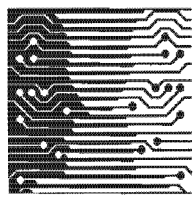
GARBAGE COLLECTION OF THE PLAVA OBJECT STORE

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE,
FACULTY OF SCIENCE
AT THE UNIVERSITY OF CAPE TOWN
IN FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

By
Michael F. Schulz
November 2002

Supervised by
Dr S. Berman



"There are no rules concerning disposal of inaccessible addresses"

— The Definition of Standard ML [MTH89]

Abstract

This research investigates the implementation of suitable garbage collection schemes for a persistent store. The store that formed the basis for these experiments was the one developed for a Java Virtual Machine known as PLaVa, designed by Stephan Tjasink at the University of Cape Town [Tja99].

Two garbage collection schemes were implemented: semispace copying and a partitioned collection scheme due to Maheshwari and Liskov [Mah97, ML97]. Semispace copying was implemented to gain background knowledge of how the PLaVa store operates and how a simple garbage collection scheme could be developed for it. This work is then extended by implementing an incremental garbage collection scheme. Both implementations required modifications to the PLaVa store, in order to support semispace and partitioning algorithms. The partitioned collection scheme in particular required almost a complete re-implementation of the store.

To evaluate the implemented garbage collection schemes, a synthetic application was developed, which allowed the fine tuning of specific parameters. This facilitated the evaluation of specific features and mechanisms used in both schemes, so that bottlenecks within the implementation could be determined.

The evaluation results show that both garbage collection schemes are suitable for small to medium stores with reasonable amounts of idle time. Semispace copying overhead is linearly proportional to the amount of live data that exists during copying (with store size having far less of an impact), but requires stores to be twice as large. Incremental garbage collection using the partitioned store method performs well under most store configurations, but becomes an ever increasing bottleneck as the number of inter-partition references increases. It is thus sensitive to the object placement scheme being used as well as the partition selection policy.

Acknowledgments

Thanks to the following individuals, without whom this thesis would never have happened :

My parents who made my university career possible by supporting me financially throughout the years.

My supervisor, Dr Sonia Berman, who patiently let me decide a topic, who encouraged and motivated me during dark days, and whose advice was invaluable.

My fiancé Collette who stuck with me through thick and thin, who constantly motivated me and who could at this stage write her own Masters degree about garbage collection.

Stephan Tjasink, who took the time to meet with me when he was back in the country, and help clarify issues concerning PLaVa.

John Leuner, who embarked on similar work regarding PLaVa and whose correspondance was extremely helpful.

And to all those other "victims" that I managed to somehow get involved in this research.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the late 20^{th} century we have witnessed the evolution of computer programming languages to advanced and powerful programming tools based on different abstractions and methodologies. Most of these programming languages make use of a memory heap to manipulate dynamic data structures. Heap memory is limited in size and may therefore become exhausted. A common problem among programming languages is how heap memory that is no longer accessible may be reclaimed for reuse. This is often a requirement, as the heap may run out of memory causing subsequent heap allocations to fail.

One mechanism of freeing heap memory is to swap particular sections to another memory source, for example virtual memory. The swapped memory is then loaded back into the heap as it is required. Heap memory may also be freed by reclaming the memory that is no longer accessible from the executing program. Memory that is not longer accessible is known as *garbage*, and therefore the process of reclaiming inaccessible memory is known as *garbage collection*. Many garbage collection techniques exist, suitable for different environments and architectures.

Later programming languages became more powerful with the introduction of orthogonal *persistence*. Such persistent programming languages allow for their entire execution state to be stored on a disk or other permanent storage devices. The storage used by these persistent languages is known as the store, and is usually much larger than memory heaps. As with any memory, a store may become exhausted. In this case the store may be increased in size or inaccessible data on the store may be reclaimed to allow allocation to continue. In this way garbage collection became applicable to persistent environments as a means of freeing space on the store.

The notion of persistence and garbage collection of persistent environments began a new area of research. Since heap garbage collectors preceded secondary storage garbage collectors, most persistent garbage collection techniques are based on heap memory reclamation techniques. The architectures of some persistent environments, for example distributed enviroments, necessitate changes to ordinary heap collection techniques. As a result much research and development of garbage collection in persistent environments has occured [KLW89, FCW89, HM92, ONG93, CWZ94, FS95, FS96, ML97, Mah97, HMMM97, MBMM98].

1.1 Motivation and Aim

In 1998, Stephan Tjasink [Tja99] developed a light-weight persistent Java Virtual Machine called PLaVa at the University of Cape Town. He concentrated on making the machine as small as possible so that it would require little memory for execution. PLaVa was tested using a simple store created specially for this purpose. Tjasink did develop a mark and sweep garbage collector for the PLaVa heap, but his store was basic and contained no space reclamation mechanisms whatsoever.

The aim of this research was to develop and evaluate garbage collection mechanisms for this store. In particular, our aims included :

- to research garbage collection techniques for persistent stores.
- to redesign the PLaVa store in order to develop both a conventional and an incremental garbage collector for it.
- to develop a test application so that the implemented garbage collection techniques may be evaluated.

Note that the store modifications and the implementation of the garbage collectors for the PLaVa store do not presume to run specifically on small machines. Therefore we do not concentrate on using minimal memory in our implementation, but rather attempt to develop successful and efficient storage reclamation techinques.

1.2 Approach

This research began by studying literature concerning garbage collection techniques for heap and secondary storage management. This gave us a sound foundation on the ideas behind garbage collectors for persistent stores and their implementations. We then chose to implement and evaluate two different garbage collection schemes: a simple copying collector and a partitioned garbage collection scheme due to Maheshwari and Liskov [Mah97, ML97]. This gave us two stores to use in future development of our persistent systems - one with a simple structure and minimal store overheads, the other with the ability of being garbage-collected incrementally over time.

The main idea in implementing a semispace copying garbage collector for the PLaVa store was to gain understanding of how PLaVa and the PLaVa store operate, so that a partitioned garbage collection scheme could be implemented at some later stage. The implementation of semispace copying involved alterations to the existing PLaVa class descriptors, persistent identifiers and the PLaVa store interface. On completion of this implementation we thus had a simple store with a simple mechanism for reclaiming unused space. This could be useful for applications using small to medium size stores which have enough idle time to permit regular execution of a garbage collector that runs over the entire. At the same time it server to provide a point of comparison for the other incremental collector.

We next developed a partitioned store and implemented Maheshwari and Liskov's partitioned garbage collection scheme. With the partitioning of the store, inter-partition references are brought about. Keeping track of inter-partition references so that the roots of a partition may be determined seems less problematic than it really is. If the number of inter-partition references on the store becomes high, the

work required to maintain inter-partition reference state on the store increases severely. We implement Maheshwari's novel techniques for handling inter-partition references and reclaiming inter-partition cyclic garbage, and their algorithms for garbage collecting individual partitions independently, since only when a single partition is collected independently do the subtleties in the scheme become apparent.

To complete the study, we evaluate the implemented garbage collection schemes using a synthetic database application. By using an artificial application, parameters like the number of inter-partition references and the amount of inter-partition cyclic garbage may be independently controlled. Thus the effect of these conditions on garbage collection overhead may be determined.

1.3 PLaVa Background

This section gives the reader an insight into the Persistent Light-weight Java Virtual Machine (PLaVa), the Java Virtual Machine for which out garbage collection algorithms are implemented. PLaVa was implemented by Stephan Tjasink at the University of Cape Town [Tja99], and is an extension to a conventional Java Virtual Machine implemented by Tjasink himself [TB98]. The virtual machine was designed to be portable and to execute on small machines with limited memory, specifically for installation in ROM on a Digital Satellite Television (DSTV) decoder.

With PLaVa, Tjasink investigated the feasibility of running an orthogonally persistent version of Java on small machines with limited memory. His work details implementation decisions in providing persistence, including heap memory management, object faulting, caching and the swapping of reachable objects to the persistent store, in order to make more heap memory available.

Tjasink tested PLaVa using PJSL [PAD+97], the persistent store developed for PJama. He also implemented his own store, which was unpartitioned and simply appended promoted objects to the next available location on the store. Although the virtual machine provides heap memory management in the form of a standard mark and sweep garbage collector, the simple store has no garbage collection mechanism.

1.3.1 PLaVa Memory Management

PIDs are assigned to objects that are promoted, and are represented as the negated offset of the object location on the store. PIDs are negative to distinguish them from memory addresses and are 32 bits in length, making them the same size as memory addresses. This facilitates the swizzling of memory references to PIDs and vice versa [Tja99].

Tjasink implements a Resident Persistent Object Table (RPOT) which stores an object's PID to local address (LA) mapping. When an object is faulted into the heap, its PID to LA mapping is stored in the RPOT, so that future residency checks will find it in the RPOT. A complementary table called the Temporary Persistent Object Table (TPOT) is used for transient (non-persistent) objects that are swapped out of the heap. They are assigned PIDs just like persistent objects, and contain object PID to LA mappings. Both the RPOT and TPOT ensure that objects can be located, whether they reside on the store or in main memory. The heap garbage collector is a standard mark and sweep collector and is invoked when heap allocation fails due to lack of space.

1.3.2 The PLaVa Store

In the design of PLaVa, Tjasink concentrated on the persistent JVM implementation, and therefore only a simple store was built for the virtual machine. The store contained minimal information in its header and objects that persist are simply appended to the store. Table 1.1 describes the information contained in the store header and Figure 1.1 illustrates the store organization.

`firstavail`	stores the value of the next free space on the store. New objects that persist use `firstavail` and are appended to the store
`root PID`	contains the PID of the root `PStore` object

Table 1.1: Original PLaVa store header information.

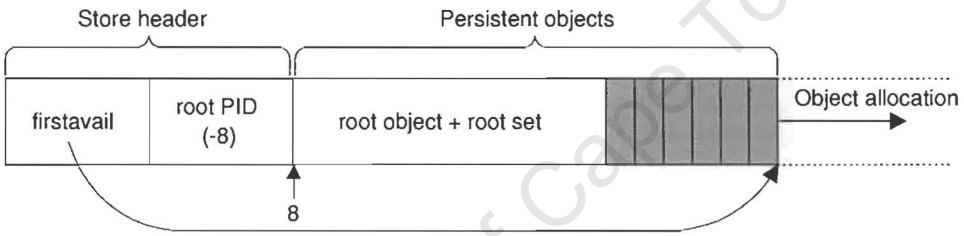


Figure 1.1: The original PLaVa store organization.

When a store is created, a `PStore` object is initialized and written to the store at `firstavail` (position 8 on the store referring to Figure 1.1). Figure 1.2 shows the `PStore` and `PRoot` class structure. `PRoot` objects contain a `String` object for uniquely identifying a root object and an `Object` reference to that root object. `PStore` objects contain a `roots` array used for storing references to persistent roots on the store, a boolean for setting the state of the store (open or closed), an integer `maxNumRoots` defining the maximum number of roots (initially set to 5), and an integer `numRoots` that keeps track of the number of root objects.

The `PStore` class is equipped with functions to add and retrieve persistent roots from the store, as well as a stabilizing function to promote and/or update objects reachable from the persistent roots. Tjasink did not include functions to create, open, and close stores, and instead stores were specified via command line options [Tja99]. In the following chapter in Section 3.3 we extend the `PStore` class to contain functions for creating, opening and closing stores. When more than 5 root objects are added to the `PRoot` array, the array size is doubled to accomodate more root objects. After root objects have been added to the root set (the `PRoot` array) and the store is stabilized, all new objects reachable from the root set are appended to the store at `firstavail`, which is updated to reflect object promotion. When objects are written to the store, PLaVa is responsible for *swizzling* any references contained in them to the PIDs of the objects they reference.

No garbage collection was implemented, so the store simply continued growing as new objects were promoted and the TPOT objects swopped out. We chose to rectify this by implementing a copying

```
public class PStore
    {
     private PRoot roots[];
     private boolean storeOpen = false;
     private int maxNumRoots = 5;
     private int numRoots;

     public void addRoot(String n, Object o);
     public Object getRoot(String name);
     public void stabilizeAll();
    }
```

```
public class PRoot
    {
     String name;
     Object obj;

     public PRoot(String n, Object o);
    }
```

Figure 1.2: The PStore and PRoot classes.

garbage collection, as dicussed in the following chapter, and Maheshwari's partitioned garbage collection algorithm, described in detail in Chapter 4.

1.4 Organization of this Thesis

The remainder of this thesis is organized as follows :

Chapter 2 provides an overview of the literature relevant to the garbage collection of persistent storage. We discuss the aims of garbage collection and examine common garbage collection techniques, as well as the theory of persistence and persistent programming languages. We then explain garbage collection in the context of persistent object stores and present some background on persistent store garbage collectors.

Chapter 4 deals with the design and implementation of a semispace copying garbage collector for the PLaVa store. We discuss modifications made to the store to support semispace copying, and explain the implementation of depth-first and breadth-first copying algorithms.

Chapter 5 presents the implementation of a partitioned garbage collection algorithm based on Maheshwari and Liskov's partitioned garbage collection technique [Mah97, ML97]. We describe modifications made to PLaVa to support partitioned garbage collection, as well as the implementation of data structures required for partitioned garbage collection.

Chapter 6 evaluates the implemented garbage collectors by measuring specific parameters. A synthetic database application is used to generate stores with different configurations, so that the stores may be garbage collected to determine the effect of the store configuration on garbage collection overhead.

Finally, Chapter 7 concludes by providing a summary of the implemented stores and garbage collectors along with our results and achievements.

Chapter 2

Background and Related Work

This chapter provides an overview of relevant literature and implementations concerning the garbage collection of persistent object stores. We provide reviews on heap garbage collection schemes, and how these schemes are implemented to suit garbage collection in persistent evironments. The chapter is organized as follows :

- Section 2.1 describes the terms and definitions used throughout the thesis.
- Section 2.2 discusses the aims of garbage collection.
- Section 2.3 reviews several well-known garbage collection techniques.
- Section 2.4 introduces *persistence* and persistent programming languages.
- Section 2.5 deals with garbage collection in the context of persistent environments.
- Section 2.6 reviews garbage collection techniques for persistent stores.

2.1 Terms and Definitions

Throughout this thesis we use the term *object*, whereby we mean a continuous block of memory that forms a single logical structure. Objects form the unit of allocation and deallocation.[1] We define an *object graph* to be a set of objects, with a set of references (edges) connecting the objects (refer to Figure 2.1 on page 8 as an example). For the purpose of describing garbage collectors, systems are divided into the *mutator* or *program* and the *garbage collector*. The mutator is responsible for allocating objects and may *mutate* the connectivity of objects within the graph, while the garbage collector is responsible for reclaiming storage that is no longer in use.

