. Since objects vary in size, object compaction is very difficult to perform efficiently. However, since page size is small³, page compaction can be performed in memory. The compactor can move a whole page into memory in one disk operation and begin sliding objects to one end of the page. This task is not time consuming since there is a small number of objects in a page. The compacted page is then written back to disk. Thus, page compaction can be performed in two disk accesses.

The above solution does not result in zero disk fragmentation since the unused space in each page fragments the disk storage. However, this turns out to be a desirable feature since it allows objects to grow without being relocated.

Compaction problem

Compaction (Comp) Problem: Given a disk storage where the sizes of the fragmented areas are $k_1, k_2, k_3, \dots, k_j$ and a set of objects $\mathcal{O} = \{o_1, o_2, o_3, \dots, o_n\}$, where the size of an object $o_i = s_{o_i}$ for $i = 1, 2, 3, \dots, n$.

Question: Can disk compaction be done in less than n object relocations to achieve zero memory fragmentation?

Theorem 1 *The subset sum (SS) problem [22] is polynomially transformable to the compaction problem, therefore, the compaction problem is NP-complete.*

Proof:

The compaction problems are in NP since given a certificate containing the new location of each object, we can move these objects to their corresponding locations and therefore, verify the certificate correctness in $O(n)$ time.

³A page size usually ranges from 1 to 10K.

Given a subset sum problem which has a set of integers, $S = \{i_1, i_2, i_3, \dots, i_n\}$, and an integer k where

$$S = \sum_{j=1}^n i_j > k$$

the following construction transforms an SS instance into a Comp instance in polynomial time of $|S|$, such that the Comp instance answers **yes** if and only if the SS instance also answers **yes**.

Construction:

Create a set of objects $\mathcal{O} = \{o_1, o_2, o_3, \dots, o_n\}$ where the size of o_j is i_j for $j = 1, 2, 3, \dots, n$. Create a storage fragmentation of the size k and place the first object adjacent to this fragmentation. Subsequent objects are separated by a fragmentation whose size is greater than S .

$\Rightarrow$ A **yes** answer to a Comp instance implies zero fragmentation can be achieved in less than n relocations. This can be achieved if the first object does not need to be relocated and the rest do. Hence, there exists a subset of $\mathcal{O}$ whose size sums to k . Therefore, the SS instance also has a **yes** answer.

$\Leftarrow$ If an SS instance has a **yes** answer, we can move the objects whose sizes are in the subset, to the first fragmentation. Then slide the rest of the objects behind the first object. Therefore, zero fragmentation is achieved in $n - 1$ object relocations. Hence, the Comp instance also answers **yes**.

□

2.6.2 Object clustering

In order to compact storage, objects must be moved to fill the fragmented areas. Since objects must be moved, an increase in object locality can be gained by grouping related objects into the same unit of physical storage. Objects that are always used together should be placed on the same page. This process is called object clustering.

The placement of objects has a direct effect on the performance of an application since it directly affects the amount of disk I/O. Effective clustering allows all objects required in a computation to be transferred to main memory in one disk access. This is a significant performance improvement since disk I/O is minimized.

There are two ways to cluster objects. First, the user is given the responsibility of ensuring that related objects reside on the same page. These objects cannot be moved to any other area unless instructed by the user. Alternatively, the garbage collector can perform clustering automatically. In this case, the garbage collector is also responsible for detecting object structure changes as a result of computation and moving the affected objects to more appropriate locations. This method is preferable since it removes a burden from the user.

Chapter 3

Secondary Storage Garbage Collectors

There are many techniques for main memory garbage collection. The best of which consumes as little as three percent of the total processor time [41]. Surprisingly, little research is aimed at the development of garbage collectors for secondary storage. However, the recent interest in persistent objects has increased the need for secondary storage garbage collection.

This chapter reviews the major contributions to secondary storage garbage collection. Particularly, the work of Bjornerstedt [7], Campin [9] and Bishop [6] are examined. Bjornerstedt applied a mark/sweep technique to garbage collect secondary storage in a decentralized object-based system. Campin was the first to apply a secondary storage garbage collector to a persistent object environment. Bishop recognized the problems of applying the conventional garbage collector in a large address space and recommended a new system to solve these problems.

3.1 Bjornerstedt system

Bjornerstedt [7] devised a secondary storage garbage collector for a decentralized system. His system is modeled by a set of nodes where each node owns and controls its local resources. All nodes run a homogeneous software platform called an object manager. One of the object manager's tasks is administering a node resource by providing a transaction mechanism to ensure repository consistency, and a garbage collector to handle storage allocation and deallocation. In addition, the object

manager provides a remote procedure call (rpc) mechanism so that a node can communicate with other nodes in the system.

Bjornerstedt argues that the addressing scheme used is important since it has a direct influence on the method used for garbage collection. Each object in the system is addressed by a unique *persistent identifier*. At each node in the decentralized system, identifiers are mapped to a local persistent address, i.e., there is at least one level of indirection between identifiers and persistent addresses. This technique allows graceful object migration as objects are moved between nodes and enables several different persistent addressing mechanisms.

The problem of garbage collecting the global object space is delegated to the mechanism which collects the persistent object storage. Garbage collection of main memory objects is managed locally at a node during normal operation. Bjornerstedt advocates that garbage collection of persistent objects be done as rarely as once a week or even less often since persistent objects do not become garbage as fast as main memory objects. Hence, it is acceptable to temporarily suspend or degrade local computation while performing secondary storage garbage collection.

Bjornerstedt divides objects in the secondary storage of a node into three disjoint sets: *export*, *import*, and *local*. The import set contains objects referenced by this node which reside elsewhere in the system. The export set contains objects located at this node which are directly referenced by other nodes. The local set is the set of objects in this node which are not be directly referenced from other nodes.

An *Export-Import* (EI) list is maintained by the object manager during normal computation and keeps track of exported and imported references at a node. This is achieved by checking every outgoing and incoming message for references not in the list. A reference is marked as imported if it is found in an incoming message and is not in the EI list. Similarly, a reference is marked as exported if it is found in an outgoing message but not in the EI list. Using this method of determining imported and exported references, actual object lookup is avoided.

The Bjornerstedt garbage collection technique is based on the mark and sweep approach but consists of seven phases. The first three phases are run in a quies-

cent state where all transactions and object migrations are suspended. The first phase walks the entire secondary storage, marking all reachable objects as *MExport*, *MImport* and *MLocal* (where M stands for marked). Using the EI list, unreachable objects are further divided into three groups, *UExport*, *UImport* and *ULocal* (where U stands for unmarked). The second phase builds an *external reachability message* (ERM). It begins by marking objects in the *UImport* and *ULocal* as accessible from the *UExport* group. Hence, the *UImport* and *ULocal* are further divided into *MUImport*, *UUImport*, *MULocal* and *UULocal* sets. Figure 3.1(a) shows the result obtained from node N3 after the second phase. The first part of the ERM (ERM-1) contains the *MImport* object set. The second part (ERM-2) contains a set of pair references; the first reference is an object in the *UExport* set, the second is an *UImport* object reachable from the first object as illustrated in Figure 3.1(b). The third and fourth phases sweep the entire secondary storage reclaiming objects identified as *UUImport* and *UULocal* respectively. The fifth phase exchanges the ERMs between all nodes. This phase may run in parallel with normal operations since it may require a considerable amount of time to obtain the ERMs from all other nodes. Once a node has received all the ERMs, it begins the sixth phase by building a global graph. The members of the ERM-1s form a global root set which is used to trace the paths in the ERM-2. Once all the objects are traced and marked (a global graph is formed), the seventh phase begins by collecting unmarked objects and cleaning up the EI table.

This algorithm is one of the first garbage collectors designed for a decentralized, persistent object system. Considerable node autonomy is achieved as a node performing purely local processing has no overhead for the garbage collector since there is no EI list maintenance.

This algorithm, however, is very naive since the main mechanism for garbage collection is based on the mark and sweep approach. This technique, as pointed out earlier, is infeasible in large storage since the entire storage space must be traversed causing excessive disk I/O operations.

Another drawback of this algorithm is the size of an ERM since it can be as

(a) System snapshot

(b) External reachability message

Figure 3.1: An illustration of Bjornerstedt's algorithm.

large as the computational space of a node. This case occurs when all objects in a particular node are locally unreachable, but are globally reachable. Exchanging such large ERM's in a network is costly and clumsy. Moreover, other nodes in the system may not have enough space to receive such large ERM's.

Since a node cannot begin the sixth phase until all ERM's are received, the speed of garbage collection is dictated by the slowest node in the system.

3.2 Campin's Garbo

Campin and Atkinson [9] implemented Garbo, a breadth-first copying garbage collector to reclaim and compact secondary storage used by the PS-algol [2] persistent programming language. In this system, secondary storage is divided into persistent stores. Each persistent store is a collection of Unix files in a single directory. A persistent store is further partitioned into a number of zones, called databases, enabling Garbo to garbage collect the secondary storage piecewise. A PS-algol process is restricted to access one persistent store during its execution life time.

Each database has two permanent files and one temporary file. The *data file* containing the objects and the *index file* containing offsets of objects in the corresponding data file are the permanent files. The position of an object in an index file is computed from an object's unique 32-bit pointer which is called a *persistent identifier*. The *shadow file* is a temporary file used to recover from crashes which occur while writing to a database.

The users can explicitly delete a database. This "soft" deletion merely replaces the database name field in the directory file with an empty string, inhibiting the users from directly accessing this database by name. Databases are only removed from the system when all of the associated bindings have been destroyed.

The database type determines the frequency of garbage collection. For instance, a bibliographic database contains long persistent items; therefore such databases seldom need garbage collection. Bitmap image databases are constantly being modified, therefore, they are short lived (days or hours). Consequently, they require

daily or hourly garbage collection.

Each database maintains a list of referenced databases in a *linked databases vector*. This permits Garbo to garbage collect a single database without scanning the whole persistent store. This vector is stored in the data file of a database and is updated during the compaction phase of Garbo.

Garbo consists of two phases. In the first phase, the database directory and all of the databases are locked. This locking allows Garbo to scan the entire persistent store starting from named databases and build up a table of cross-references between databases. The second phase begins by backing up the data, index and shadow files before starting a potentially destructive operation. These backup files are used by a recovery process when a system crash occurs while garbage collecting. As part of the second phase, Garbo uses the cross-reference table as a root set of live data. The live data is traced and copied into the new data file, while a new corresponding index file is concurrently updated. Databases with no external references are removed from the directory file and their partitions are reclaimed.

Since cross-reference table construction in the first phase is time consuming, Campin proposed that this table be constructed during the second step of Garbo while references of all live objects are traced. The information stored in the data file of a database is used the next time garbage collection occurs.

While garbage collecting, Garbo also produces a statistics file containing changes in each database and the access frequencies of each object. Using this information, an increase in database locality can be obtained by migrating objects to locations where they are more likely to be accessed.

The major contribution of Garbo is the recognition that garbage collection in secondary storage must be done piecewise. As described, Garbo reclaims a single database without scanning the entire persistent storage. Garbage collecting piecewise also allows Garbo to concentrate on databases which contain a large amount of garbage. In this way, Garbo can avoid wasted effort in reclaiming storage of databases that contain very little, if any, garbage.

One of Garbo's weaknesses is its inability to reclaim circular structures that span

more than one database. Objects in the cross-reference table of each database are assumed to be reachable even if they are garbage. User assistance is required to break the circular links before Garbo can reclaim this storage.

Campin argued that the breadth-first search is superior to depth-first search since all live pointers in one database can be traced before advancing to the next database, avoiding the frequent opening and closing of files. The depth-first search also requires many databases to be opened simultaneously; possibly exceeding the upper limit on the number of opened files imposed by Unix. However, Campin does not observe that depth-first search can improve data locality [15], reducing the cross-references between databases.

The design and implementation of Garbo is not highly regarded since the persistent stores are just an extension of the Unix file system (allowing files to be accessed not only by their name but also by references in another file). Thus, a file name may be deleted from the system directory to prevent direct access by users, but its contents are still preserved. This approach is only practical if object size is large, and the rate of object creation and destruction is low. Otherwise, Campin's approach becomes awkward and clumsy since the disk space is reclaimed on a per file basis. This unit of reclamation is convenient for the implementers since it eliminates the task of managing the allocation and deallocation of disk space. On the other hand, the unit of space allocation is too large, thus storage is wasted. Furthermore, Garbo performs poorly. It takes approximately 10 to 15 minutes to collect and compact one megabyte of data if the inter-database references are small. Otherwise, up to 50% more of the above time is required.

3.3 ORSLA garbage collector

Bishop [6] described a garbage collector for a new computer system called ORSLA (Object References in a Single, Large Address space). The ORSLA garbage collector not only reclaims unused storage, but also eliminates storage fragmentation and increases locality of references. Bishop realized that garbage collection in a very

large address space must be done piecewise independently from the rest of the system. Furthermore, garbage collection must be applied to areas with a high rate of mortality. The following is an overview of ORSLA which gives insight into Bishop's collection algorithm.

3.3.1 Area

ORSLA partitions storage into *areas* which are sets of adjacent pages. Bishop recommended an area size to be five pages since internal area fragmentation can be minimized and garbage collection can be performed in a reasonable time. Areas are categorized into two classes: permanent areas containing long term data, and local computation areas (LCA) containing temporary results of some computations. ORSLA provides a mechanism to ensure related objects are placed in the same area to reduce page swapping and inter-area links.

3.3.2 Inter-area links

Each area contains a list of incoming and outgoing references to provide a complete list of the objects within the area which are referenced from other areas and vice-versa. These lists allow the garbage collector to reclaim a single area separately by assuming all the objects in the incoming list are accessible. The elements in each list are called *inter-area links* (IALs). IALs provide a method for objects in one area to reference objects in a different area. Each IAL contains three fields:

1. a reference to an object in a different area.
2. a next IAL of an incoming list of an area linked by an outgoing link.
3. a next IAL of an outgoing list of an area where the link resides.

As illustrated in Figure 3.2, *a* in area *A* requires IALs to reference *b* and *c* in area *B*. The outgoing list of area *A* is threaded through the o-field in the IALs. Similarly, the incoming list of area *B* is threaded through the i-field in the IALs of area *A*.

Figure 3.2: ORSLA storage organization.

The lists of IALs are continuously maintained as part of normal computation, which may create a large overhead. To reduce this overhead, Bishop developed a special virtual memory hardware to trap IALs during the loading and unloading of objects. Bishop also suggested that IALs should only be used as links between objects in different permanent areas, reducing the number of IALs. The reduction of IALs is further achieved through the use of *cables* described in the following section.

3.3.3 Cables

Since references in LCAs are short lived, and the maintenance, creation and destruction of IALs are expensive, a *cable* mechanism is used to allow objects in an LCA to contain direct references to objects in permanent areas. If area $\mathcal{A}$ has an outgoing cable to area $\mathcal{C}$, objects in $\mathcal{A}$ can have direct references to those in $\mathcal{C}$ reducing the number of inter-area links.

Each cable consists of a reference pointing to a cabled area and two other references to thread the list of incoming and outgoing cables. ORSLA automatically constructs cables when objects in an LCA references those in a permanent area. However, cables can also be created explicitly by the users. In such cases, the user must weigh the increased size and time of garbage collection over the savings of IALs.

Figure 3.2 shows an outgoing cable from area $\mathcal{A}$ to area $\mathcal{C}$. Object a can therefore reference d without creating an IAL. Since area $\mathcal{C}$ has one incoming cable, the *i*-field of the cable in $\mathcal{A}$ is empty. Similarly, $\mathcal{A}$ only has one cable, hence, its *o*-field is also empty.

3.3.4 Area header

Each area has a header containing the information necessary to manipulate objects within it. Information inside an area header is maintained by ORSLA when an area is created, it is not modifiable by the user. The header information includes the roots of the incoming and outgoing fields of IALs and cables. In Figure 3.2,

the header of each area is represented by the shaded region. The IAL and cable information in an area's header is used exclusively by the garbage collector.

3.3.5 ORSLA garbage collector

Garbage collection can be invoked by the user or is invoked automatically when there are many inaccessible objects degrading the system performance. The garbage collector begins by locking an area to be scavenged and all of its cabled areas. The locked areas suspend all processes trying to access objects in them, and prevent other processes from creating new cables to these areas. All cabled areas must be garbage collected together; thus, chains of cables should be kept short. In fact, cables are rarely used in permanent areas, and are only used in local computation areas.

When all locks are obtained from scavenging areas, the garbage collector constructs a cable from its LCA to each collected area, and begins to evacuate accessible objects to their respective new areas. Since the garbage collector does not have any information about objects outside the collected areas, it assumes all outside objects are accessible. Thus the root of a collected area is found by scanning the list of incoming links. When an object is copied to a new location, its incoming references are also updated to point to the new location.

Implicit cables are copied to the new area if direct references exist in the new area to another area; otherwise, they are removed. All explicit cables are copied except for those that have been deleted by the users. References associated with the removed cable are converted into IALs when they are copied into the new area. When all objects are evacuated, all outgoing references of inaccessible objects are removed, suspended processes are resumed and storage of the old areas are reclaimed.

3.3.6 ORSLA contribution

Like Campin, Bishop recognized the importance of incremental garbage collection in a large address space. Garbage collection of a few areas in sequence provides

a method to reclaim the space occupied by inaccessible objects incrementally. In addition, garbage collection can occur at different rates in different areas so that the garbage collection effort can be concentrated on the most needed areas. Furthermore, the garbage collection process does not interfere with the computation processes not involved with the collected areas.