We define an object to be *persistent* if it continues to exist once the program that created it has been unloaded. Persistent data resides on disk in some sort of *store* or database. The object from which all objects in the graph are ultimately descendants of is known as the *root*. An object is *reachable* from the root if it is referenced by a root, or is referenced by a reachable object, i.e. if it can be reached from the root by following references. In persistent systems, *stabilizing* is the process of writing new or

[1] In the context of (Persistent) Java and this thesis, we refer to *both* objects and arrays as objects, i.e. we do not distinguish between objects and arrays unless otherwise stated.

mutated objects that are reachable from the persistent root to the store. New objects are *promoted* to the store, while existing objects that have been mutated are *updated*. When objects are promoted, *persistent identifiers* (PIDs) are assigned to each object, so that objects on the store may be uniquely identified.

Persistent systems rely on the theory of persistence by reachablity (PBR) [ABC+83, MABD88], which ensures that all objects reachable from persistent objects themselves become persistent. In particular, all objects reachable from a persistent root become persistent. All objects reachable from the root object are considered *live*, while unreachable, and therefore unusable, objects are known to be *garbage*.

Garbage collection involves determining which objects are live in the object graph, reclaiming those that are unreachable. Collectors may be classified into 2 categories according to how they detect garbage : *direct* methods detect garbage directly, by maintaining information about the liveness of all objects. *Indirect* or *tracing* garbage collectors determine the liveness of objects indirectly, without the use of information associated to objects, by deriving liveness from other objects. This is usually achieved by performing a *trace* or *scan* of the object graph. We will see examples of both categories in Section 2.3.

2.2 The Aims of Garbage Collection

In many programming languages, deallocating memory so that is may be reused is the programmer's responsibility, for example the programming language C [Ler93]. This is tedious and may result in *dangling references* and *memory leaks*, as shown in Figure 2.1. A dangling reference occurs when a surviving reference points to an object that no longer exists at that address, and may cause programs to crash or cause segmentation faults. Memory leaks are a result of not freeing allocated memory, although it is never used again. In other words memory is wasted, since the programmer has left it unfreed.

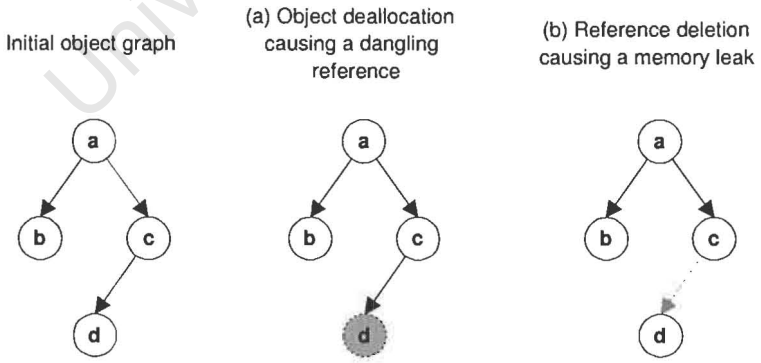


Figure 2.1: Dangling references and memory leaks.

The prinicple aim of garbage collection is to *automatically* reclaim allocated memory or storage that is no longer is use, thereby relieving the programmer from this burden. However, this is not the sole responisiblity of garbage collection. According to Moss *et. al* [MBMM98], garbage collectors, while freeing unused memory, should reorganize data in an attempt to improve performance.

2.3 Garbage Collection Techniques

This section describes several well-known garbage collection techniques, which apply to heap garbage collection. Accorinding to [But87, AFG95, Pri00], traditional garbage collection techniques developed for programming languages are not directly applicable in a database or persistent store context. Database Management Systems (DBMS) introduce complications not addressed in primary memory implementations, which include atomic transactions, disk Input/Output (I/O), concurrency and fault tolerance etc, as will be discussed in Section 2.6.

However, some are applicable and suitable for secondary storage (secondary memory) garbage collection. Each garbage collection technique is briefly described, including the benefits and drawbacks of each method. Wilson [Wil92] and Jones and Lins [JL99] provide excellent surveys on garbage collection algorithms, should the reader require more details. Distributed collection techniques are not described in this section, but discussed later in Section 2.6.2 in the context of persistent storage.

2.3.1 Reference Counting

Reference counting is a *direct* garbage collection technique which was originally implemented by George Collins [Col60] for Lisp in 1960. The principle of reference counting is to associate a count with each object, indicating the number of references to it. The algorithm maintains a *reference count invariant* which requires an object's reference count to eqaul the number of other objects referencing it. When an object's reference count decreases to zero because of a reference deletion or modification, it may be assumed to be garbage since it is not reachable from any objects, and may be collected immemdiately by appending it to a *free list.*[2]. In other words, reference counting simply keeps track of whether objects are in use (reference count greater than one) or not (reference count equal to zero).

Figure 2.2 illustrates a reference counting example. (1) shows the initial object graph, and (2) shows the reference count modifications after the reference from object a to b is deleted. Note that objects d and g comprising the garbage cycle are not reclaimed, since their reference counts can never be decreased to zero, because neither object is reachable. Further, objects that become garbage (shaded objects) may cause a *cascade* of decrementing reference counts to occur, for example the deletion of the reference from a to b causes the references from b to e and b to f to be deleted. This cascading effect may become severe in cases when an object that contains many object references and is close to the object graph's root, becomes garbage.

Many languages and applications have adopted the reference counting algorithm for storage management, for example Modula2+ [DeT90] and Smalltalk [GR83]. The Unix operating system also makes use of reference counting to determine whether a file may be deleted from the disc. Reference counting is often used because it can be implemented without any support from the language or compiler. Reference counting garbage collection implementations for persistent stores include Yong *et. al.* [YNY97] and Butler [But87].

[2]Note that the root object of a object graph will always have a reference count of zero, since no other object references it, however we do not classify the object as garbage

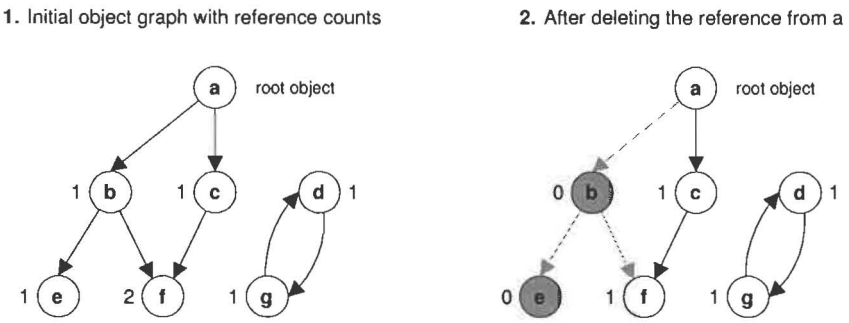


Figure 2.2: The reference count garbage collection technique.

Benefits and Drawbacks

Reference counting is advantageous when considering its incremental nature, its "spatial *locality of reference* is likely to be no worse than that of its client program" [JL99], and objects are reclaimed as soon as they become garbage. The pause times caused by updating reference counts are typically fairly short, unless the updating of a single reference causes severe cascading to occur.

On the other hand, reference counting suffers from a high processing cost incurred in maintaining the reference count invariant by updating and checking counters, not to mention its inability to reclaim cyclic garbage (e.g. objects *d* and *g* in Figure 2.2). Further, space within objects is required for the storage of the reference count, and reference count garbage collection tends to fragment live objects and the free space reclaimed, due to the nature of the reclamation process. Fragmentation increases the work required to allocate (in heap garbage collection) and promote (in persistent object store garbage collection) objects, because suitably sized free space blocks must be found to accommodate the objects [Wil92].

Despite the drawbacks of referencing counting, adaptations to reduce these drawbacks exist. *Deferred reference counting* [DB76] aims to reduce the overhead incurred by reference count maintenance by deferring reference count updates. The space required for storing reference counts within objects may be reduced by using smaller reference count fields at the cost of taking precautions to handle counts that overflow. Wise [WF77, Wis93] seems to be the limited-field reference count specialist, while other limited-field methods include the research of Stoye *et. al.* [SCN84] and Chikayama and Kimura [CK87]. As far as the inability to collect cyclic garbage is concerned, [Knu73, DB76] suggest combining reference counting with a tracing garbage collector, which is invoked periodically to remove cyclic garbage.

2.3.2 Mark-Sweep Garbage Collection

In 1960, McCarthy [McC60] devised the first algorithm for automatic memory reclamation, developed for memory management in Lisp. His scheme was a tracing garbage collector that made use of the *mark-sweep* or *mark-scan* method.

Mark-sweep is an *indirect* garbage collection technique, since it regenerates the set of reachable objects when tracing before memory is swept. Mark-sweep collection is a kind of tracing garbage collection that operates by marking reachable objects, then sweeping over memory and recycling objects that are unmarked (which must be unreachable), appending them to a free list. *Mark bits* associated with each

object are used to record whether an object was reached by the marking trace or not.

Mark-sweep methods, like other indirect collection techniques, are based on Wilson's *Two Phase Abstraction* [Wil92], as described below :

1. Garbage detection : this phase is responsible for distinguishing the live objects from the garbage. Mark-sweep methods achieve this by performing a trace (usually depth-frist or breadth-first) of the object graph from the root set, marking the objects it reaches. Unreachable objects remain unmarked.

2. Garbage reclamation : this phase is responsible for reclaiming garbage objects. Mark-sweep collection achieves this by sweeping (i.e. exhaustively examining) the memory and reclaiming the unmarked objects by appending them to a free list. The sweep phase is also responsible for clearing the mark bits of the live objects in preparation for the next garbage collection trace.

Mark-sweep's two phase abstraction is illustrated in Figure 2.3, which shows an object graph before (1) and after (2) mark-sweep garbage collection. Objects unreachable (i.e. garbage objects) after the marking trace are swept to reclaim the memory they occupy, indicated by white objects. Note that the garbage cycle comprising the unmarked objects h and j is reclaimed.

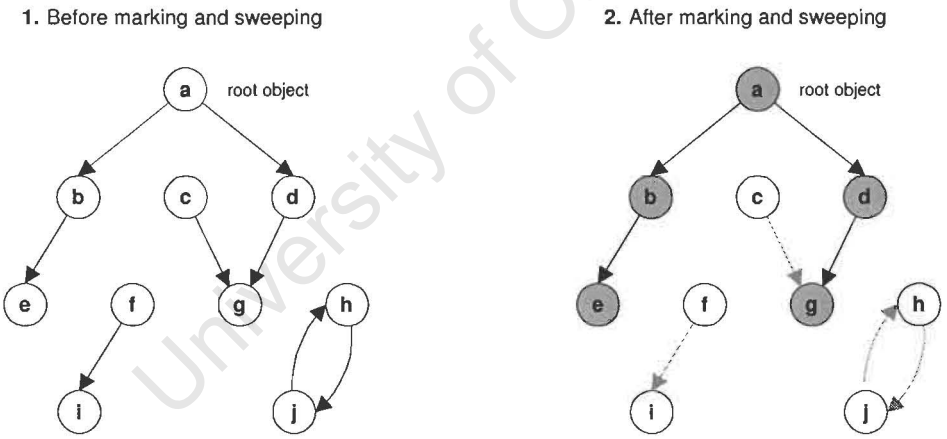


Figure 2.3: The mark-sweep garbage collection technique.

Mark-sweep algorithms are regularly used as backup garbage collectors to reclaim cyclic data structures, for example Modual-2+ [DeT90]. Mark-sweep techniques are used for heap garbage collection in Lisp [McC60], for which the algorithm was originally designed. Familiar languages like C and C++ use the Boehm-Demers-Weiser mark-sweeping garbage collector [BW88, Boe93] for heap management. Novel ideas for increasing the efficiency in mark and sweep phases are described in detail in [JL99], and include techniques like pointer reversal, bitmap marking, and lazy sweeping.

Mark-sweep techniques have also been implemented in persistent environments. Yong *et. al.* [YNY97] present two variants of conventional mark and sweep in their client-server persistent store, namely server-oriented halting mark and sweep, and incremental mark and sweep. Butler [But87] implemeted mark and sweep algorithms for an object-oriented database system.

Benefits and Drawbacks

An attractive feature of the mark-sweep algorithm is the ability to collect cyclic garbage. In fact all tracing garbage collectors are able to reclaim cyclic garbage. Further, mark-sweep techniques place no overhead on pointer manipulations, compared to reference counting which requires the maintenance of a reference count invariant.

On the other hand, mark-sweep collection requires two passes over the object graph to successfully reclaim garbage. Mark-sweep is not incremental in nature, since processing must be halted while the mark-sweep garbage collector is running. Garbage collection algorithms that halt processing during collection are known as *stop/start* collectors. The work required to complete a mark-sweep garbage collection is proportional to the number of live *and* garbage objects present in the object graph [JL99]. The reason for this is that all live objects are visited by the marking phase (to be marked), and all live and garbage objects are visited in the sweep phase (to be unmarked and reclaimed respectively). Further, mark-sweep techniques tend to fragment the memory being garbage collected, since the sweeping phase usually simply appends the reclaimed free space to a free list.

2.3.3 Mark-Compact Garbage Collection

Mark-compact garbage collectors rememdy the fragmentation drawbacks of the previous two schemes by compacting the live objects once they have been traced. Mark-compact collection may be regarded as a variation of mark-sweep collection, with extra work required to eliminate fragmentation. As with mark-sweep garbage collection, the mark-compact technique traverses the object graph marking all object reachable from the root set. Once complete, live objects are compacted into a contiguous area, while free space is "pushed" to another contiguous area. In general, compacting collectors have three phases, namely :

1. Mark phase: the collector traces the object graph from the root object and marks objects it reaches to distinguish them from garbage objects which remain unmarked. This usually involves a depth-first or breadth-first marking traversal.

2. Compact phase: the live (marked) objects are compacted into a contiguous area of memory, by calculating new locations for the objects and relocating them.

3. Reference update phase: this final phase updates pointers to reference the compacted objects at their new locations.

Compaction algorithms are classified according to the relative positions or ordering of the objects after compaction. These are described below with the use of examples :

Arbitrary Compaction : compactors are arbitrary, if objects are compacted without regard to their original ordering. The Two-Finger algorithm due to Edward [Sau74] is an example of arbitrary compaction. It makes use of `free` and `live` pointers to sweep from separate ends of the memory towards the center. `live` scans for live objects, and when found moves them into holes discovered by `free`, as `free` scans for free space. The compaction completes when the two pointers meet. Refer to [Sau74, JL99] for more details.

Linearising Compaction : linearising algorithms compact objects that originally referenced one another into adjacent positions. Examples of linearising compactors are copying collectors that trace and copy objects in the graph in depth-first fashion, for example the recursive copying algorithm presented by Fenichel and Yochelson [FY69].

Sliding Compaction : objects are slid to one side of memory, effectively "squeezing" out the free space and maintaining the original order of objects. The Lisp 2 algorithm is an example of a sliding compactor [JL99], which uses forwarding addresses stored in objects to modify the references to compacted objects. The algorithm has three phases : compute the new address of objects storing them in the forward address field within each object, update object references using the forwarding addresses, and finally move the objects to their new locations and clear the forwarding address for the next garbage collection. No mark bits are required, as an object is effectively marked by setting its forwarding address.

Mark-compact garbage collection algorithms have also been implemented in persistent systems. Maheshwari and Liskov for example compact live objects within pages that have been marked by a trace in their implementation of a partitioned garbage collector [ML97].

Benefits and Drawbacks

An important benefit of compacting garbage collectors is the elimination of fragmentation at the cost of compaction, which improves the locality among associated objects. It also reduces the work required by the allocator to locate free space in order to allocate space for objects. Therefore, compaction collectors are usually used in systems that wish to eliminate fragmentation in an attempt to increase performance. Further, mark-compact collectors are able to collect cyclic garbage, since cyclic garbage is left unmarked by the collector's mark phase and is swept by the sweeping phase.

On the other hand, compaction is expensive, since sveral passes (mark, compact, and reference update) of the object graph are required. Compared to mark-sweep collection, the work required to mark-compact garbage collect is proportional to the total size of live objects [JL99]. This is true since compacting depends on the sizes and number of objects that are compacted.

2.3.4 Copying Garbage Collection

Unlike any other garbage collection techniques discussed so far, the concept underlying copying collecting is to divide the memory equally into two contiguous regions called *semispaces*. One semispace contains live data, and the other contains obsolete (garbage) data. In fact the first copying garbage collector was Minsky's garbage collector for Lisp 1.5 [Min63], which used secondary tape storage instead of two semispaces in main memory.

Copying garbage collection does not really "collect" garbage [Wil92], instead all processing is halted and the live objects are traced from the root set. All objects reached by the trace are marked and copied from one semispace (known as the *fromspace*) into the other empty semispace (known as the *tospace*). The remaining objects in the fromspace are assumed to be garbage, since they were not reached by the collector's trace and therefore not copied. Once the copying is complete, the mutator may continue. Note that before every garbage collection the roles of the fromspace and tospaces is flipped. The trace of the

object graph and the copying process are integrated, and the amount of work required is proportional to the amount of live objects [Wil92]. The term *scavengers* is given to copying garbage collectors, since live objects are "picked" out amongst the garbage during tracing and preserved by copying them to the free semispace.

Probably the most famous copying garbage collection algorithm is Cheney's iterative breadth-first semispace copying technique [Che70], which is illustrated in Figure 2.4. To cleary describe the algorithm, we need to introduce a colour marking scheme, known as the *tricolour marking abstraction.*

Tricolour Marking Abstraction

The tricolour marking abstraction, first used in Dijkstra's *On the fly* concurrent marking algorithm [DLM+78], assigns colours to objects as follows :

White objects are those that have not been visited when the garbage collection traversal completes, i.e. unreachable and therefore garbage objects.

Grey indicates that an object has been visited, but that its children objects have not completely been visited by the garbage collection traversal. In this case, the garbage collector needs to revisit the grey object.

Black denotes that an object and its children have been visited and traced. Once objects become black, the garbage collector is done with them, never needing to visit them again.

Note that although the tricolour abstraction was designed to describe concurrent garbage collection algorithms, the abstraction may also be helpful in describing iterative and incremental collection techniques. With the help of the tricolour abstraction, we now discuss Cheney's copying algorithm.

Cheney's Copying Algorithm

Cheney's algorithm [Che70] is an *iterative* copying garbage collection technique which makes use of two pointers and two semispaces, fromspace and tospace. The algorithm utilizes a queue data structure to store the branch points of the object graph. However, no additional memory is required for the queue because it is assembled using the objects that are copied to the tospace. All objects are assumed bo be white before tracing begins. The two pointers `scan` and `free` point to each end of the queue. The `scan` pointer marks the boundary between black and grey nodes, and the `free` pointer indicates the next free location in the tospace. The Cheney collector uses a breadth-first traversal to repeatedly copy live (reached) objects to the tospace, and scans the copied objects for references to other objects that have not been copied. The collection is complete when no such objects can be found. Cheney's algorithm is further clarified with the use of Figure 2.4.[3]

The fromspace in Figure 2.4 shows the object graph before copying collection, and the tospace shows the steps taken in copying objects into the tospace. Note that in the figure, a' represents the tospace copied version of a.

In brief, Cheney's algorithm operates as follows : garbage collection begins by flipping the roles of the semispaces. The `scan` and `free pointers` are initialized to point to the bottom of the *new* tospace, as

[3]Note that this figure is apdated from an illustration in [Wil92].

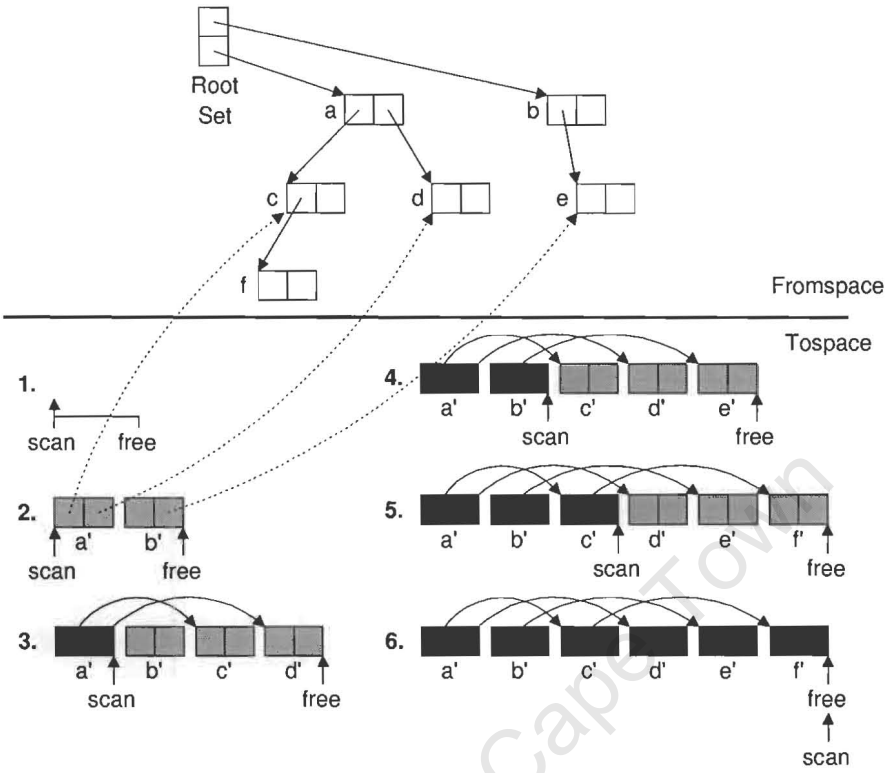


Figure 2.4: Cheney's copying algorithm.

shown in (1). In step (2) objects a and b are traced from the root set, then copied to the tospace. The objects are coloured grey, since they have been visited by the trace, but the children remain untraced as yet. Cheney uses the grey objects in the tospace as the queue, so in step (3) a' is traced resulting in the copying of any objects referenced from it. Note that a' is now coloured black and the copied objects are coloured grey, according to the tricolour abstraction. This is followed by the trace of b' in (4). The copying procedure is complete when all objects in the tospace have had their references traced, i.e. when the `scan` pointer meets the `free` pointer. The tospace objects are therefore coloured black when copying completes.

Copying garbage collection is primarily used in systems to reorganize data and to eleminate fragmentation. Examples of copying collectors include Kolodner's stop/start semispace copying collector [KLW89] and Fenichel and Yochelson's recursive copying algorithm [FY69]. Since the pause times incurred by copying algorithms are usually large, they are typically used in off-line garbage collection techniques. Yong *et. al.* implement a version of Baker's incremental copying and partitioned incremental copying for a client-server persistent object store [YNY97], whereas Butler [But87] investigated copying garbage collection for object-oriented databases.

Later in Chapter 3 we describe the implementation of such an off-line copying garbage collector for the PLaVa store.

Benefits and Drawbacks

Copying collectors have the ability to reclaim cyclic structures, and as shown by Cheney, may be incremental. Copying naturally eliminates fragmentation, since the live objects are compacted into the bottom of the tospace. This compaction also has the advantage of improving the locality among objects. Further, allocation costs are very low, since new memory is allocated simply by incrementing the free space pointer.

The most obvious disadvantage of copying collection is the need of two semispaces, doubling the amount of required memory. The cost of copying garbage collection depends on the amount of live data residing in the fromspace, compared to other tracing schemes like mark-sweep, where the collection cost in proportional to the number of live and garbage objects.

2.3.5 Generational Garbage Collection

Research has shown that in most programs written in a variety of languages, *most objects live a very short time, while a percentage of them live much longer* [LH83, Ung84, Sha98, DeT90, Zor90, Hay91]. Further, a large percentage of the objects that are garbage collected once, survive a number of further collections [Wil92]. These objects are subjected to garbage collection during every collection invocation, and as a result the collector spends most of its time repeatedly collecting the same objects, making the collector inefficient.

The aim of *generational* garbage collection [LH83] is to avoid the described inefficiency, by segregating objects into multiple *generations* (regions) in memory according to their ages. These regions may be independently garbage collected rather than collecting the entire memory, thus reducing the pause times incurred by garbage collection. The collector concentrates on collecting the younger generations (called *minor* collections), since they are likely to contain the most garbage [LH83, Ung84, Sha98, DeT90, Zor90, Hay91]. Objects surviving minor collections are promoted to older generations, which are collected less frequently (*major* collection).

Figure 2.5 shows a simple generational garbage collector that divides memory into two generations. Objects are allocated in the youngest generation, while objects surviving minor collections are promoted to the older generation. (1) shows the initial object graph, while (2) shows the object graph after a minor collection. The surviving objects a, b, c, and d are promoted to the older generation. The unreachable object e in the young generation is garbage and reclaimed. Note that in (2) the objects i and j are allocated after the minor collection has completed, and therefore reside in the youngest generation.

Generational garbage collectors are primarily used in systems that require short collection pause times. Programming languages that make use of generational garbage collection include Standard ML of New Jersey [AM91], Lisp [Moo84], Modula-3 [HD90], and Smalltalk [HD90, JL99]. Bartlett has implemented a generational compacting garbage collector for C++ [Bar90], which is also being used in Computer Aided Design (CAD) tools. Appel presents an efficient, low-overhead generational garbage collector with fast allocation, suitable in a Unix environment [App89]. Appel's collector has also been implemented in the runtime for Standard ML of New Jersey [AM91], showing low collection overheads of between 5 and 10 percent.

Defining a promotion policy for generational collectors is important, since if the *promotion threshold* is too high, the cost and pause time of collecting the youngest partition becomes too large. On the other

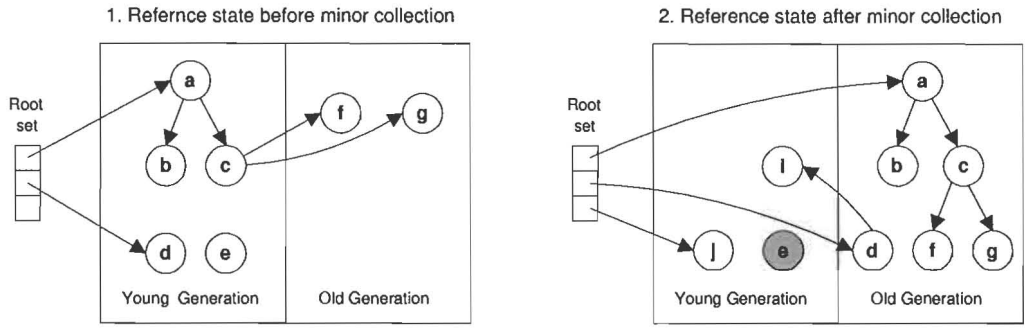


Figure 2.5: Generational garbage collection before (1) and after (2) a minor collection.

hand, if the promotion threshold is too low, objects will be promoted prematurely, which does not allow as many objects as possible to die in the youngest generation. This may result in *tenured* garbage residing in generations that are less frequently garbage collected.

Since generational garbage collection relies on the "ages" of the objects it collects, an age recording mechanism is a prerequisite. Shaw proposes subdividing each generation into two or more regions called *buckets* [Sha98]. The youngest generation contains a *new space* and an *aging space*. Every n scavenges, objects are promoted from the new space into the aging space, while objects in the aging space are promoted to the next generation. Buckets implicitly encode the ages of objects according to which bucket they reside in, and therefore do not require age fields in object headers. Refer to [Wil92, JL99] for more details on age recording mechanisms.

In order for generational garbage collection to be successful, it must be possible to scavenge younger generation(s) independently of older generation(s). In order to accomplish this independancy, inter-generational pointers need to be tracked, since they make up the root sets of the generations. *Write-* or *read-barriers* are commonly used to "trap" inter-partition references, and are further discussed in Section 2.3.6. The barrier is used to record the inter-generational references in a suitable data structure, which is used during garbage collection to determine the root set for the particular generation. Recording methods include entry tables [LH83], remebered sets [Ung84], page marking [Moo84], word and card marking [Sob88], and lists [App89]. Refer to the specified literature or [Wil92, JL99] for more details on inter-generational reference management.

Partitioned garbage collection is closely related to generational garbage collectors. The memory is divided into partitions which may be independently collected, without taking object ages into account. In this case we have inter-partition references which need to be tracked and recorded. Implemetations of partitioned garbage collectors for persistent stores include [YNY97, AFG95, ML97], which are discussed in Section 2.6. In Chapter 4 we describe the implementation of such a partitioned garbage collector for the PLaVa store.

Benefits and Drawbacks

The main advantage of generational garbage collection is the ability to reduce garbage collection pause times because generations may be independently garbage collected. Generational collection techniques also reduce the overall cost of garbage collection by concentrating on short-lived objects in the youngest

generations and delaying the collection of long-lived objects in older generations.

On the other hand, generational garbage collection requires increased complexity in the use of segregated generations for collection. Inter-generational references are naturally introduced, which can be quite problematic since they form part of the root set for specific generations and need to be tracked and located during collection.

2.3.6 Incremental Garbage Collection

Stop/start garbage collectors share the characteristic that once they start a collection cycle, they must either fully complete the cycle, or abandon the work done so far. This is due to the fact that the mutator may possibly alter the object graph while the garbage collector is collecting is. This is often an appropriate restriction, but is unacceptable when the system must guarantee response times, e.g. real-time hardware control systems. The (reduced) pause times caused by garbage collecting generations are usually insufficient for time-critical processing. However, *incremental* garbage collection may be used in such systems, so that garbage collection and time-critical processing can proceed effectivley in "parallel", without wasted effort.