Maintenance of the IALs is expensive since each object reference must be checked for IALs before it can be used. Furthermore, the IAL maintenance can cause excessive disk operations if the incoming IALs to be updated are not in main memory. To reduce this overhead cost, a cable mechanism is devised to reduce the number of IALs in the system. To further reduce the overhead incurred by IALs, Bishop proposed a modification to the virtual memory hardware so that IALs can be trapped by the load and store operations.

3.4 Summary

Bjornerstedt's garbage collector is concerned with reclaiming unused, shared persistent objects in a distributed environment. However, his algorithm fails to address the problem of garbage collecting in a large secondary storage. The local sweeping of the secondary storage in the first phase, for example, can take days to perform. Furthermore, this phase must be performed in a quiescent state; therefore, the system must be shut down during that time. Bjornerstedt's technique, therefore, is not applicable to systems with large secondary storage.

Both Campin and Bishop recognized that secondary storage garbage collection must be done piecewise since the disk address space is too large to garbage collect at once. Bishop pointed out that just scanning the disk space alone can take weeks, not to mention the large storage required to store the scanned information. Moreover, piecewise garbage collection avoids wasting processor time in areas or databases which contain little or no garbage.

Bishop's approach requires maintenance of the lists of incoming and outgoing inter-area links. These links allow an area to be garbage collected separately from

the rest of the system. To avoid incurring substantial run-time overhead for link maintenance, Bishop altered the virtual memory mechanism to trap inter-area links when they are stored into cells. He also proposed special virtual memory hardware so that this extra service can be performed efficiently.

Campin, on the other hand, divides the disk space into databases, and garbage collects each database separately. To avoid scanning the whole disk space, a cross-reference table is employed to contain external references between databases. Although Campin failed to describe his system in detail, his garbage collector performs poorly largely because the system is built on top of the Unix file system.

Neither Campin nor Bishop considered the cost of swapping pages caused by the garbage collector when updating the cross references (inter-area links in Bishop's case and cross-reference tables in Campin's system). Reference updates can be slow if referenced objects must be brought in from secondary storage. In summary, both Bishop and Campin provide a good background for secondary storage garbage collection and motivate the incremental nature of the Creepy algorithm.

Chapter 4

Creepy

In this chapter, we present the Creepy algorithm and show how it is designed for secondary storage. A combination of the copying and reference count techniques is used to enable Creepy to incrementally garbage collect in near real time. Copying garbage collection is performed on a page by page basis using the reference count to provide the root set for a page. This eliminates the need for scanning the whole disk. Unlike many reference count techniques [18] [14] with a large overhead, Creepy has a very low overhead since the reference count updates are only performed at load and unload time.

Throughout this chapter, the notation Rf_A is used to represent a set of objects immediately referenced by A, RC_A denotes the entry in the RefCountTable indexed by A. This notation is used to simplify the explanation of the Creepy algorithm.

First, an overview of the algorithm is described for a single processor environment. Following this, a more detailed version is described which includes collecting circular structures as well as extending Creepy to a multi-user system. A proof of correctness for the algorithm is provided. A comparison between this and other algorithms points out Creepy's advantages and drawbacks.

4.1 Overview

Creepy is an incremental garbage collector for secondary storage designed to work in a system where the ratio between secondary storage and main memory is a factor

of one hundred or more. In such systems, garbage collecting the whole secondary storage at once is infeasible. Too much time is required to trace all accessible objects or to scan the whole storage. Instead, Creepy divides the disk space into pages and garbage collects each page independently from the others. Since objects are moved between main memory and secondary storage in units of page, objects used together are clustered on the same page to avoid disk I/O operations.

Creepy works in two modes: on-line and off-line. On-line mode is used when user transactions are active. Off-line mode is executed when no transactions are active.

In on-line mode, Creepy performs two tasks which demand very little processor time to avoid degrading the performance of the user transactions. The first of these maintains reference counts of objects stored on disk. The second reclaims the space of unused objects in pages fetched into memory by the user transactions.

Creepy's reference counts have the same role as inter-area links in Bishop's algorithm, namely providing the root set for a page. Reference counts are stored inside their object on disk. However, when an object is moved into main memory, its reference count is removed. The reference counts are maintained instead, by enumerating the number of loaded and unloaded referents of an object. This enumeration enables Creepy to calculate the reference count of an object.

A page is garbage collected using the copying technique. The root set is formed from the objects which have unloaded referents since Creepy assumes that unloaded objects are accessible. Using this root set, a page can be garbage collected without scanning the disk. Starting from the root set, a copying garbage collection technique evacuates all accessible objects into the new page. This page is written to disk when all accessible objects are copied, hence, reclaiming the space occupied by inaccessible objects.

In the off-line mode, Creepy exploits the idle processor time and unused memory to perform its computationally intensive tasks. It begins by filling half of main memory with pages from secondary storage. Then, in addition to the copying algorithm described in the on-line mode, the following tasks are also performed:

1. "short-circuiting" forwarding pointers in these pages.

2. clustering related objects into the same page.
3. compacting these pages to reduce storage fragmentation.

The above pages are written back to disk after they have been garbage collected. Even in the off-line mode, disk I/O operations are kept to minimum since all pages are only read and written once while garbage collecting. Furthermore, the entire secondary storage space is only scanned once to reclaim storage space of inaccessible objects.

4.2 Creepy algorithm

The reference count technique provides the root set of a page so that it can be garbage collected independently using the copying method. This section provides a detailed description of the on-line and off-line modes of Creepy to justify the earlier claims.

Objects stored on disk contain a count of the number of their referents. A unique *object-location* is associated with each of these objects. Object-locations are used solely by Creepy, the user applications do not have any knowledge of them. Object-locations are used as indices into the LoadedTable and the RefCountTable. These tables are maintained in stable storage [5] so that they can be recovered in the event of a system crash. In addition, they enable Creepy to calculate the reference count of an object (discussed below).

The LoadedTable contains the list of object-locations of objects which have been fetched into main memory. When an object is unloaded from memory, its object-location is removed from this table. Figure 4.1 is a snapshot of a running system. Loaded objects are represented by circles and unloaded objects by rectangles. The LoadedTable contains entries for objects A and B, hence they are loaded (shown as circles).

The RefCountTable allows Creepy to defer the reference count update until an object is unloaded to secondary storage, eliminating unnecessary disk I/O. In addition, the reference count can be calculated from this table.

Figure 4.1: A snapshot of a running system.

The RefCountTable indexed by the object-location contains entries for loaded objects and their immediate references. The entries depend on the corresponding object's status. If an object is loaded, the entry indicates the number of unloaded referents. Otherwise, it indicates the number of loaded referents to the object.

In Figure 4.1, A and D are referents of B, therefore B's reference count is two. Since A and B are loaded, B has only one unloaded referent, namely D; consequently, the value in the RefCountTable for B is one. Object C is not loaded and B is a loaded referent of C; hence, the value in the RefCountTable for C is also one.

In the traditional reference count technique, a count is kept inside an object representing the number of objects pointing to it. This count must be incremented/decremented as an object's referent is created/destroyed. Therefore, reference count maintenance consumes a large amount of processor time and can cause excessive disk I/O operations if applied to secondary storage.

Creepy only keeps reference counts inside the corresponding objects' page on disk. When an object moves into memory, the reference count is discarded. Instead of enumerating the reference count, Creepy enumerates the number of loaded and unloaded referents of an object. This enumeration is stored in the RefCountTable as described above. The reference count can be calculated from the sum of loaded and unloaded referents of an object. This observation allows Creepy to overcome the high cost of updating reference counts since they are only updated when objects are unloaded to disk.

4.2.1 On-line operation

Creepy's on-line mode minimizes its interference with user transactions by garbage collecting only those pages brought into memory by the user transaction. The most often used pages are reclaimed more frequently than the inactive ones since they are more likely to be fetched into main memory.

The on-line Creepy algorithm shown in Figure 4.2 is invoked whenever a page in memory cannot accommodate a given object. This (old) page is condemned and a new page is created containing the accessible objects from the old page. Like


```

procedure onLineCreepy(page)
  newPage  $\leftarrow$  create new page.
  rootSet  $\leftarrow \{x \mid x \in \text{LoadedTable} \text{ and } RC_x > 0\}$ .
  copy all accessible objects from rootSet in page to newPage
  foreach  $x \in \{\text{inaccessible objects}\}$  do
    decrement referenceCountRefx
  endforeach
  if newPage is empty then
    return newPage to page pool
  endif
  overwrite page with newPage
endprocedure

```

Figure 4.2: Creepy algorithm in on-line mode.