Incremental collectors execute asynchronously with the mutator, i.e. interleaving the processing of the mutator and the garbage collector. The main problem with incremental techniques is that asynchronous execution may introduce inconsistencies because mutator activity is able to modify the object graph that the garbage collector is working on, and vice versa. Incremental collection techniques may not be confused with *concurrent* (or parallel) garbage collectors, which execute simultaneously with the mutator, usually on a multi-processor machine. Concurrent collectors include concurrent reference counting in Cedar [Rov85] and Modula-2+ [DeT90], the concurrent compacting algorithm for persistent heaps described by O'Toole *et. al.* [ONG93], and Detlefs' concurrent garbage collector for C++ [Det91]. For a complete study of concurrent algorithms, refer to [JL99].

Skubiszewski and Valduriez [SV97] describe a novel mechanism for ensuring concurrency during garbage collection in object-oriented databases. They propose a technique known as *GC-consistent cuts*, which are sets of virtual copies of database pages. The garbage collector examines the copies rather than collecting pages of the real database. Copies or cuts are made of database pages at times when an object in the cut is garbage only if it is garbage in the real database. Their technique is easy to implement and is scalable.

The tricolour abstraction [DLM+78], as was discussed in Section 2.3.4, is very helpful in understanding incremental garbage collection techniques. To recap, the tricolour abstraction categorizes objects into three colours, namely black, grey and white, and assumes all objects are white when they are allocated. White objects are those that are not reachable from the root set and have therefore not been visited by the garbage collector's trace. White objects make up the set of garbage, while black objects are those whose immediate children , including themselves, have been visited and traced. Once an object is black, the collector does not need to visit it again. Finally, grey objects are those that have been visited by the collector's trace, but not all their children have been traced or the mutator has modified the object's connectivity in the object graph.

Read and Write Barriers

The foundation of incremental garbage collection techniques is to coordinate the collector with the mutator. However, the mutator may alter the object graph currently being traced by the collector. The garbage collector is notified of any tricolour abstraction violations caused by the mutator and the violating objects are retraced, which eliminates any inconsistencies in the mutator and collector's views of the object graph. Figure 2.6 shows how the mutator may disrupt the tricolour abstraction by writing white references into black objects and how the mutator rescans objects to correct the violation.

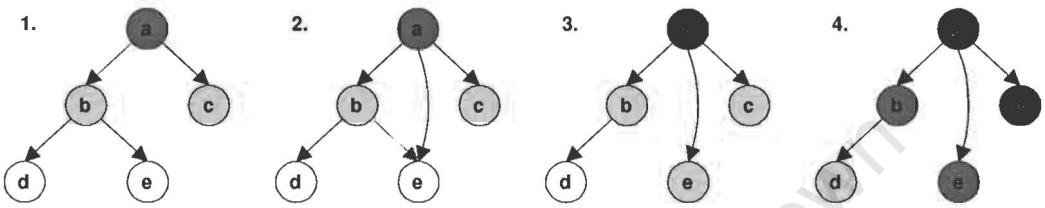


Figure 2.6: Mutator modification violating the tricolour abstraction.

Figure 2.6 (1) shows the object graph once the garbage collector has traced object a and visited objects b and c in breadth-first fashion, and the objects are coloured accordingly. In (2) the mutator deletes the reference from object b to e, and creates a new reference from a to e as shown. This modification causes the abstraction to be violated, since we have a black object referencing a white one. The garbage collector is notified, and a is rescanned to correct the inconsistency, as shown in (3). (4) indicates progress in the garbage collector's trace, marking objects b, e, and c black, and object d grey. Cheney's algorithm illustrated in Figure 2.4 on page 15 reiterates the tricolour abstraction with the garbage collector's trace.

Two distinct barrier methods exist for preventing inconsistent mutator and collector views of the object graph, as described below :

Read-barriers detect mutator access (reads) of pointers to white objects. Whenever the mutator attempts to access white objects, they are immediately visited and marked grey by the collector. The best-known read-barrier algorithm is Baker's incremental copying collector [Bak78]. Baker modified Cheney's copying garbage collector to allow the collector to run incrementally with the help of a read-barrier. Modifications were also made to the tospace, so that the scavenger may compact surviving data to one end of the tospace, while allocation is made from the other end. Refer to [Bak78, Wil92, JL99] for more details on Baker's read-barrier algorithm.

Write-barriers record mutator reference writes from black to white objects. Whenever this occurs, the black object is reverted to grey and is retraced. The example illustrated in Figure 2.6 makes use of a write-barrier to protect white objects. Other examples of write-barriers include Yuasa's sequential algorithm [Yua90] and the *On-the-fly* garbage collector due to Dijkstra *et. al.* [DLM+78]. Compared to read-barriers, write-barriers are usually cheaper since heap writes are less common than heap reads [Wil92].

Write-barriers may be further classified into two approaches, according to Wilson [Wil92]. *Snapshot-at-beginning* write-barriers ensure that no objects ever become inaccessible to the garbage collector. This is achieved by creating a *copy-on-write* virtual copy [Wil92] of the object graph before collection, which

the collector uses for tracing. In this respect snapshot-at-beginning algorithms are *conservative*, since garbage created during the current phase will only be collected in the next phase. An example of such a write-barrier is the snapshot garbage collection algorithm designed by Yuasa [Yua90]. *Incremental-update* write barriers incrementally record potentially disruptive mutator writes, rather than making an estimate of the object graph before collection. The barrier ensures that black objects never reference white objects, by trapping the write and revisiting the black object. Incremental-update approaches are less conservative than snapshot-at-beginning algorithms, since garbage created during the current phase will be detected and reclaimed. An example of such a write-barrier is the incremental-update *On-the-fly* garbage collector due to Dijkstra *et. al.* [DLM+78].

Benefits and Drawbacks

The purpose of incremental garbage collection algorithms is to reduce collection pause times because collection proceeds incrementally.

On the other hand, barrier methods that synchronize the object graph introduce complexity and can be quite expensive, since reference reads and/or writes need to be tracked. The implementation of any read- and write-barrier should therefore be as efficient as possible, in an attempt to minimize this cost.

2.4 Persistence and Persistent Programming Languages

The concept of persistence is to allow arbitrary data structures to be preserved after one execution, so that they may be used in later executions. Researchers believe that persistent language implementations should aim to satisfy *orthogonal persistence* [ACCM83, MA90, AM95], which is based on the following three principles :

The Principle of Data Type Orthogonality : all data of any type should have equal rights to persist.

The Principle of Persistence independence : the form of program code should be the same irrespective of whether it is manipulating persistent or transient data.

The Principle of Persistence Identification : there should be a straightforward, consistent mechanism for determining which data will become persistent.

Persistence is provided by persistent programming languages. These languages provide functionality to allow users to create and modify a database or *store* associated with the language. The language also provides functionality to retrieve and update data to the store.

An advantage of the persistent abstraction is a reduction in complexity. In traditional database systems the language responsible for maintaining information in the database and the language to manipulate information in the database can be and usually are seperate. For example data residing on an Oracle database is usually accessed and manipulated by some sort of querying language, like SQL for instance. Persistent environments integrate the view of the information residing on the database and the view of the programming language, thus reducing the complexity. In other words the language responsible for maintaing the database information is exactly the language used to manipulate data residing in the database. Therefore persistent programming languages are sometimes known as *database programming*

languages. Well known persistent programming languages include DBPL (relational) [MS89], Napier88 (store and multi-paradigm) [MBCD89], Persistent Java (PJava) (object-oriented) [AJDS96], Persistent Prolog (logical) [Col89] and E (object-oriented) [RCS90].

Applications written in persistent programming languages require less code and less time to execute. It has been estimated that persistent programming languages use 30 percent of the code required in traditional databases [ABC+83]. Further, the time taken to execute and store persistent programming language code is less in physical terms.

Probably the only disadvantage of persistent programming languages is the cost of persistence. This includes the cost of constructing a persitent object store and the cost of providing language independent binding mechanisms. Further, there is an efficiency loss in implementing certain algorithms within a persistent store, since the user does not have total control of the physical properties of the machine, e.g. control of paging mechanisms.

2.5 Garbage Collection in Persistent Environments

In Section 2.3 we discussed garbage collection techniques suitable for primary memory. Although these techniques may be suitable for the garbage collection of stores, persistence on secondary storage introduces complications not addressed in primary memory implementations. These issues include [But87, YNY97, AFG95, MBMM98, Pri00] :

Atomic transactions and fault tolerance: modifications in persistent environments require atomic transaction semantics. This ensures fault-tolerance, in that the abort of any transaction may be fully recovered to its original state, in effect undoing the transaction. Persistent environments require both the store and the garbage collector to be fault tolerant.

Persistent environments: non-persistent languages (e.g. C, C++) require memory allocated to a volatile heap to allow the execution of programs. In persistent environments, both a volatile heap and a persistent heap (store) are required.

Store size: primary memory garbage collectors usually operate on relatively small heaps compared to the size of possible persistent stores. Therefore garbage collectors for stores need to cater for a much larger data set. For example reference counting in a small heap may be sufficient, but may fail in the case of a large persistent store containing millions of objects. Similarly, semispace techniques are unworkable since they require double the amount of space.

Disk-resident data: heap garbage collectors reclaim storage in main memory and should be optimized for manipulating data within main memory. In persistent environments data resides on disk, and therefore the garbage collector should be efficient as far as disk I/O is concernced.

Data movement: when data is copied or moved during garbage collection, updates to persistent reference locations are required. These cost involved in performing these updates may be high when updated locations are held in persistent data residing on secondary storage.

Client-server paradigm: persistent architectures are often client-server based, with the user programs executing on clients that access information on the store at the server. The concurrent running of multiple clients creates more opportunities for inconsistencies.

2.6 Garbage Collecting Persistent Object Stores

In this section we explore garbage collectors for persistent object stores in the context of partitioned garbage collection. We show how some of the garbage collection techniques discussed in Section 2.3 are used in these implementations. The related work includes object oriented database systems, client-server stores, distributed stores and partitioned object stores. We describe the algorithms and the ideas and data structures used in the implementations.

2.6.1 Server-based Garbage Collection Algorithms

Persistent object stores or databases may be accessed by clients on server-based architectures. Garbage collection algorithms for such schemes are required to run concurrently with client transactions (applications). Here we describe two server-based garbage collection algorithms implemented in the Exodus store manager [CDG+90].

Amsaleg-Franklin-Gruber Collection Algorithm in Exodus

In [AFG95], Amsaleg *et. al.* describe a partitioned mark and sweep garbage collection algorithm that is server-based, works in the context of ACID transactions [GR93], is incremental and non-disruptive, and is fault tolerant. The algorithm is designed for integration into existing Object Oriented Database Management Systems (OODMS) and was implemeted in Exodus. The collector requires very little synchronization with client transactions, performs minimal logging, and requires no client callbacks or special hardware. However, the algorithm relies on reference updates to be done in ACID fashion. In other words, object graph manipulations will only be processed and recognized when the transaction causing the modification is complete. Further, communication between the client transactions and the server are based on the *write-ahead-logging* (WAL) protocol [Hag87], and storage may only be reclaimed when the freeing of the space is reflected on the store.

The partitioned mark and sweep collection algorithm may collect partitions independently. To accomplish this, certain garbage collection data structures are required. A *colour map*, which is not part of the store, is used for marking objects during the mark phase. Amsaleg *et. al.* introduce a *Pruned Reference Table* (PRT) to store deleted references that have not been committed. This forces the garbage collector to be conservative with respect to uncommitted modifications. When transactions terminate, its entries in the PRT are flagged, which are removed at the beginning of the next garbage collection.

A *Created Object Table* (COT) is used in similar fashion to the PRT and protects uncommitted *created* objects. The PRT is used during marking, whereas the COT is used during the sweeping phase of garbage collection. As with the PRT, the COT entries are flagged when their respective transactions have completed, and are removed at the beginning of the next garbage collection.

Since the transaction updates occur at clients while garbage collection is performed at the server, traditional write-barriers used to update the PRT and COT would be highly inefficient [AFG95]. However, the WAL protocol ensures that log records of any modifications reach the server before the relevant pages are collected. The log records are examined at the server and corresponding updates are made to the PRT and COT.

Mark and sweep garbage collection of pages is performed as follows : individual colour maps are

initiliazed for pages that are to be garbage collected, and are set to indicate that all objects are garbage. A marker traverses the object graph from the persistent root, marking all objects that it reaches as live. The PRT is then traversed in the same fashion. Since the PRT supports the interleaving of transaction updates and marking operations, the PRT allows the marker to be run incrementally. When the marking traversal is complete, the sweeper scans the pages linearly, sweeping only those pages containing both garbage and live objects. Pages containing only unreachable objects are deallocated by the sweeper, since all objects residing in them are known to be garbage. As with the marking phase, sweeping may be run incrementally on a page by page basis.

The above algorithm is extended to collect disjoint partitions (sets of pages). Since partitions are to be collected independently, inter-partition references need to be tracked. Each partition must have a separate persistent root object, as well as a list of incoming references from other partitions, referred to as the *Incoming-Reference List* (IRL). IRL entries contain the target object PID, along with the partition ID of the source object. The drawbacks of this approach is that IRLs have to be managed in a fault-tolerant manner, and cycles of garbage that are distributed across multiple partitions cannot be collected.

The algorithm was implemented in the Exodus storage manager and tested according to garbage collection overhead, off-line collection performance and on-line garbage collection performance. [AFG95] concluded that the overhead imposed by garbage collection on user transactions is small. The performance of the marking phase scales linearly with the partition size for all percentage garbage values. The performance of the sweeper is independent of the percentage of garbage in a page. Finally, their results show that the garbage collector can run in a way that does not negatively impact the performance of the clients.

Yong-Naughton-Yu Collection Algorithm in Exodus

Yong *et. al.* have developed and evaluated garbage collection algorithms for client-server persistent object stores [YNY97]. They concentrated specifically on an incremental, partitioned, copying garbage collection algorithm. The advantages of this incremental algorithm are that it maintains transaction semantics and recoverability, is incremental and non-disruptive, and reclusters and compacts data as it collects. In the context of this thesis we will only discuss their partitioned collection algorithm.

The persistent object store is divied into partitions, which can be divided logically or physically. Partitions are independently scavenged, and a write barrier is used to trap changes made to the object graph. [YNY97] optimize their write-barrier by trapping only alterations made to objects pertaining to the partition being scavenged.

Remebered sets are assigned to each partition for storing inter-partition reference information, which are constructed and maintained by the write-barrier. When a partition is garbage collected, its remembered set is updated by removing entries that reference unreachable objects. Since garbage collection is performed by copying, a *correspondance table* is used to map object PIDs to their locations on the store. When objects are copied during collection, updates to the correspondance table are required.

The partitioned copying algorithm makes use of the tricolour marking abstraction [DLM+78], and new objects are allocated as grey and are assumed to be untraced. Partition selection policies are based on the highest rate of mutation.

A major downfall of the described partitioned garbage collector is its inability to collect inter-partition cyclic garbage. Yong *et. al.* reason that since partition sizes are quite large, inter-partition cyclic garbage

is uncommon. However, they do suggest a weak technique for dealing with inter-partition cyclic garbage. When many inter-partition garbage cycles exist, and as a result storage reclamation is ineffective, adjacent partitions are merged, in an attempt to eliminate inter-partition cyclic garbage. Merging may continue until the entire store forms a single partition, and so all garbage is guaranteed to be collected. The more merging occurs, the longer the collection pause times will be, since the partition sizes increase.

Yong *et. al.* tested their algorithms on a simluation built using C++/CSIM [Sch95b] and a protoype implementation of the Exodus store manager. Benchmark operations for the tests included database generation, database traversals, and databse reorganization. It was found that partitioned incremental copying garbage collection trades locality improvement for time and space. Their results show that improvements in locality are achieved by careful selection of partition sizes.

2.6.2 Garbage Collection in Distributed Stores

The partitioned garbage collector presented by Maheshwari and Liskov is based on the distributed garbage collector presented by Maheshwari himself [Mah97]. Here we study two distributed garbage collection implemetations to gain understanding into how they garbage collect a single unit of garbage collection, e.g. a single site or partition. We take time to study distributed garbage collection techniques since they have a similar architecture to partitioned environments and share many of the same problems, like inter-site or inter-partition references. Figure 2.7 demonstrates how distributed environments apply to partitioned environments.

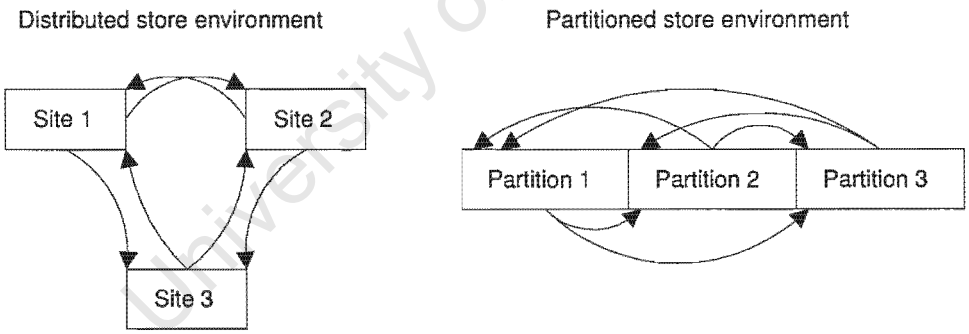


Figure 2.7: Distributed and partitioned environments.

Lang-Queinnec-Piquer Distributed Collection

Lang *et. al.* present a new distributed garbage collection algorithm in [LQP92], which is fault-tolerant, allows multiple concurrent active garbage collections, and reclaims all garbage, including distributed cyclic garbage. They define the distributed store environment as a collection of *nodes* that can communicate with one another via messages. Further, each node contains *cells* or objects. A *group* garbage collection is the collection of multiple nodes, whereas a *local* garbage collection is the collection of a single node. Remote references, i.e. references from one node to another, are represented by an *exit item* at the soure node and an *entry item* at the target node. Each entry item contains a reference count that keeps track of the number of exit items that reference it.

[LQP92] use a *two-phase* marking scheme when garbage collecting a group. Entry items are marked *hard* if they are referenced from outside the group, and marked *soft* if they are referenced from within the group. Cells within a node that are reached during tracing are marked using single bits, indicating liveness with respect to the local collection. The two-phase marking scheme propagates entry item marks when garbage collecting a group as follows :

- Initially, all marks on exit items are reset to *none*.
- The first tracing phase propagates marks from the hard entry items and internal roots. Exit items reached are marked hard.
- The second tracing phase propagates marks from soft entry items, and exit items reached that are *not* already hard marked are marked soft.

Cells not reached by the two-phase marking scheme are assumed to be garbage and may be reclaimed. The same holds for exit items that remain marked as none. In this case, a *decrement message* is sent to the correct entry item referenced by the exit item, which is reclaimed if its reference count reaches zero. Items and cells marked hard by the trace are reachable from either a hard marked entry item or a local root, and are therefore preserved. Items and cells marked soft are not reclaimed, since it is only known if they are reachable from outside the group.

By using this two-phase marking scheme, which propagates marking via entry and exit items, the distributed garbage collection algorithm presented by Lang *et. al.* successfully reclaims all unreachable objects, including distributed cyclic garbage. [LQP92] only provide their algorithm, and the implementation and testing was left as an extension to their paper.

Ferreira-Shapiro Distributed Collection in Larchant

In [FS95], Ferreira and Shapiro provide a study of the garbage collection of the Larchant persistent distributed shared store [FS96]. Their approach allows independent collection of areas of the store and is similar to the algorithm presented in [LQP92], as far as data structures for maintaining inter-site references are concerned.

A *bunch* is a unit of caching and garbage collection containing any number of objects. To facilitate the independent garbage collection of bunches, a bunch records cross-bunch references. A reference into a bunch is known as a *stub*, whereas an outgoing reference from a bunch is known as a *scion* (cf. entry and exit items presented by [LQP92]). Both scions and stubs contain reference counts for keeping track of the number of referencs referencing them.

The garbage collection of a single bunch is performed as follows: at the start of a garbage collection run, the collector allocates a new empty set of stubs to the bunch. Any objects reachable from scions are considered live by the garbage collector. Stubs are allocated when an object in the bunch references an object outside the bunch. When all objects in the bunch have been scanned, any objects that are unreached are considered garbage. The collector then compares the new stub set with the old one. For each newly created stub, a `create` message is sent to the target object, and for every deleted stub a `delete` is issued to the target obejct. This effectively updates the scions for the objects residing in target bunches. Finally the collector discards the old stub set and the new stub set becomes the current one. Garbage collection of the bunch is now complete.

Cross-bunch garbage cycles are reclaimed using the same algorithm. However, when determining the root set for the group, scions external to the group i.e. referenced by an object outside the group, are omitted from the root set. Ferreira and Shapiro only present their algorithm in [FS95] and do not evaluate an implementation of their technique.

2.6.3 Partitioned Garbage Collection

Partitioned garbage collector techniques, first proposed by Bishop [Bis77], are aimed to run on stores (or memory) that is divided into partitions. The aim of dividing the store into partitions is to allow for the independent garbage collection of each partition, in an attempt to reduce collection pause times by garbage collecting smaller regions. This section discusses in detail a partitioned algorithm called PMOS, and briefly introduces the algorithm due to Maheshwari and Liskov. Later on in Section 2.7 in the following chapter, we provide an extensive overview of Maheshwari's partitioned garbage collection scheme.

PMOS

The Persistent Mature Object Space (PMOS) algorithm presented by Munro *et. al.* in [MMH96, MBMM98] is an incremental, partitioned garbage collection technique for storage reclamation in persistent stores. Their algorithm is an extension of the Mature Object Space (MOS) algorithm due to Hudson and Moss [HM92]. The Distributed Mature Object Space (DMOS) garbage collector [HMMM97] is a further extension of the MOS and PMOS algorithms, designed for garbage collection in distributed environments.

In PMOS the store is divided into at least two regions called *trains*. Each train is further divided into regions called *cars*.[4] Trains are ordered according to the time they were created; therefore a train is said to be *youger* or *older* than another train. One or more cars are garbage collected in each invocation of the collector, which copies all reachable objects into other cars within a younger train. Munro *et. al.* show that inter-car cyclic garbage is eventually "herded" into a single train, only then being successfully reclaimed. Partitioned copying is based on the following rules [MBMM98] :

1. Objects reachable from roots are copied to a younger train, adding a car if required.
2. Objects that are reachable from younger trains are copied to the those younger trains, adding a car if needed.
3. Objects reachable from older trains are copied to any car in the current train, adding a car if necessary.
4. Objects that are reachable from other cars within the same train are copied to any other car of the train, adding a car if required.
5. All objects that are unreachable are considered garbage and are reclaimed immediately.

To facilitate the independent garbage collection of cars, data structures to keep track of inter-car references and the locations of moved objects are required. In PMOS a remebered set (remset) is associated

[4]The PMOS algorithm is often referred to as the "Train Algorithm" because of the abstraction used to describe it.

with each car to store information about incoming references into the car. PMOS uses a single $\triangle$loc set to record information about the locations of objects that are moved during collection, so that references to those objects may be updated at some later time. Finally, a $\triangle$ref set is used to record information regarding inter-car reference creations or deletions, allowing the updating of remsets to be deferred. The use of the $\triangle$loc and $\triangle$ref sets and the deferring of remset updates helps reduce the I/O impact of garbage collection [MBMM98].

Remsets are stored together with their cars and are updated using the $\triangle$ref set when cars are faulted in. When an (inter-car) reference to an object residing in a non-resident car is created or deleted, the modification is inserted in the $\triangle$ref set. Remset entries are removed as follows: when a car is faulted in, its outgoing references are determined and stored in a table. When a car is about to be written back to secondary storage, its outgoing references are determined, and compared to the initial set of outgoing references, inserting any differences into the $\triangle$ref set. This mechanism ensures that the remsets of cars are up to date before cars are garbage collected.

The PMOS algorithm was tested with the 001 benchmark (written in Napier88) on the Napier88 generic persistent store. A 20MB store containing 20 000 interconnected parts was created, and queries were applied to the store (lookups, scans, traversals, inserts, etc.). The results show that PMOS performs well in the collection of persistent stores. However, the algorithm needs to be "tuned" as far as car and train sizes are concerned, as well as the sizes of the remsets, $\triangle$ref and $\triangle$loc sets, in an attempt to minimize I/O.

2.7 Maheshwari-Liskov Partitioned Garbage Collection

Maheshwari and Liskov present a partitioned garbage collection algorithm in [ML97]. Their algorithm is based on the distributed collection algorithm due to Maheshwari [Mah97]. Here we provide an overview of Maheshwari's partitioned garbage collection algorithm, so that the reader is familiar with the ideas and terminology before the algorithm implementation is described in Chapter 4.

When a store is divided into partitions, inter-partition references are naturally created as the store becomes populated, as illustrated in Figure 2.8. The collector aims to collect partitions independently and so inter-partition references comprise the root set for collections. We explain how the algorithm maintains and keeps track of inter-partition references in an efficient manner and discuss how a global marking scheme ensures the collection of inter-partition cyclic garbage. Finally we provide an example to illustrate global marking and the garbage collection scheme.

2.7.1 Inter-Partition Reference Management

All garbage collection algorithms designed for partitioned storage require a mechanism to maintain the inter-partition state of the store, since garbage collection requires complete knowledge of the incoming references into a partition in order to garbage collect it independently and successfully.

Figure 2.8 shows a persistent root object referencing other objects on a partitioned store. When garbage collecting partition 2, the garbage collector needs to know that objects `d` and `e` are referenced from outside the partition, and therefore form part of the root set for the partition. If the collector fails to do so, `d` and `e` will be recognized as garbage as they seem unreachable, and will be reclaimed.

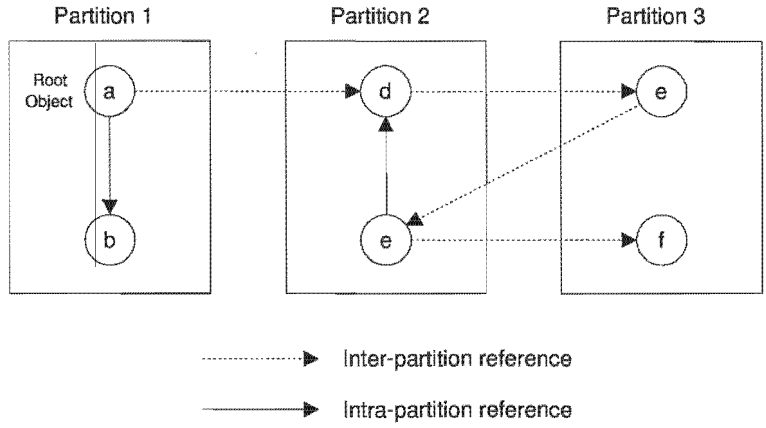


Figure 2.8: A partitioned store.

We therefore require data structures to store inter-partition references and an efficient mechanism of keeping them up to date. Maheshwari uses *inlists* to record incoming references into a partition, and *outlists* to record outgoing references from a partition. Outlists are not a necessity for garbage collection, but Maheshwari maintains them since they are an efficient mechanism for updating inlists. The implementation of these lists is discussed in Chapter 4; conceptually they are lists of addresses, each associated with a source and destination partition.

Before garbage collecting a partition, its inlists need to be updated to update the root set for the partition. It is inefficient to update inlists on-the-fly as inter-partition references are created, and therefore Maheshwari makes use of *delta lists*. Delta lists are memory-resident data structures that store created inter-partition references. When a partition is garbage collected, its inlists are updated using the delta list. Thus the updating of a partition's inlists is deferred until the partition is garbage collected.

2.7.2 Global Marking

In order to garbage collect a partition, we require a *partition trace* through the partition to determine live objects. Tracing starts from the root set for the partition (inlists and delta-lists), setting a trace bit on all objects reached, after which all objects that have their trace bit unset are reclaimed since they are known to be garbage. Either copying or mark-sweep can be used to collect the partition, and afterwards the outlists of the partition are updated. To handle inter-partition cyclic garbage, Maheshwari makes use of an incremental global marking scheme to ensure the collection of inter-partition cyclic garbage [Mah97, ML97]. The process that distinguishes live objects from garbage when collecting a single partition is called tracing, and the mechanism for detecting cyclic garbage is called (global) marking.

Each object has associated with it a *mark bit* used for global marking, and a *trace bit* used for partition tracing. If an object's mark bit has been set, it means that the object has been reached by global marking. Similarly, if an object's trace bit has been set, it indicates that the object has been reached by partition tracing. Global marking is "piggybacked" on partition tracing, thus making the marking scheme efficient.

Global marking and garbage collection proceed incrementally in phases. Maheshwari maintains a global phase counter and maintains a local phase counter associated with each partition. Only the

persistent root object is marked at the beginning of a global marking phase. Thereafter, each partition trace propagates marks from the root set of the partition to its outlists, which are used to update the markmaps of further partitions.

At the end of a global marking phase all objects reachable from the persistent root are marked, since the algorithm ensures that the mark from the persistent root has been fully propagated through all partitions. When garbage collecting partitions in the next phase, all unmarked objects are known to be garbage and are reclaimed since they are unreachable from the persistent root. This is how the global marking scheme ensures the garbage collection of inter-partition garbage cycles.