Bishop's algorithm, all objects outside of the collected page are assumed accessible; therefore, the *root set* consists of those loaded objects which have unloaded referents¹. Loaded objects are traversed breadth-first starting from the root set. Traversal ends when all paths have been walked. A path is terminated by an unloaded object. Reachable objects are evacuated to the new page. When all accessible objects have been copied, the objects which are reachable from a garbage object must have their reference count decremented. If all objects in a page are inaccessible, the empty page is returned to the free page pool for reuse. The collection terminates by writing the new page in place of the old one.

The write operation must be atomic to ensure storage consistency. This requirement can be satisfied through the use of a shadowing or logging technique. The shadowing technique copies a new page to stable storage and then alters the page pointer to reflect the change. The logging technique writes the new page to a logged file before overwriting the original page. Both of these methods are discussed by Bernstein [5], the logging technique is preferable since it preserves page clustering.

In Figure 4.3 when P_1 is condemned, RC_A (A's entry in the RefCountTable) is greater than one. Therefore, the root set contains only A. A breadth-first traversal starting from this root set will find that B is also accessible. Hence, A and B

¹Objects whose entry in the RefCountTable is non-zero.

Figure 4.3: An example of Creepy working in on-line mode.

```

procedure offLineCreepy
  while memory < 1/2 filled do
    loadedObject  $\leftarrow$  load an object from secondary storage
    if loadedObject is a forwarding pointer then
      put loadedObject in ForwardTable
    endif
  endwhile
  rootSet  $\leftarrow$  {loadedObject |  $RC_{loadedObject} > 0$ }
  use rootSet to perform copying collector and
  short-circuit forwarding pointers in ForwardTable
  foreach  $x \in$  {inaccessible objects} do
    decrement referenceCountx
  endforeach
  unload the new pages in place of the old ones
endprocedure

```

Figure 4.4: Creepy algorithm in off-line mode.

are evacuated into a new page. C and D reference each other, but they are not reachable from the root set. Hence, C and D will not be evacuated, and their space is reclaimed. F's referent is the garbage object C, therefore, the reference count of F must be decremented by one.

4.2.2 Off-line operation

Creepy runs in off-line mode when no user transaction is active. Creepy takes advantage of the idle processor and unused main memory to perform its time consuming tasks such as disk compaction, "short circuiting" forwarding pointers and object clustering.

Figure 4.4 is the algorithm executed by Creepy in the off-line mode. In this mode, half of main memory is filled with pages read from secondary storage. In addition to the maintenance of the RefCountTable and the LoadedTable, the ForwardTable is created to process forwarding pointers. The ForwardTable is indexed by the old object-location, its corresponding value is the new object-location of the moved object. A loaded referent containing forwarding pointers in this table is updated to the new locations. When all references (loaded and unloaded) to a forwarding

pointer have been short-circuited, it is removed from the ForwardTable, and its space is reclaimed.

The root set of the pages to be collected is a set of the loaded objects whose value in the RefCountTable is greater than zero. Using the root set, Creepy behaves as in the on-line mode. Loaded objects are traversed depth-first starting from the root set. Traversal ends when all paths have been walked. A path is terminated by an unloaded object. Reachable objects in the old pages are evacuated to the new pages. To avoid creating new forwarders, objects whose entry in the RefCountTable is zero, are allowed to relocate to a different page. All referents of these objects are in memory so they can be updated immediately to point to the new location.

After all accessible objects have been copied, the reference count of objects which are reachable from a garbage object, is decremented. Then, the new pages are written in place of the old pages. Unless there is a user transaction active, Creepy repeats this cycle by refilling half of memory with a next set of pages.

Figure 4.5 is an illustration of Creepy garbage collecting P_1 and P_2 in the off-line mode. Figure 4.5(a) shows the snapshot of the system after the pages P_1 and P_2 are loaded. Object D, a forwarder, is put in the ForwardTable. A and F form the root set since RC_F and RC_A are one. A and F are referenced by an object in P_3 , they cannot be moved outside of P_1 or P_2 respectively. B and G, on the other hand, are relocatable without creating forwarding pointers. After P_1 and P_2 are garbage collected, the forwarder, D, is removed, since B is "short-circuited" to point directly to G. The space occupied by the circular structure illustrated by objects C and E, is reclaimed since their reference counts are zero and they are not reachable from the root set. Figure 4.5(b) shows the result after pages P_1 and P_2 are garbage collected and unloaded.

4.3 Reference count maintenance

The efficiency of Creepy is due to the inexpensive maintenance of the reference counts. In this section, the maintenance of Creepy's reference counts is described in

Figure 4.5: An example of Creepy working in off-line mode.

```

procedure load(objectLocation)
  if objectLocation  $\in$  LoadedTable then
    return(the loaded object)
  else
    loadedObject  $\leftarrow$  an object on disk at objectLocation
    add loadedObject to LoadedTable
     $RC_{loadedObject} \leftarrow DiskReferenceCount_{loadedObject} - RC_{loadedObject}$ 
    foreach  $x \in Rf_{loadedObject}$  do
      if  $x \in$  LoadedTable then
        decrement  $RC_x$ 
      else
        increment  $RC_x$ 
      endif
    endforeach
  endif
  return(loadedObject)
endprocedure

```

Figure 4.6: Load algorithm.

the context of the load and unload operations.

4.3.1 Loading

It is important that the maintenance of reference counts in the on-line mode does not demand much processor time since it can affect the performance of user transactions. Creepy's load algorithm in Figure 4.6 shows how the RefCountTable and the LoadedTable are updated when an object, A , is loaded into main memory. When A is loaded into main memory, its object-location is added to the LoadedTable. The number of unloaded referents of A and the number of loaded referents of its immediate references (Rf_A) are updated in the RefCountTable to avoid disk I/O. The RC_A (the value of A in the RefCountTable) is updated to contain the number of unloaded referents of A . This number is obtained by calculating the difference between the on disk reference count of A (the total number of referents of A stored on disk) and the current RC_A (the number of loaded referents of A).

Since A has been loaded, there is one more loaded object and one less unloaded

Figure 4.7: An example of object loading.

object. The loaded or unloaded count of each object in Rf_A must be updated in the RefCountTable. If an object in the Rf_A is loaded, its RC (the value in the RefCountTable) is the number of unloaded objects pointing to it. Therefore its RC number is decremented by one to reflect that there is one less unloaded object pointing at it. On the other hand, if an object in the Rf_A is not loaded, its RC contains the number of loaded objects pointing to it. Thus, its RC is incremented by one to indicate that there is another loaded object pointing at it.

Figure 4.7 illustrates how the RefCountTable and LoadedTable are updated using the algorithm in Figure 4.6. In Figure 4.7(a), A, is loaded and is referenced by D

on disk, hence RC_A is one. B, on the other hand, is not loaded, and is referenced by A, thus RC_B is also one.

When an object, D, is loaded into memory, an entry for D is added to the LoadedTable. RC_D remains zero since both the on disk reference count of D and RC_D are zero. A and C are immediate referenced objects of D, therefore their reference count must be updated. Since A is loaded, RC_A is decremented by one. C, on the other hand, is unloaded so RC_C is incremented by one. The RefCountTable of Figure 4.7(b) shows the results after the above updates.

4.3.2 Unloading

The unloading process is similar to loading. When an object A is unloaded to disk, the RefCountTable and the LoadedTable must be updated. Figure 4.8 shows the algorithm for updating the RefCountTable and the LoadedTable when A is unloaded to disk. A's entry in the LoadedTable is removed to reflect its unloaded status. Let $\mathcal{A}$ be the number of loaded objects which have a direct reference to A. The number $\mathcal{A}$ can be computed by performing a garbage collection in main memory or by tracing accessible objects in memory. The reference count of A is the sum of RC_A and the number $\mathcal{A}$. The RC_A is replaced by $\mathcal{A}$ to conform with the RefCountTable rule stated earlier.

Since A has been unloaded, there is one more unloaded object and one less loaded object. The loaded or unloaded count of each object in Rf_A must be updated in the RefCountTable. If an object in the Rf_A is loaded, its RC is the number of unloaded objects pointing to it. Therefore its RC number is incremented by one to reflect that there is one more unloaded object pointing to it. On the other hand, if an object in the Rf_A is unloaded, its RC contains the number of loaded objects pointing to it. Thus, its RC is decremented by one to indicate that there is one less loaded object pointing at it.

Figure 4.9 illustrates the updates to the RefCountTable and the LoadedTable when unloading an object, A. A is originally referenced by D and F as shown in the Secondary Storage of Figure 4.9(a). During the course of computation, E becomes


```

procedure unload(anObject)
  remove anObject from LoadedTable
   $\mathcal{X} \leftarrow$  number of loaded referents of anObject
   $DiskReferenceCount \leftarrow RC_{anObject} + \mathcal{X}$ 
  store anObject with  $DiskReferenceCount$  to disk
   $RC_{anObject} \leftarrow \mathcal{X}$ 
  foreach  $x \in Rf_{anObject}$  do
    if  $x \in LoadedTable$  then
      increment  $RC_{Rf_x}$ 
    else
      decrement  $RC_{Rf_x}$ 
    endif
  endforeach
endprocedure

```

Figure 4.8: Unload algorithm.

A's referent. When A unloads, there are two referents of A in memory, namely E and D. Since RC_A is one, there exists one object on disk (namely F) referencing A. There are three objects in total which reference A. When A is unloaded to disk, A's reference count is updated, and its entry is removed from the LoadedTable. The RC_A is updated to contain the number of loaded referents of A (in this case two). B and C are immediate referenced objects of A, therefore, their count of unloaded referents must be incremented. Since B is loaded, its count of unloaded referents is incremented by adding one to the RC_B . On the other hand, C is unloaded, thus, its count of loaded referents is decremented by subtracting one from the RC_C . The results of these updates are shown in the RefCountTable of Figure 4.9(b).

Note that it is possible for the value in the RefCountTable to be negative. This occurs if more referents of a same object are unloaded than loaded. In Figure 4.10, when A is loaded into memory, the RC_B is incremented by one since B, an immediate referenced object of A, is not loaded. Similarly, the RC_C is one when D is loaded into memory. As a result of a computation, D is modified to reference B as illustrated in Figure 4.10(a). If both A and D unload to disk, the RC_B is decremented by two. Thus, the RC_B is negative one as shown in the RefCountTable of Figure 4.10(b).

Figure 4.9: An example of object unloading.

Figure 4.10: An example of negative value in the RefCountTable.

4.4 Correctness of the Creepy algorithm

This section provides a proof of correctness for the Creepy algorithm. The inductive technique is used to show that the sum of the loaded and unloaded referents of an object is properly maintained in the RefCountTable. That is, if a new referent is created, the reference count of an object is incremented and if an existing referent is destroyed, the reference count of an object is decremented.

4.4.1 Inductive proof

Let X_i and Y_i be the number of loaded and unloaded referents of $\mathcal{A}$ at time i respectively. Also, let Z_i be the reference count of $\mathcal{A}$ at the same time i . Since the reference count of $\mathcal{A}$ is the sum of loaded and unloaded objects of $\mathcal{A}$, $Z_i = X_i + Y_i$.

From Figure 4.8 and Figure 4.6 the number of unloaded objects referencing $\mathcal{A}$ at time i is defined as:

$$Y_i = \begin{cases} RC_{\mathcal{A}_i} & \text{if } \mathcal{A} \text{ is loaded} \\ DiskReferenceCount - RC_{\mathcal{A}_i} & \text{if } \mathcal{A} \text{ is unloaded} \end{cases}$$

At time instant 0, when an object has no referents, $Z_0 = 0$.

Since the object has no referents, X_0 , the number of loaded objects at time instant is 0.

$RC_{\mathcal{A}_i} = 0$ for $\mathcal{A}$ either loaded or unloaded, therefore, $Y_0 = 0$.

Thus, $Z_0 = X_0 + Y_0$.

Let $\mathcal{B}$ be a new object referencing $\mathcal{A}$ at time $i + 1$. We are thus required to prove that $Z_{i+1} = Z_i + 1$ when $\mathcal{A}$ is loaded as well as unloaded.

Case 1: $\mathcal{A}$ loaded

At time i , $Z_i = X_i + Y_i$ and $Y_i = RC_{\mathcal{A}_i}$, therefore, $Z_i = X_i + RC_{\mathcal{A}_i}$.

When $\mathcal{B}$ unloads, $RC_{\mathcal{A}_{i+1}} = RC_{\mathcal{A}_i} + 1$ (Figure 4.8)

$X_{i+1} = X_i$ and $Y_{i+1} = RC_{\mathcal{A}_{i+1}}$

$$\begin{aligned}
Z_{i+1} &= X_{i+1} + Y_{i+1} \\
&= X_{i+1} + RC_{A_{i+1}} \\
&= X_i + RC_{A_i} + 1 \\
&= X_i + Y_i + 1 \\
&= Z_i + 1
\end{aligned}$$

Case 2: $\mathcal{A}$ unloaded

When $\mathcal{B}$ unloads, $RC_{A_{i+1}} = RC_{A_i} - 1$ (Figure 4.8)

$X_{i+1} = X_i$ and $Y_{i+1} = RC_{A_{i+1}}$

$$\begin{aligned}
Z_{i+1} &= X_{i+1} + Y_{i+1} \\
&= X_{i+1} + DiskReferenceCount - RC_{A_{i+1}} \\
&= X_i + DiskReferenceCount - (RC_{A_i} - 1) \\
&= X_i + DiskReferenceCount - RC_{A_i} + 1 \\
&= X_i + Y_i + 1 \\
&= Z_i + 1
\end{aligned}$$

Similarly, it can be proved $Z_{i+1} = Z_i - 1$ when a reference to $\mathcal{A}$ is destroyed. Therefore, the reference count is properly maintained in the RefCountTable when a new referent is created.

4.5 Large RefCountTable

When an object is loaded or unloaded, the count of this object and its immediate references are updated in the RefCountTable. This table may potentially grow so large that it cannot be stored in memory. This section describes an optimization

to reduce the RefCountTable size and proposes two schemes to guarantee that this table will always fit in memory.

The entries in the RefCountTable are non-zero when:

1. A loaded object has unloaded referent(s).
2. An unloaded object has loaded referent(s).

Since a transaction requires a set of related objects to be loaded into memory for its computation, the loaded objects usually have no unloaded referents. Thus, the values in the RefCountTable indexed by these loaded objects are usually zero. A similar argument can be used to show that the values in this table are zero for those objects not required by a transaction. Therefore, a significant size reduction is achieved by defaulting the value of those objects not in the table to be zero.

Although this step reduces the RefCountTable size, it does not guarantee that the RefCountTable can fit in memory. This problem can be tackled in two ways. First, an "out-of-memory" error is signaled forcing the user to decide which objects are essential to his/her applications and unload the obsolete objects. Second, a background process which runs occasionally depending on the need, forces objects to unload if they are not required in any active transactions. Both methods ensures that the RefCountTable will always fit in memory, but the second approach is preferable to the first since it can be performed without user assistance.

4.6 Advantages and disadvantages

Creepy is a garbage collector for secondary storage that can reclaim storage of inaccessible objects incrementally in near real time. This is achieved by having Creepy operate in two modes: on-line and off-line.

In the on-line mode, Creepy minimizes its interference with a running transaction by deferring the reference count updating task. Creepy differs from Deutsch and Bobrow deferred technique [18] by maintaining the number of loaded and unloaded

referents of an object. As a result, Creepy updates reference counts when the objects move between main memory and secondary storage as opposed to when object pointers are created and destroyed. Hence, the deferring mechanism in Creepy is more efficient than Deutsch and Bobrow technique.

In the on-line mode, Creepy requires no additional disk I/O since it only collects pages which have already been read into memory. Objects which grow and cannot be stored in the current page, are forced to move to another page, leaving behind a forwarding pointer. Thus, no updates are required to the referents of the moved objects. The forwarding pointers are "short-circuited" when they are used, so the overhead of a forwarding pointers is a one-time cost. Unused forwarders are eventually "short-circuited" in the off-line operation of Creepy.

Creepy is designed for a transaction based system in which objects are first read into memory at the beginning, and written at the end of a transaction. The cost of updating the reference count when loading an object is low, requiring a simple calculation. The cost of unloading is relatively expensive since a live object traversal is performed at the end of each transaction. However, this traversal can be achieved as a side-effect of the main memory garbage collection, making the updating process essentially free.

Since Creepy examines a large collection of pages in the off-line mode, it can short-circuit the forwarding pointers created in the on-line mode and reclaim space occupied by inaccessible circular structures. As Creepy encounters an object containing a forwarder, it makes immediate updates and hence reduces pointer chasing in the on-line mode. When a large collection of related objects are moved into memory, the values in the RefCountTable indexed by these objects are likely to be zero since there are no objects on disk referencing them. Since a copying garbage collection is used, all circular structures contained within these pages are removed. Unfortunately, if two objects are inaccessible and reference each other outside of these collected pages, they are not reclaimed. However, this situation is rare since in the off-line mode, Creepy can have up to four thousand object pointers in memory².

²An average object size is 50 bytes and our machine has two megabytes of main memory.

In contrast to many virtual memory garbage collectors [3] [18] [21] where the cost of expensive disk I/O operations is hidden, Creepy explicitly recognizes and avoids these operations. As a result, Creepy avoids disk I/O by performing all of its computations in main memory. The live object traversal is performed while the objects are in memory, avoiding a disk scan during garbage collection.