2.7.3 Example

Here we give a brief example of how partitions are garbage collected and how global marking ensures the collection of inter-partition cyclic garbage. Figure 2.9 shows the stages to complete a global marking phase as marks are propagated from the root object on a store containing three partitions. Objects are either unmarked, delta marked or marked. In the figure, inlists of partitions are represented by triangles and for simplicity we do not indicate whether objects are untraced or traced.

Initially only the persistent root of the store is marked. Figure 2.9 (1) shows the state of the store after we have garbage collected the partition containing the persistent root. During the tracing of this partition, marks are propagated from the root thus marking object `b`, and delta marking `d`. In (2) the second partition has been garbage collected. Marks are propagated marking object `d`, and delta marking `f`. Note that the trace will visit object `e` because of the inlist reference to it, so it will not be reclaimed at this stage. Similarly for the garbage collection of partition 3, where object `f` is marked and `c` is delta marked. Note that since object `g` was unreachable in the partition trace, it is known as garbage and we reclaim it in (3). When we need to garbage collect partition 1 for a second time, object `c` will become marked via its inlist reference (from the delta markmap).

At this stage the global marking phase is complete. All objects reachable from the root are marked and all unreachable objects, except for those belonging to inter-partition cyclic garbage, have been reclaimed. Unmarked inter-partition garbage cycles with be reclaimed when partitions are next traced, since they have been left unmarked by global marking. This is ensured by maintaining a global phase counter and local phase counters assigned to each partition. When tracing a partition whose phase counter is one less than the global phase counter, it is known that unmarked objects may be safely reclaimed, because they are part of an unreachable/garbage inter-partition cycle (e.g. `e` and `h` in Figure 2.9).

Further details on Maheshwari's algorithm are provided later on in Chapter 4 where we describe the implementation of the partitioned garbage collection algorithm. We discuss the finer details of global marking in Section 4.6 and illustrate the individual stages of garbage collecting a partition independently in Section 4.8.

2.7.4 Fault Tolerance and Synchronisation

The data structures responsible for maintaining inter-partition reference information (i.e. translists, inlists, outlists) are stored as regular persistent objects [ML97]. Maheshwari's implementation ensures that all modifications to objects are logged and installed on disk later, thus guaranteeing that inter-partition information survives crashes or errors during garbage collection. Fault tolerance issues concerning delta

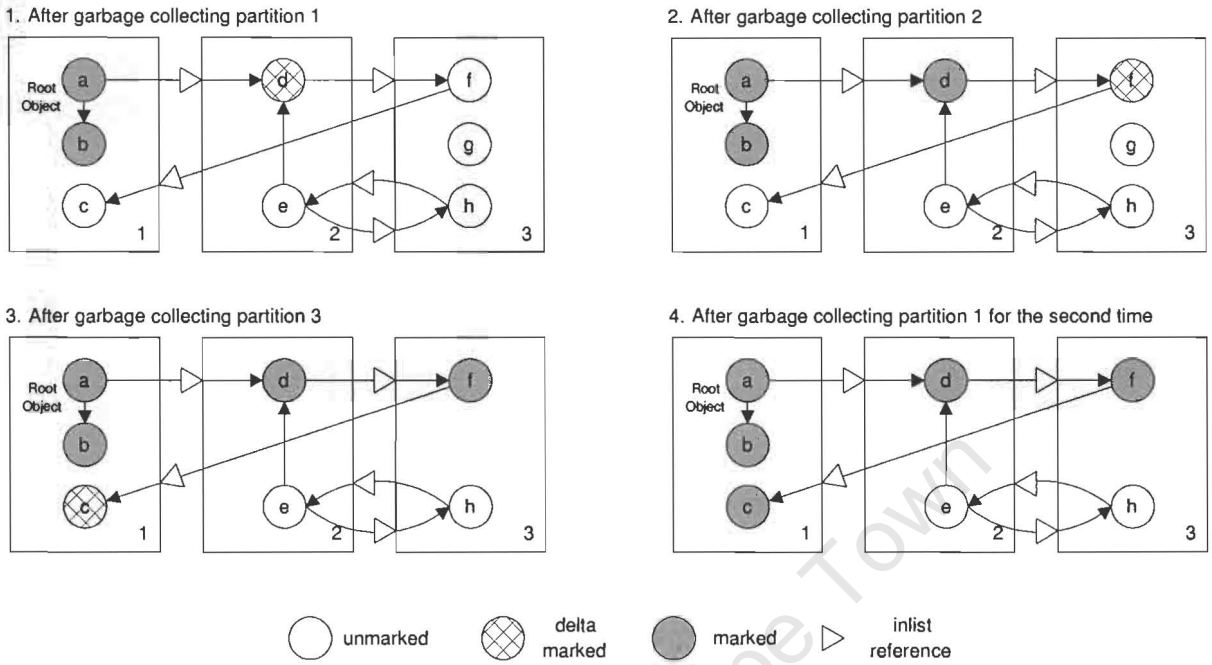


Figure 2.9: Global marking phases propagating marks from the root.

lists may be disregarded, since the information in delta lists is recovered by reprocessing the log, whenever a crash occurs.

Maheshwari and Liskov propose that their tracing and marking approach may be used in combination with various concurrent collectors, to ensure synchronisation for partitioned garbage collection. They use examples of a replicating collector [ONG93], or a mark-and-sweep collector, as used in [AFG95]. The replicating collector requires little synchronization with applications, but requires primary memory to contain sufficient space for two partitions. The mark-and-sweep locks pages during sweeping, thus ensuring synchronisation.

An important factor in garbage collection is how often to garbage collect. It makes no sense to garbage collect a large store when little garbage exists; at the same time it makes no sense to not garbage collect a store with large amounts of garbage. Cook *et al.* [CKWZ96] have researched this problem of how often to perform garbage collection. Two semi-automatic, self-adaptive collection policies for controlling collection rates are described. The policies are based on user preferences. The Semi-Automatic I/O policy attempts to limit garbage collector I/O, while the Semi-Automatic Garbage policy attempts to limit the amount of garbage in the database. These policies were tested using trace-driven simulations. The results show that the choice of collection rate can have a significant impact on application performance and that the "best" rate depends on the dynamic behaviour of the application [CKWZ96]. Implementing and testing such collection policies is beyond the scope of this thesis, and is therefore left to further work.

2.8 Choice of Garbage Collection Algorithms

This thesis is concerned with the implementation and testing of garbage collection algorithms for the PLaVa store. In Chapter 2 we presented numerous heap and secondary storage garbage collection schemes. Here we explain the reasons for choosing to implement firstly a simple offline semispace copying collector and then Maheshwari's partitioned garbage collection algorithm.

2.8.1 Semispace Copying

Since the PLaVa store lacked any form of garbage collection, we decided to first implement a simple semispace copying garbage collector. The goal of this implementation was to understand the basic operation of PLaVa, to understand how object traces are performed, and how the PLaVa store operates. Semispace copying was chosen since it is a relatively straightforward collection technique and could provide a useful point of comparison. We would be able to determine the tradeoffs involved in choosing between a simple, low-overhead but space-expensive collection method and a more complex, higher-overhead method that is economical in space consumption. Although semispace copying techinques require double the amount of storage because of the need for two semispaces, the technique is efficient in the following ways :

- Copying naturally compacts data during garbage collection, thus eliminating fragmentation.
- Cyclic garbage is reclaimed during copying garbage collection.
- Copying collection can be made iterative using Cheney's algorithm [Che70].

2.8.2 Maheshwari's Partitioned Collection Algorithm

The second algorithm chosen was Maheshwari's partitioned incremental garbage collector [Mah97, ML97]. Collection pause times are reduced by incrementally garbage collecting smaller regions or partitions of the store. Further, partitioned garbage collectors can be "tuned" to perform most efficiently by altering page size, partition size, the number of pages per partition, and the number of pages. However, garbage collecting a partitioned store introduces issues of inter-partition reference management and inter-partition cyclic garbage collection.

Of the partitioned techniques discussed so far [YNY97, ML97, MBMM98], we found Maheshwari's partitioned collection algorithm of particular interest. His scheme is an adaptation of his own distributed algorithm [Mah97] for persistent stores, and its characteristics relevant to this thesis are as follows :

- The structures and mechanisms for maintaining inter-partition references are efficient.
- Global marking is efficiently piggybacked on partition tracing and ensures the collection of inter-partition garbage cycles with little overhead.
- The algorithm does not require moving objects to new pages during collection, thus any clustering mechanisms in place can be maintained.

All in all Maheshwari's algorithm is solid, complete and safe, and is an efficient method of garbage collecting partitioned secondary storage. It is efficient in propagating marks and maintaining inter-

partition reference information, and ensures the garbage collection of inter-partition cyclic garbage by using a global marking strategy.

2.9 Evaluating the Algorithms

This thesis describes the implementation of semispace copying in the following chapter and of Maheshwari's partitioned garbage collection algorithm in Chapter 4. Later in Chapter 5 we evaluate the implemented algorithms. We observe the effect of the amount of garbage on the garbage collection times with both algorithms. For semispace copying, we compare recursive depth-first tracing to iterative breadth-first tracing. For the partitioned algorithm, we observe the effect of the number of inter-partition references, the amount of inter-partition cyclic garbage, and the partition size on collection time. We also investigate the effect of partition selection policies on garbage collection time.

2.10 Summary

This chapter has introduced garbage collection and its aims. We have provided an in-depth review of heap garbage collection schemes, how and when they are used, and what their advantages and disadvantages are. We discussed the abstraction of persistence and the programming languages associated it. Finally we explained how garbage collection in persistent environments differs from garbage collection in heaps, and reviewed some well-known garbage collection algorithms for persistent stores.

Chapter 3

Semispace Copying Garbage Collection

This chapter describes the implementation of an offline, semispace copying garbage collector for the PLaVa store. The implementation details cover the modifications made to the PLaVa store to suit semispace copying, and the copying algorithms used in copying garbage collection. The chapter is organized as follows :

- Section 3.1 describes class descriptors which are used to identify references within objects.
- Section 3.2 discusses the descriptor hash table used to map class names to descriptor PIDs.
- Section 3.3 details the modifications made to the PLaVa store.
- Section 3.4 discusses object PID modifications and the implementation of a store handle table.
- Section 3.5 discusses the implementation of copying algorithms.
- Section 3.6 discusses issues concerning store exhaustion and size increase.

3.1 Class Descriptors

In order to perform traversals and copying of objects, it is imperative to be able to efficiently identify all references within objects. When an object is loaded in binary format from the store, its references to other objects need to be found. *Class descriptors*, which are structures containing information about classes (meta data), may be used to locate references within objects. Objects are associated with their corresponding class descriptor by containing a reference (PID) to the descriptor.

All persistent store implementations require some mechanism to determine an object's references. The Persistent Object Store designed for PJama [PAD+97] allocates descriptor objects for each type of object, which contain information about the positions of references within objects. In the case of Napier88 [Bro89, LY96], the objects themselves are optimized so that all the references within an object are grouped together at the beginning of the object, thus allowing efficient reference identification.

The implementation of PLaVa class descriptors took a similar approach to the PJama descriptors. As shown in Figure 3.1, class descriptors are structures containing the size of the class name, along with the

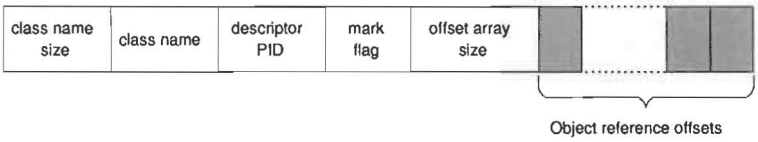


Figure 3.1: The PLaVa object descriptor format.

class name, the descriptor PID, and a mark flag used in garbage collection. We require descriptors to store the class name of the classes they describe. This is because in PLaVa, Tjasink uses an object's class name to locate the correct object, in order to resolve its methods and other structures that are required [Tja99].

Objects that contain references have their descriptors extended to contain the number of references, as well as the reference offsets within the object. Arrays that contain objects or references do not require reference offset information to be stored in their descriptors, since the array's elements may be scanned to determine references. All other array types do not require reference offset information to be stored in their descriptors, since they contain no references to other objects. To efficiently distinguish the types of arrays and objects, Tjasink stores the array/object type in the header of the array/object when it is declared [Tja99].

As in the PJama implementation [PAD+97], objects/arrays of the same type share a single descriptor. This is advantageous because only one descriptor per object type is required, thus saving space on the store. On the other hand, locality of reference among objects and their descriptors may cause an increase in store Input/Output (I/O) when objects, along with their class descriptors, are loaded from the store.

To minimize I/O costs due to disk reads and writes, class descriptors are always kept in memory. This approach may become inefficient when working with very many descriptors, which are all memory-resident. This inefficiency problem is addressed in the following chapter by treating class descriptors like objects and arrays, i.e. they reside on the store and are faulted in as required.

Figure 3.2 illustrates the use of class descriptors according to the object graph shown. Object 1 is of type A, and the remaining objects are of type B. The figure shows how the reference locations within objects are stored in their corresponding descriptors, as well as the sharing of descriptors among objects of the same type.

3.2 The Descriptor Hash Table

When an object of a *new* type is promoted, the PLaVa stabilizing procedure creates a corresponding *new* descriptor, writing both to the store. However, when a new object is promoted whose object descriptor already exists, the stabilizing procedure does *not* create another descriptor. Instead we need to determine the PID of the existing descriptor, so that its descriptor reference in the object may be set to the correct PID.

We use a memory-resident *descriptor hash table* to map class names to their corresponding descriptor PIDs. The hash table is created whenever a new store is created, and PIDs are inserted into the hash

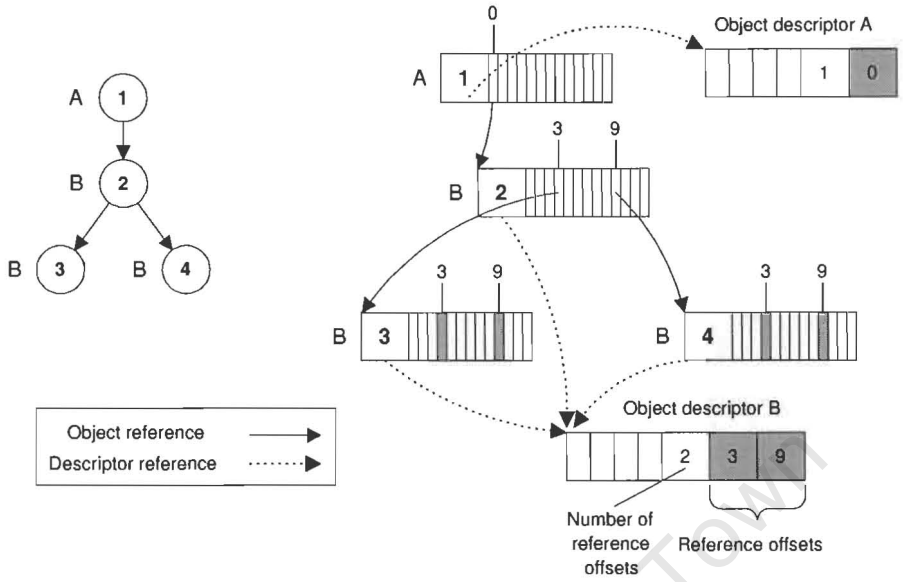


Figure 3.2: An object graph showing object descriptors.

table as descriptor objects are created. We require the table to persist, and so we append the hash table to the store. The table is updated after a stabilize procedure completes or whenever the store is closed. Whenever an existing store is opened, the descriptor hash table is loaded from the store into memory.