Baker claimed [3] that his technique should be used instead of a dual garbage collector which combines the reference count and copying techniques. He also claimed that dual garbage collectors are not effective since the cost of updating reference counts is expensive and more stack space is required. However, when the size ratio between secondary storage and main memory is a factor of one hundred or more, Baker's algorithm performs poorly, requiring excessive amount of disk operations to swap the pages in the ToSpace and the FromSpace. Creepy's dual garbage collectors reduce the disk I/O operations by only scanning a segment of secondary storage when garbage collecting allowing a segment of storage to be reclaimed independently.

4.7 Multi-user secondary storage system

With a few minor changes, Creepy can be applied to a multi-user system. In this system, one machine is dedicated as an object server, which owns and controls the secondary storage. The other machines are clients and are connected to the server to form a network. The server and clients transfer objects via a remote procedure call mechanism [8].

Creepy's LoadedTable and RefCountTable are stored in and maintained by the server. The loading procedure described in Figure 4.6 can be applied to this system without any modification. The unloading procedure in Figure 4.8, however, requires modification to handle object replicas. When an object unloads, the server asks each client for its set of loaded referents of this object. The number N in Figure 4.8 is the size of the union set. Once the number N has been computed, the unloading procedure can be carried out as described in Figure 4.8.

therefore fourty thousand pointers can be in memory

The *premature zero setting* problem is often encountered in a multi-user system [23] [20] [32]. Suppose objects A and B are residing on processors P_A and P_B respectively, and the B is the only object referencing A . Then the reference count of A is one. If object D residing in P_D copies the A -reference in B , P_B sends an increment message to P_A . At some point after the copy of A -reference arrives at P_D , it is discarded immediately so P_D sends a decrement message to P_A . It is possible that the decrement message from P_D arrives before the increment message from P_B , making the reference count of A zero even though B is still referencing A . Creepy does not suffer from the premature zero setting problem since an object is reclaimed only if its reference count is zero and there are no other objects in main memory referencing it.

Chapter 5

Prototype Implementation

This chapter describes a prototype implementation of Creepy executing in the Sticky system. Sticky objects are stored to disk enabling Creepy to garbage collect those that are inaccessible. A brief overview of the major classes constituting the Sticky system is presented to show the functionality of Creepy as well as its integration in the system. Many design issues concerning Creepy and Sticky are discussed to show the strengths and limitations of the current implementation.

5.1 Hardware Configuration

The current Creepy prototype was designed and implemented in a single processor system. It is written in Smalltalk/V 286, therefore, any machines supported by this software can run the program. Appendix three of the **Smalltalk User's Guide** [16] [17] contains the detailed information on the hardware and the Smalltalk/V configuration for different memory environments.

The prototype works on both AT compatibles and MAC II's. Each machine requires at least two megabytes of main memory and a twenty megabyte hard-disk. However, to fully exploit the functionality provided in Creepy, the secondary storage should be much larger.

In the future, Creepy will be extended to run in a network consisting of a set of nodes. One node is a dedicated object manager which owns and controls the secondary storage. The rest of the nodes are clients of the server node and are

connected to it via Ethernet. The remote procedure call communication mechanism is employed to transfer objects from the server to its clients.

The physical location of a node is irrelevant. However, in practice, sending a message to a node in the local network is faster and cheaper than sending one to an external node. Nodes are not expected to run at the same clock speed and no clock synchronization between nodes is required. All nodes in the network can communicate with each other via message passing. A node can also broadcast its message to every node. An object may reside in one or more nodes for computation, but only one node can successfully make an update to this object.

This model was chosen since it is relatively inexpensive to build using the current technology. Moreover, the object maintenance overhead is low since the server is the only node that must ensure proper object updates.

5.2 Sticky

A complete implementation of Sticky is beyond the scope of this thesis since Sticky is only used as a testbed for Creepy. Instead, only important features of Sticky have been prototyped. However, mechanisms for other features of Sticky that are not essential to this thesis are provided so that the Sticky prototype can be enhanced in the future. In this section an overview of an implementation of Sticky is described.

The Sticky prototype consists of two components: repository and database. The database component is not essential to this thesis, hence, only a simple Transaction class is implemented.

Sticky is written in Smalltalk/V, which allows Smalltalk objects to individually persist in secondary storage instead of persisting through a snapshot of the whole object space as provided in Smalltalk/V. Objects which persist in secondary storage are called *Sticky objects* to distinguish them from Smalltalk/V objects.

5.2.1 Transaction

A transaction mechanism is employed to ensure consistency of the Sticky objects. Currently, there can only be one transaction running in Sticky at any time. A transaction maintains a *readSet*, a *writeSet* and a *referencedSet*. The *readSet* contains a list of loaded objects in a transaction. The *writeSet* holds a group of objects which have been modified in a transaction and therefore, must be updated in secondary storage. The *referencedSet* contains a list of objects that are referenced by the objects in the *readSet*. Creepy uses these sets to check for objects "in-use". Objects belonging to these three sets are not garbage collected. This eases the undo process in the event of a transaction abort.

A *Transaction* class is implemented which provides the three basic operations: *begin*, *commit*, and *abort*. The *begin* operation creates a transaction which allows subsequent manipulations on Sticky objects to be performed. Once a transaction is created, Sticky objects may be created or read from secondary storage. The *readSet* and the *referencedSet* are updated as objects are read into main memory. Objects created during the lifetime of a transaction are not Sticky objects, but may become so by one of two methods. The user can explicitly create a new Sticky object through the *becomePersistent* message. Alternatively, they are automatically transformed to Sticky objects during the *commit* operation if they are referenced by an existing Sticky object.

The *commit* operation is an atomic operation which checks for modified and newly created Sticky objects while concurrently adding them to the *writeSet*. An atomic write to secondary storage updates the objects in the *writeSet*. When all of the objects have been successfully updated, the transaction terminates.

The *abort* operation terminates a transaction without writing modified objects to secondary storage. This operation is used when a transaction detects an inconsistent object update.

5.2.2 Sticky Objects

Sticky objects are represented by instances of the `PersistentObject` class. This class provides a set of high level functions making the access to Sticky objects transparent to the application programmer. An instance of this class encapsulates a Smalltalk/V object making it a Sticky object. A Sticky object instance is called an *encapsulator* when the object it encapsulates is in main memory. It is called a *proxy* when the object resides in a secondary store. The object enclosed in an instance of the `PersistentObject` is placed in secondary storage called the *repository*. Objects in the repository are identified by their *object-location*. Objects may be moved from one node to another as part of a transaction but their object-location in a repository is not changed. Moreover, the object-locations and how they are manipulated is transparent to the application programmer. The encapsulated objects are automatically loaded into main memory when their proxy is manipulated. As this object is loaded into memory, their corresponding proxy becomes an encapsulator. An encapsulator reverts to a proxy when this object is unloaded to a repository.

The `PersistentObject` class maintains the `RefCountTable` and the `LoadedTable`, as described in Chapter 4, as objects are loaded and unloaded. The `RefCountTable` is a Dictionary keyed by the object-location. The `LoadedTable` is an instance of the `Set` class. According to the description of the algorithm in Chapter 4, both of these tables must be stored in a non-volatile memory, currently simulated by logging everything in these tables to a sequential file.

An important role of the encapsulators or proxies is message trapping. `PersistentObject` implements a *doesNotUnderstand:* method to trap any messages to the encapsulated object via its proxy or encapsulators. Instead of causing an error message to display when encountering an unknown message, the message is forwarded to the encapsulated object. The problems with message trapping are discussed later in this chapter.

Sticky objects can be created explicitly by the user by sending a *becomePersistent* method to a Smalltalk/V object. An instance of `PersistentObject` is created to encapsulate this object. Not only does the above object become a Sticky object, but

as a side-effect to the *becomePersistent* method, all objects referenced by this object are also implicitly transformed into Sticky objects. Alternatively, Sticky objects are created when a transaction commits. Any Smalltalk/V objects referenced by an existing Sticky object automatically become Sticky objects.

Before a Sticky object can be manipulated, a transaction must be created. The optimistic concurrency control mechanism is used by a transaction to ensure accessing consistency. In addition, a transaction provides a recovery mechanism when a system crash occurs. Neither the concurrency control nor the recovery mechanism is implemented in the current prototype. However, the work by Stringham [58] (currently in progress) describes these features.

5.2.3 Repository

The repository is automatically opened when entering Smalltalk, and closed on exit. Repositories are divided into pages which allow Creepy to garbage collect a portion of a repository at a time. A Sticky repository is implemented by two classes: Page and PageManager. A repository page is an instance of the Page class. An instance of the PageManager controls the page resources of a repository.

PageManager

Repositories are managed by the PageManager class. This class determines where a persistent object can be stored in the repository. Furthermore, it initiates garbage collection when a page in memory is unable to hold a Sticky object. The PageManager buffers pages currently used in active transactions to minimize disk I/O. As pages are garbage collected, the free pages are kept in a table maintained by this class.

Sticky objects are stored in a repository by a *store* operation, which puts these objects in a page of the repository. Page locality and object security are two factors which must be taken into consideration when determining object-location. However, in this prototype an object resides in the first free page. When a Sticky object is

stored in a repository, its reference count, time stamp and size are stored in its header (described in the **Page** section below).