Hash table searches become inefficient as the table fills up [Wei94], and so we double the table size when the number of elements reaches a specific watermark.

3.3 PLaVa Store Modifications

To support semispace copying, the original PLaVa store required modifications. Here we discuss alterations made to the store and the `PStore` class, along with the organization of the store to support semispaces.

3.3.1 Store Isolation and PStore Class Extentions

The original version of PLaVa [Tja99] was implemented so that stores are automatically created when executing any Java program. The JVM accepts the store name from the command line. If a store with the correct name exists, it is opened and the program execution continues. If on the other hand no such store exists, a new store is created using the store name from the command line. Further, if no store name is specified in the command line, the JVM automatically creates a new store with a default name. Therefore non-persistent programs would have stores created for them, even though the program would not instruct the JVM to do so.

The store was isolated from the JVM allowing users to explicitly create and open a store when programming with persistence. The function to create stores was altered to accept an optional parameter

for specifying semispace sizes. This allows users to define store sizes to specifically suit applications. The `PStore` class was modified to include code for explicit store functionality, and native Java code was built into the JVM to support the store functions. The extended `PStore` class is shown in Figure 3.3, with the new functions shown in italics (cf. to the original `PStore` class as was discussed in the previous chapter in Section 1.3.2).

```
public class PStore
      {
       private PRoot roots[];
       private boolean storeOpen = false;
       private int maxNumRoots = 5;
       private int numRoots;

       public PStore newStore(String storeName);
       public PStore newStore(String storeName, int storeSize);
       public PStore openStore(String storeName);
       public void closeStore();
       public void addRoot(String n, Object o);
       public Object getRoot(String name);
       public void stabilizeAll();
      }
```

Figure 3.3: The extended `PStore` class.

3.3.2 Metaspace and Semispace Store Organization

To support copying garbage collection, the original PLaVa store [Tja99] and store header were modified. The store was partitioned into two equally sized *metaspaces* and two equal sized *semispaces*, and the store header was altered to contain metaspace and semispace information. This is shown in Figure 3.4(1) and (2) respectively. Metaspaces are dedicated to storing class descriptors and the semispaces are used for storing objects and arrays. We user a contiguous area (metaspace) to efficiently read/write descriptors from/to the store, because if descriptors were written among objects and arrays, the store I/O associated with loading and writing descriptors would be inefficient.

An alternative would be to discard the descriptor memory and treat class descriptors like objects and arrays, i.e. loaded individually from the store as required. Then metaspaces would no longer be required and descriptors would be located among objects and arrays on the store. In the following chapter in Section 4.2 we explain how this was done when we built the partitioned store.

Generally a persistent store contains more objects and arrays than descriptors, not to mention that object sizes are usually larger than their respective descriptors. Therefore the metaspace and semispace sizes may be separately defined in the store header, so that metaspaces may be smaller than semispaces

Table 3.1 shows the information contained in the store header. The `firstavail` variables are used when objects and descriptors are promoted to the store. An object that is promoted is written to the store at the corresponding `firstavail` location, causing the `firstavail` variable to be increased by the size of the object. The `start` variables are used to initialize the `firstavail` variables, when the spaces

are flipped during copying garbage collection.

`metaspace0start`	stores the start position of metaspace0
`metafirstavail0`	stores the first available postion in metaspace0
`semispace0start`	stores the start position of semispace0
`firstavail0`	stores the first available position in semispace0
`metaspace1start`	stores the start position of metaspace1
`metafirstavail1`	stores the first available postion in metaspace1
`semispace1start`	stores the start position of semispace1
`firstavail1`	stores the first available position in semispace1
`whichspace`	indicates which metaspace and semispace represent the tospaces
`PIDCounter`	stores the PID count for persistent object allocation
`objectMark`	stores the mark flag used for traversing and copying objects
`rootPID`	stores the `PStore` root object PID
`descriptorHashTableStart`	stores the location of the descriptor hash table on the store

Table 3.1: Information contained in the semispace store header.

Figure 3.4 (2) shows the meta- and semispaces of a store containing class descriptors and data. When a store is created, a `PStore` object is initialized and written to the store. The `rootPID` field is updated to the position of the `PStore` root object on the store. Once root objects have been added to the root set (the `PRoot` array) and the store is stabilized, all objects reachable from the root set become persistent and are appended to semispace 0 at `firstavail0`. Before the store is closed, the descriptors stored in memory are written to metaspace 0 at `metafirstavail0`.

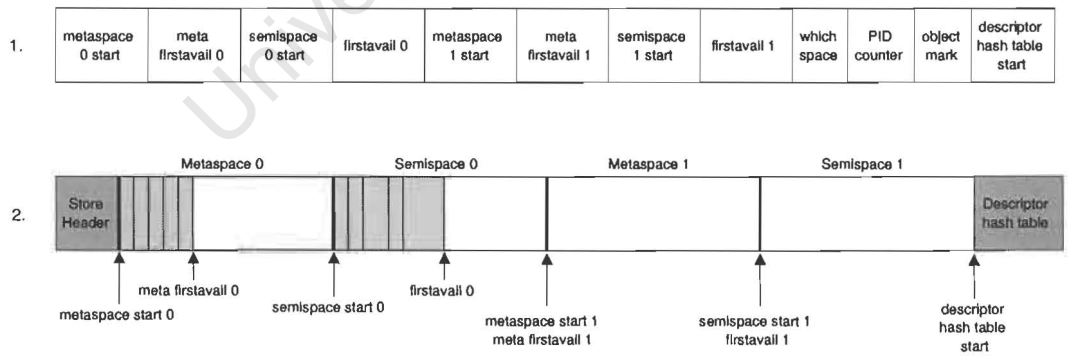


Figure 3.4: Store header and store organization for semispace copying.

3.4 PID Modifications and the Store Handle Table

During copying garbage collection, i.e. when live data on the store is copied to another location, the PIDs referencing any copied data need to be updated in the persistent objects they reside in. The work

required for updating PIDs may be substantial and is avoided with the use of a *handle table*, which maps object PIDs to the object locations on the store. When objects are copied to other locations on the store, the corresponding handle is updated to contain the new position of the object on the store, and no PID updates occur within the objects themselves. The handle table is similar to the PJSL indirectory [PAD+97]. Yong *et. al* make use of a correspondance table in their client-server database garbage collector [YNY97], which has the same function as the handle table.

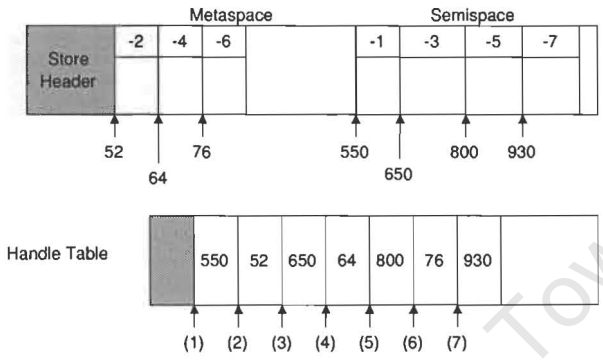


Figure 3.5: The store handle table.

[Tja99] implemented PIDs as the negated location on the store the object is assigned to. Since the handle table stores the locations of objects within the store, PIDs are no longer required to do so. Instead the following PID modifications were made : we have a PID counter which keeps track of PID allocation and is initially set to -1. Each time a new PID is allocated to an object, the PID counter is decremented. However, once an object becomes garbage and the PID is no longer required, it is *not* available to assign to other objects.

As shown in Figure 3.5, the handle table comprises a binary file containing object positions and is accessed using PIDs as indexes. In other words, if the location of an object with PID -5 is required, the handle table is scanned at position 5. The handle table is not updated when objects are garbage collected, since to reuse handles would require additional space and the use of free handle management is not worth the effort, since we are merely trying to implement a simple garbage collection scheme for PLaVa.

3.5 Semispace Copying Garbage Collection

Previous sections in this chapter discussed the implementation of class descriptors, the descriptor hash table and modifications made to the PLaVa store to support semispace copying. We also looked at depth-first and breadth-first traversal algorithms and how cycles are handled in object graphs. In this section we deal with the implemenation of semispace copying algorithms. We implement both recursive depth-first and iterative breadth first copying algorithms by adapting the traversal algorithms in the previous section, allowing them to copy the objects they reach.

We begin with a demonstration of the semispace copying collection process illustrated in Figure 3.6 (note that shaded objects represent marked objects). A program stabilizes after creating garbage and the resulting store is shown in (1). The copying garbage collector is then invoked. Firstly, `whichspace` is

set to -1, to indicate the flipping of the spaces, and `firstavail1` is set to `semispace1start`. Similarly, `metafirstavail1` is set to `metaspace1start`. The garbage collector begins tracing the object graph from the root, marking and copying those that are reached into semispace 1. Descriptors reached during the trace are marked and copied from the descriptor memory into metaspace 1. During the copying of objects and descriptors, corresponding entries in the store handle table are updated to reflect the movement of data. Once the garbage collection has completed, we copy the live descriptors into metaspace 1. The resulting store is shown in (2).

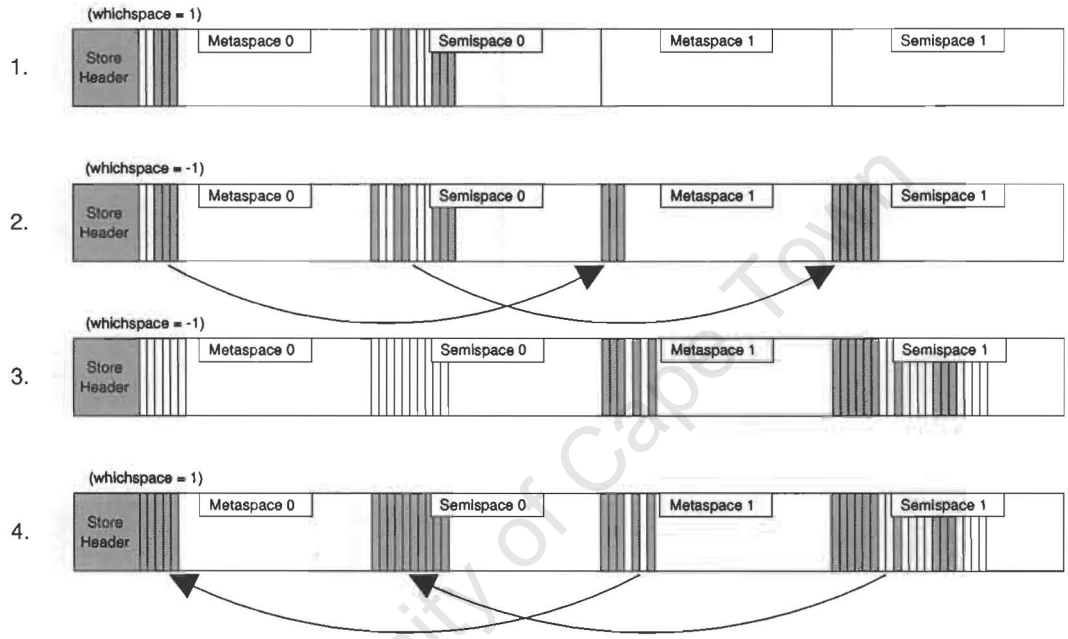


Figure 3.6: Semispace copying garbage collection.

Figure 3.6 (3) shows the store after a program has modified the object graph by adding objects to the graph and creating garbage. The store is stabilized and the copying garbage collector is invoked on the store. The resulting store is shown in (4). The copying proceeds as before, except the reachable objects are marked and copied from semispace 1 to semispace 0, and descriptors are copied into metaspace 0.

This copying collection example demonstrates how live objects and live descriptors form contiguous areas on the store as indicated.

This copying collection example demonstrates how live objects and live descriptors form contiguous areas on the store as indicated. To maximize the use of storage, object allocation in the semispaces and descriptor allocation in the metaspaces should grow towards each other. For example Printezis *et. al* [PAD+97] use a similar approach by allowing object allocation and the indirectory entries to grow towards one another. However, it was found that the stabilizing procedure in PLaVa cannot be interupted [Tja99], and therefore this allocation approach is unsuitable. In Section 3.6 we explain how stabilizing procedures that exhaust semispaces are taken care of.

As with the POGT, the copying garbage collector supports both recursive depth-first and iterative breadth-first copying. We now discuss these copying traversal algorithms with the help of diagrams in the following sections.

3.5.1 Recursive Depth-first Copying

Recursive copying proceeds exactly the same as the recursive tracing, except that objects and descriptors that are reached by the trace are marked and copied to the empty semispace.

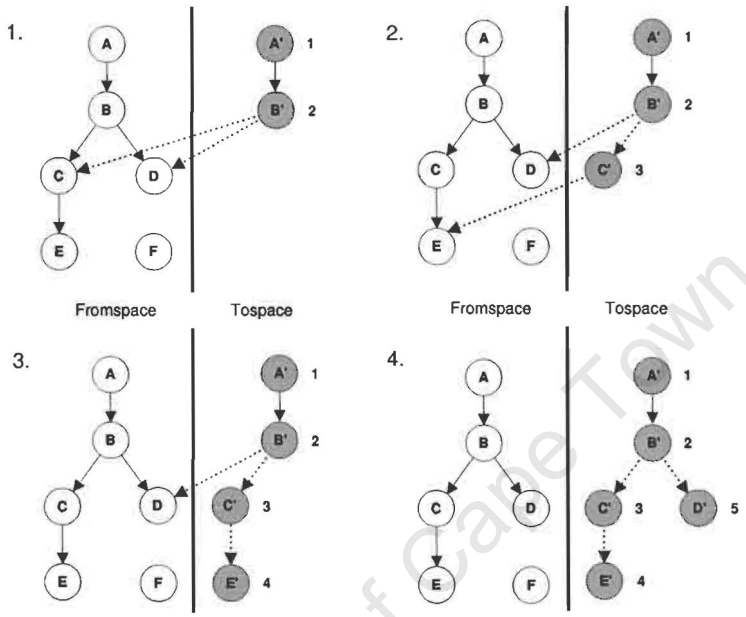


Figure 3.7: Recursive depth-first semispace copying.