Objects are read into main memory via a *fetch* operation, given its location in a repository. The time stamp and the reference count of this object on disk are also read when an object is fetched into memory.

The *fetch* and *store* methods are only used by the `PersistentObject` class to load and unload Sticky objects. These methods must be hidden from the application programmer to obtain seamless integration between secondary storage and main memory.

Page

A page is the smallest unit of data that can be transferred between main memory and secondary storage in every disk read or write. Each page contains its starting position (the position of the first byte in it). Associated with each page is a boolean flag which acts as a dirty bit, and is set to true when the contents of a page is modified.

A page is divided into three areas: *page header*, *object description* and *object space*. The page header contains general information concerning the page and is maintained by the `Page` class as objects are stored into or removed from it. There are three fields in a page header. The first is used to coalesce pages to store objects whose size is bigger than the physical page size. The number of pages required to store an object, excluding this page, is indicated by this field. Therefore, if an object fits in a page, zero is stored in this field. The second field contains the number of objects in a page. When this number is zero, the page is added to the free page list maintained by the `PageManager`. The third field contains an offset to the next available free space (all free spaces are chained together).

The object description area of a page contains headers of stored objects. Each object header holds:

1. the size of the object.

2. the time stamp indicating when the object was stored.
3. the reference count of the object.
4. the offset into the object space where the object resides.

An object whose size is zero indicates that it has been relocated to another page. The forwarding pointer is contained in the object space pointed to by the offset of its object header. The object-location of a Sticky object points to an object header. An object can thus be relocated anywhere within a page and only the offset field of its object header needs to be updated. The time stamp field provides a mechanism for optimistic concurrency control. The reference count field is used to determine object reachability and contains the number of objects outside of this page referencing this object header.

The object space of a page is the largest area of the three, and is used to hold objects. When a page is garbage collected, the space in this area is reclaimed and compacted.

5.3 Restrictions and Limitations

The prototype has several restrictions and limitations. For example, not all objects can become Sticky objects. Instances of `SmallInteger`, `True`, `False` and `UndefinedObject` uniquely encode their value in their object pointer. Therefore, any attempt to make these objects to become Sticky results in an error.

Another limitation is the inability to change some of the system classes since their source code is hidden or their methods are implemented as primitives. Symbol instances, for example, cannot be made Sticky due to the above reasons.

Although some objects can become Sticky objects, they lose their behavior in the process caused by bypassing the `doesNotUnderstand:` trap of the Sticky object. The methods of these objects are usually written for speed and therefore assume that a correct type has been given. `CompiledMethod` and `BitBlt` instances are examples of

this problem. A `CompiledMethod` instance can become `Sticky` but cannot be used since the `Smalltalk` interpreter assumes that it is an array of bytes.

The major problem exhibited in this prototype is the absence of synchronization between the main memory garbage collector and `Creepy`. The main memory garbage collector is forced to run for every unloaded object to calculate its reference count. Consequently, the unloading process is time consuming.

5.4 Implementation Problems

There were a number of problems encountered while implementing `Creepy` for the `Sticky` system. These were caused by the difficult nature of compaction, the handling of composite objects and object replications in a single processor system. This section discusses these three major problems in the current implementation.

The disk compaction problem is NP-complete as proved in chapter two. The most popular method for compaction is a sliding technique. This is inefficient since all objects must be relocated to eliminate disk fragmentation. Since `Creepy` employs a copying strategy to reclaim a collection of pages in the off-line mode, page fragmentation is removed. However, global disk fragmentation may still exist. This problem needs further investigation so that global disk compaction can be performed more efficiently.

The current implementation does not support composite objects as it makes no attempts to localize them. However, storing these objects leads to a significant problem. The size of such an object cannot be determined until it is actually stored in secondary storage. Unfortunately, the size is required by the `PageManager` so that an appropriate page can be assigned to store an object. This "bootstrapping" problem forces composite objects to be unloaded twice: the first time to determine its size and the second to actually store it.

Chapter 6

Conclusion

In this thesis we have presented Creepy, an incremental garbage collector for a persistent object server called Sticky. Creepy is one feature distinguishing Sticky from all other persistent object servers. By managing the allocation and deallocation of secondary storage including the reorganization of objects on disk, Creepy aims to reduce disk fragmentation.

Creepy's unique method for reclaiming unused storage is superior to existing methods for the following reasons:

- Creepy garbage collects pages independently of one another, thus avoiding the time consuming task of scanning the whole disk.
- Pages are reclaimed incrementally enabling Creepy to run concurrently with normal computation.
- Creepy's on-line mode demands little processor time since its main task is to update reference counts. Hence, the performance of the user applications is not adversely affected.
- Creepy has a low overhead cost as the reference counts are efficiently maintained. Traditional reference count techniques have a high overhead cost since an object count must be updated whenever a reference is created or destroyed. Creepy reduces this cost by updating reference counts only when objects are moved between main memory and secondary storage.

- In on-line mode, Creepy only collects pages brought into memory by the user transactions and thus does not cause unnecessary disk I/O operations. Fetched-in pages have a higher probability of containing garbage than pages on secondary storage since manipulation of the fetched-in pages tends to generate garbage. As a result, Creepy is applied more often to pages where garbage is likely to occur.
- Creepy's off-line mode takes advantage of idle processor time to perform its time consuming tasks such as compaction, removal of forward pointers and object clustering. The end result is a significant improvement in the system performance.
- Creepy provides a garbage collection method to reclaim circular structures by combining both a reference count and a copying technique.

6.1 Future work

The work presented in this thesis demonstrates that garbage collection in secondary storage is feasible. It also opens many questions which remain to be answered.

Creepy uses a copying technique to reclaim space occupied by inaccessible objects. Page compaction is accomplished as a side-effect. However, the global disk compaction (previously shown to be NP-complete in Chapter 2) remains unsolved. We believe that there exists a heuristic or stochastic algorithm which can provide a better technique for compacting than the popular sliding method.

Compaction achieved using the copying garbage collection technique neglects the object clustering issues. A statistic based model of how objects are accessed is required in order to gain high page locality and disk integrity. This complex topic requires further investigation.

The issue of system security has not been addressed in this thesis. This issue adds another dimension to the complexity of the storage compaction problem. When the garbage collector reorganizes secondary storage, objects may move from a protected

area to an unprotected one. This allows a malicious user to delete or modify others' objects. At first glance, this problem seems like it can be solved by adding a flag to an object or a page to indicate that it is not compactable. Unfortunately, this solution may require the user's assistance.

The current Sticky implementation is incomplete as it provides little transaction support. A complete implementation of Sticky should include transaction management, object sharing among multiple users, concurrency control, authorization, backup and recovery. Nevertheless, the current Sticky implementation provides the necessary foundation to handle these issues.

6.2 Summary

This thesis presents explanations as to why many of the existing main memory garbage collection techniques are unsuitable for secondary storage. A new algorithm for garbage collection in secondary storage has been described. Sticky, a persistent object server, was prototyped as a test bed for this algorithm. During the development of Creepy and Sticky, many questions concerning secondary storage garbage collection and persistent object servers were raised. Some are answered in this thesis, others remain unanswered and are listed in the Future work section.

6.3 Postscript

Since this thesis was first written, further work on the Sticky system has been conducted by James Stringham [38]. The work of Stringham addresses the issues of transaction management, concurrency control, object sharability, composite objects, and recovery.

References

- [1] M. Atkinson, *Types and Persistent Objects in Database Programming Languages*, ACM Computing Surveys, Vol. 19, Number 2, June 1987, pp. 105-190.
- [2] M. Atkinson, K. Chisholm and R. Morrison, *PS-Algol Papers*, Technical Report PPRR-2, Department of Computing Science, University of Glasgow, 1986.
- [3] H.G. Baker, *List Processing in Real-Time on a Serial Computer*, Comm. ACM, Vol. 21, Number 4, April 1978, pp. 280-294.
- [4] J.M. Barth, *Shifting Garbage Collection Overhead to Compile Time*, Comm. ACM, Vol. 20, Number 7, July 1977, pp 513-518.
- [5] P.A. Bernstein, V. Hadzilacos and N. Goodman, **Concurrency Control and Recovery in Database Systems**, Addison-Wesley Publishing Company, Don Mills, Ontario, 1987.
- [6] P.B. Bishop, *Computer Systems with a Very Large Address Space and Garbage Collection*, Ph.D Thesis, May 1977, MIT/LCS/TR-178.
- [7] A. Björnerstedt, *Secondary Storage Garbage Collection for Decentralized Object-Based Systems*, **Object Oriented Development**, University of Geneva, July, 1989.
- [8] A. D. Birrell and B. J. Nelson, *Implementing Remote Procedure Calls*, ACM Transactions on Computer Systems, Vol. 2, No. 1, February, 1984, pp. 39-59.
- [9] J. Campin and M. Atkinson, *A Persistent Store Garbage Collector with Statistical Facilities*, Technical Report PPRR-29, Department of Computing Science, University of Glasgow, 1986.
- [10] R.G.G. Cattell, *Design and Implementation of a Relationship-Entity Datum Data Model*, Xerox Corporation, 1983.
- [11] D. W. Clark and C. C. Green, *An Empirical Study of List Structure in LISP*, Comm. ACM, Vol. 20, Number 2, February 1977, pp. 78-87.

- [12] D.W. Clark, *Measurements of Dynamic List Structure Use in Lisp*, IEEE, Vol. 5, Number 1, January 1979, pp. 51-59.
- [13] J. Cohen, *Garbage Collection of Linked Data Structures*, ACM Computing Surveys, Vol. 13, No. 3, September 1981, pp. 341-367.
- [14] G. E., Collins, *A method for overlapping and erasure of lists*, Comm. ACM, Vol. 3, Number 12, December 1960, pp. 655-657.
- [15] Robert, Courts, *Improving Locality of Reference in a Garbage Collecting Memory Management System*, Comm. ACM, Vol. 31, Number 9, September 1988, pp. 1128-1138.
- [16] **Smalltalk/V 286 Users Guide**, Digitalk Inc. Los Angeles, 1988.
- [17] **Smalltalk/V MAC Users Guide**, Digitalk Inc. Los Angeles, 1988.
- [18] L.P. Deutsch and D.G. Bobrow, *An Efficient, Incremental, Automatic Garbage Collector*, Comm. ACM, Vol. 19, Number 9, September 1976, pp. 522-526.
- [19] J. Duimovich, *Garbage Collection in a Multiprocessor Smalltalk System*, Master Thesis December 1988, Carleton University, School of Computer Science.
- [20] J.D. Eckart and R.J. LeBlanc, *Distributed Garbage Collection*, Sigplan'87 Symposium on Interpreters and Interpreter Techniques, St. Paul, Minn., June 24-26, 1987, Sigplan Notices, Vol. 22, Number 7, pp. 264-273.
- [21] R. Fenichel and J. Yochelson *A LISP garbage-collector for virtual-memory computer systems*, Comm. ACM, Vol. 12, Number 11, November 1969, pp. 611-612.
- [22] M.R. Garey and D.S. Johnson, **COMPUTERS AND INTRACTABILITY: A Guide to the Theory of NP-Completeness**, Bell Telephone Laboratories, Incorporated, 1979.
- [23] B. Goldberg, *Generational Reference Counting: A Reduced-Communication Distributed Storage Reclamation Scheme*, Sigplan '89, Conference on Programming language designed and implementation, Portland, Oregon, June 21-23, 1989, Sigplan Notices, Vol. 24, Number 7, pp.312-321.
- [24] D.M. Harland and Bruno Beloff *OBJEKT - A Persistent Object Store With An Integrated Garbage Collector*, Sigplan Notices, April, 1987.
- [25] T. Kaehler, *Virtual Memory on a Narrow Machine for an Object Oriented Language*, OOPSLA '86 Proceedings, Portland, Oregon.
- [26] D.E. Knuth, **The Art of Computer Programming Vol. I-III**, Second Edition, Addison Wesley, Reading Mass., 1968.

- [27] S. Ballard and S. Shiron, in G. Krasner, Ed. **Smalltalk-80 Bits of History, Words of Advice**, Addison-Wesley, Reading Mass., September 1983. pp. 127-150.
- [28] J. Konik, *Optical Storage System Boosts Capacity to the Terabytes*, Computer Technology Review, August, 1989.
- [29] W. Lalonde, J. McGugan and D. Thomas, *The Real Advantages of Pure Object-Oriented Systems or Why Object-Oriented Extensions to C are Doomed to Fail*. IEEE, 13th Annual International Computer Software & Applications Conference (COMPSAC '89), Orlando, Florida, September 1989, pp. 344-350.
- [30] W. Lalonde and J. Pugh, *Disk Forms*, Journal of Object-Oriented Programming, October 1988, Vol. 1, Number 4, pp. 50-54.
- [31] B. Lang and F. Dupont, *Incremental, Incrementally Compacting Garbage Collecting*, Sigplan'87 Symposium on Interpreters and Interpreter Techniques, St. Paul, Min., June 24-26, 1987. Sigplan Notices, Vol. 22, Number 7, pp.253-263.
- [32] C.W. Lermen and Dieter Maurer, *A protocol for distributed reference counting*, ACM Conference on Lisp and Functional Programming Proceeding, August, 1986.
- [33] H. Lieberman and C. Hewitt, *A Real-Time Garbage Collector Based on the Lifetimes of Objects*, Comm. ACM, Vol. 26, Number 6, June 1983 pp 419-429.
- [34] J. McCarthy, *Recursive Functions of Symbolic Expressions and Their Computation by Machine*, Comm. ACM, Vol. 3, No. 4, pp 184-195, April 1960.
- [35] J. McCarthy et al., *LISP 1.5 Programmers Manual*, MIT Press, Cambridge Mass, 1965.
- [36] D. Moon, *Garbage collection in a large Lisp system*, ACM Symposium on Lisp and Functional Programming, Austin, Texas, 1984, pp. 235-246.
- [37] J. Pallas, and D. Ungar, *Multiprocessor Smalltalk: A Case Study of Multiprocessor-based Programming Environment*, In proceedings of the SIGPLAN 88 Conference on Programming Language Design and Implementation, SIGPLAN, Vol. 23, Number 7, July 1988, pp 268-277.
- [38] J. Stringham, *Persistent Objects*, Master Thesis December 1989, Carleton University, School of Computer Science.
- [39] D.A. Thomas, W.R. Lalonde, J.R. Pugh, *Actra - A Multitasking-Multiprocessing Smalltalk*, Technical Report SCS-TR-92, Carleton University, May 1986.

- [40] D.A. Thomas, W.R. Lalonde, J.S. Duimovich, *Efficient Support for Object Mutation and Transparent Forwarding*, Technical Report SCS-TR-128, Carleton University, November 1987.
- [41] D. Ungar, *Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm*, ACM SIGSOFT-SIGPLAN Practical Programming Environments Conference, Pittsburgh, PA, April 1984.
- [42] D. Ungar, **Evaluation of a High Performance Smalltalk System**, MIT Press, Cambridge, Mass., 1986.
- [43] Kyu-Young Whang, *A Seamless Integration in Object-Oriented Database Systems*, Proceeding Fifth International Conference on Data Engineering, February 1989, Los Angeles, California.
- [44] P.R. Wilson, and T.G. Moher, *Design of the Opportunistic Garbage Collector*, OOPSLA '89 Proceedings New Orleans, Louisiana.
- [45] D.S. Wise, *Morris's Garbage Compaction Algorithm Restores Reference Counts*, Toplas, Vol. 1, Number 1, July 1979, pp115-120.
- [46] W. Wolf, *A Practical Comparison of Two Object-Oriented Languages*, IEEE Software, September 1989, pp. 61-68.

Appendix A

Glossary

Throughout this thesis, words and phrases have been printed in boldface and italicized text to represent terms and concepts of special importance. These words are commonly use in garbage collection and persistent object literature. A summary of the abbreviations, acronyms and special terms used in this thesis are provided below for the readers unfamiliar with these areas.

Consistent state A state of the database that satisfies the database's consistency predicates. Intuitively, this means that data item values are internally consistent with each other.

Garbage Collector An application which automatically reclaims space of unused objects and compacts storage to prevent fragmentation.

Dangling Reference A reference which refers to an object that has been destroyed and has had its space reclaimed.

Forwarder A "tombstone" for an object stored elsewhere in memory. It provides a fast mechanism to update references to moved object without scanning of memory [40] [30].

Immediate reference object An object, B, is an immediate reference object of A if A has a reference pointing directly to B.

Mutator A process which modifies the graph of live objects as it performs computations which makes some objects unreachable.

Object An autonomous entity which has state and behaviour. Its data is hidden from other objects and can only be accessed through the behaviour provided by an object.

Persistent Object An object created during one session which resides on secondary storage and remains available for other sessions even if the session creating it has terminated [1].

Root Set An external, well known collection of objects such as the run time stack which is used to trace out the set of live objects. This set may be represented as a directed graph with the objects as nodes and pointers to other objects as the directed edges.

Referent An object, A, is a referent of an object, B, if A references B.

Seamless An integration between application programs, operating system and secondary storage is seamless if object transitions are completely transparent to the programmer [43].

Stable storage An area of memory that is resistant to processor and operating system failures. It models secondary storage media, such as disk and tape, on typical computer system [5].

Thrash A situation where increasing the number of transactions in the system causes the throughput to drop.

END

1 9|0 2|9 1

FIN