A recursive depth-first semispace copying procedure is illustrated in Figure 3.7 using the object graph as shown. Descriptors are not shown to simplify the diagram, and a copy of object A is represented by A'. The numbers are used to indicate the order in which the objects are copied and the shaded objects represent the marked copied objects. Note that once an object has been copied, the copied version is traced to determine references, and the original object in the fromspace becomes obsolete. The figure also shows how an unreachable object F is reclaimed by copying garbage collection.

3.5.2 Iterative Breadth-first Copying and Queues

The iterative breadth-first copying algorithm is an adaptation of Cheney's copying algorithm [Che70]. The breadth-first algorithm operates in the same way as iterative breadth-first traversing, additionally copying the reached objects and descriptors into the empty semispace and metaspace respectively, before tracing the object's references. The algorithm differs from Cheney's algorithm in the fact that Cheney uses the copied objects in the tospace to construct the queue. In comparison, we construct an external queue to store branch points within the object graph. This approach was chosen to minimize store I/O.

An iterative breadth-first semispace copying traversal of an object graph is shown in Figure 3.8, which excludes class descriptors to keep the diagram simple. The numbers indicate the order in which the copying of objects occurs and the respective queue operations are also shown.

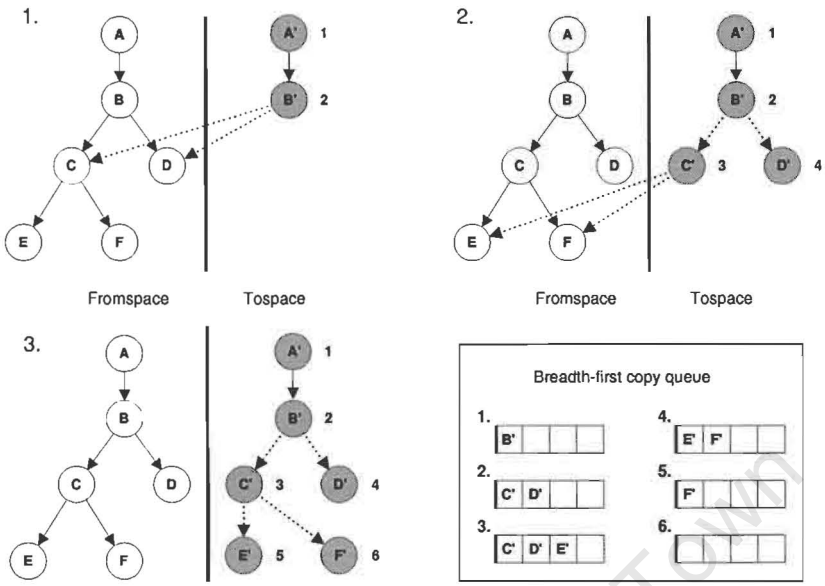


Figure 3.8: Iterative breadth-first semispace copying.

3.6 Semispace Exhaustion and Size Increase

We now discuss methods to prevent store exhaustion by increasing the store size. Possible solutions are to force a garbage collection in an attempt to free up space within the metaspace or semispace, or to increase the meta- and semispace sizes. The implementation of the PLaVa copying collector uses a hybrid approach, as will now be explained. There are two cases when the semispaces may become exhausted, requiring different methods for increasing the space sizes. The cases are demonstrated in Figure 3.9 [1] and are as follows :

Case 1 : metaspace 0 and semispace 0 form the fromspace. Semispace 0 is exhausted when the JVM attempts to write an object to the store which causes `firstavail0` to become larger than `metaspace1start`. The stabilizing procedure is allowed to complete (since it may not be interupted) which causes objects to be written into metaspace 1 if semispace 0 becomes exhausted. The metaspace and semispace sizes are then increased, as shown in Figure 3.9 (1 (ii)). Once the spaces are increased in size, a garbage collection is forced. The result is shown in (iii), where class descriptors reside in metaspace 1 and objects reside in semispace 1.

Case 2 : metaspace 1 and semispace 1 form the fromspace. Semispace 1 is exhausted when the `firstavail1` counter is larger than the sum of `semispace1start` and the semispace size. The same approach as in case 1 cannot be taken for the following reason. Enlarging semispace 1 is not a problem, but increasing the sizes of metaspace 0, semispace 0, and metaspace 1 is problematic because they cannot be extended into semispace 1. Instead, a copying garbage collection is forced, in an attempt to reclaim garbage so that the store is no longer exhausted. If the store remains

[1] Recall that metaspaces are configurable to be smaller than semispaces in size. To make the diagram simpler to understand, we have made the metaspaces and semispaces equal in size.

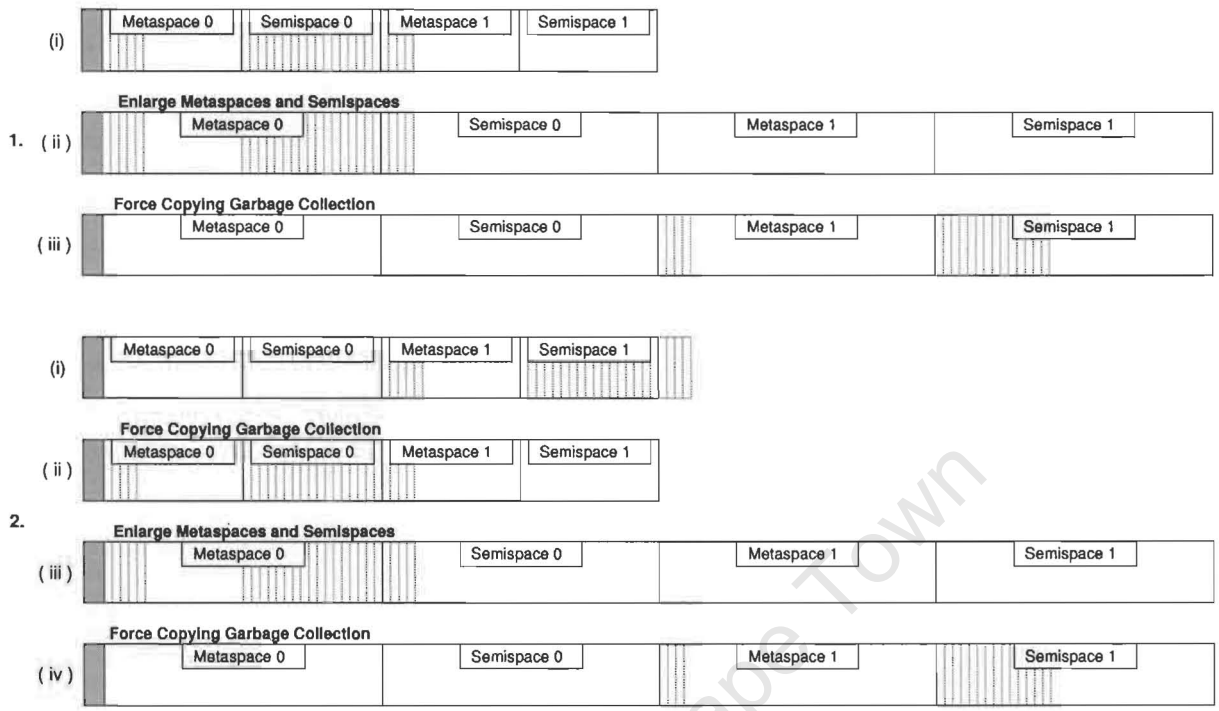


Figure 3.9: Metaspace and semispace size increase.

exhausted, the meta- and semispaces may be enlarged as in case 1 (iii). The process completes by forcing a final garbage collection (iv), copying the live data into the enlarged meta- and semispace.

3.7 Fault-Tolerance

A simple fault tolerance mechanism is used in the semispace copying collector, whereby the "whichspace" value is altered as the last atomic action of the garbage collector. In this way, any failure during garbage collection has no impact on the store (as the collector has changed only the tospace and not the fromspace - i.e. it has not altered the semispace indicated as current by "whichspace").

3.8 Summary

This chapter presented the implementation of a semispace copying garbage collector for the PLaVa store. We have shown how Cheney's copying algorithm is adapted for the garbage collector. Both depth-first and breadth-first traversals and copying were implemented. Results from evaluating the garbage collector under different store configurations will be discussed later in Chapter 5.

Chapter 4

Partitioned Garbage Collection

This chapter describes the implementation of a partitioned garbage collection scheme for the PLaVa store. The collection algorithm is based on the distributed algorithm presented by Maheshwari in [Mah97, ML97]. The implemented partitioned garbage collector is offline (stand-alone) and does not run while programs are accessing the store. Therefore we do not require the need for any mutator-syncrhonisation mechanisms for consistency. We ignore concurrency issues since PLaVa does not support concurrent execution, and fault tolerance is less complex because the garbage collector is not mutator synchronised - it is sufficient to keep a log of changes made by the garbage collector so that recovery of the collected partition is possible in the event of failure. The chapter describes the store organization in sections 4.1 to 4.4.2 and the partitioned garbage collection mechanisms in the remaining sections. The chapter is organized as follows :

- Sections 4.1 and 4.2 describe modifications made to object PIDs, objects and class descriptors.
- Section 4.3 discusses how page information is maintained using a Free Space Table.
- Section 4.4 describes the partitioning of the store.
- Section 4.5 describes how inter-partition references are managed.
- Section 4.6 discusses global marking and phase counters used in garbage collection.
- Section 4.7 presents the invariant and rules on which the collection algorithm is based.
- Section 4.8 describes the independent garbage collection of individual partitions.
- Section 4.9 provides proofs for safety and liveness.
- Section 4.10 discusses the collection of inter-partition cyclic garbage.
- Section 4.11 deals with garbage collection decisions like partition selection policies.

4.1 Persistent Identifier Modification

PIDs are assigned to all persistent objects so that they may be located on the store. For the copying garbage collector, we made use of a handle table to map PIDs to their respective locations. This improved efficiency since when objects were copied during garbage collection, only the handle table required updating.

A handle table with partition and page information for each PID is advantageous if objects may move into other pages, since only the object handles would then need to be updated. For the partitioned store we made the policy decision that objects remain in the pages they were promoted to, so that any clustering mechanisms in place are not compromised. We believe that clustering policies should be an option for a persistent store (e.g. our store can optionally enforce a scheme whereby objects are promoted to pages according to their type) and that clustering concerns should be orthogonal to garbage collection. If objects do not move from the pages they were allocated to, the use of a handle table is inefficient because of the handle table I/O.

We avoid the use of a handle table by storing page and partition information in the object PID itself. Thus PIDs were modified to contain a partition number, a page number and an object number for the page it resides in. Each page has a directory that is used to map a PID's object number to the object address on that specific page. As shown in Figure 4.1, PIDs are 32-bit signed integers, where bits 0-11 store the object number, bits 12-21 store the page number, bits 22-30 store the partition number, and bit 31 is set as required by PLaVa to distinguish PIDs from memory addresses. For example the PID shown in the figure is associated with the object whose object number is 3, and resides in page 4 within partition 2. This configuration allows a maximum of $2^{12} - 1 = 4095$ objects per page, a maximum of $2^{10} - 1 = 1023$ pages per partition, and a maximum of $2^8 - 1 = 255$ partitions per store.

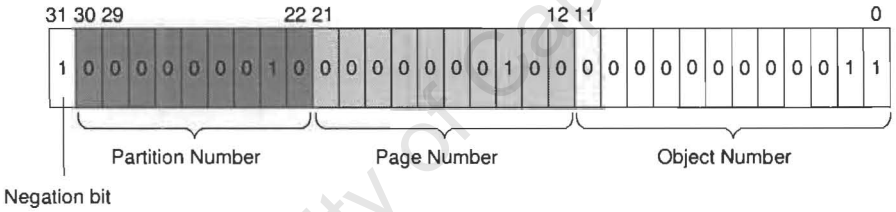


Figure 4.1: Persistent identifier structure.

4.2 Class Descriptors and Object Modifications

Class descriptors used in the copying garbage collector, discussed in the previous chapter, were modified to suit a partitioned store. The copying collector was implemented to store class descriptors in memory at all times. This approach was modified so that class descriptors are treated as objects that reside on the store. This is safer and more flexible as we do not require all the descriptors to reside in memory, as was done with the copying collector implementation.

A store can optionally be configured to retain descriptor sharing but on a partition basis, to decrease the number of inter-partition references. This is achieved by allocating descriptor objects to each partition that contains at least one object of that type, as done in the implementation of PJSL [PAD+97]. The descriptor hash table, which maps class types to their descriptor locations on the store, is extended to allow multiple descriptors residing in different partitions. Whenever an object of a particular type is written to a partition for the first time, a corresponding descriptor for the object is created in the object's target partition. Any objects of the same type that are written to the same partition use the already created descriptor relevant to the partition. When this option is not taken, the default is to have all objects of the same type share a single descriptor for that class on the store.

4.3 The Free Space Table

As explained, a handle table is no longer required because PIDs themselves store partition, page and object number information. A *Free Space Table* (FST) keeps track of the free space for all pages on the store.

The FST comprises a binary file, containing a `freeSpace` variable for each page on the store, as shown in Figure 4.2. `freeSpace` is an integer that stores the amount of free space in the page. When the free space in a page becomes exhausted, the freeSpace variable associated with the particular page is set to 0. Garbage collection compacts objects within pages to eliminate fragmentation. When garbage objects are reclaimed by the garbage collector, the `freeSpace` values in the the FST are updated. In this way, the FST is guaranteed to contain the correct `freeSpace` information for all pages on the store.

Figure 4.2 shows the FST organization and assumes that the corresponding store contains m partitions, where each partition is divided into n pages. Indexing the correct page information in the FST is trivial, and when objects are promoted to a page on the store, the page's `freeSpace` variable is decreased by the object's size.

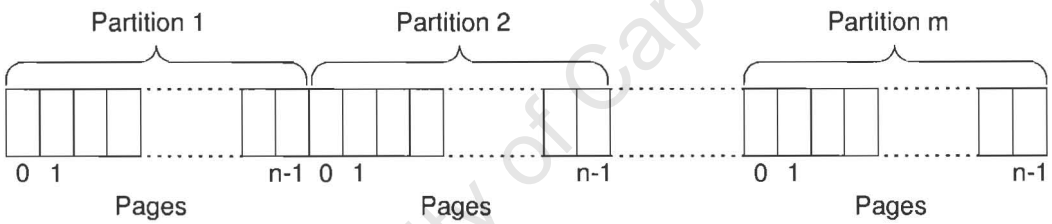


Figure 4.2: The Free Space Table.

4.4 Partitioning of the Store

This section deals with the modifications made to the store in order to partition it. We discuss the store header, pages, pagemaps and partition structure, the store setup process, and store exhaustion and size increase mechanisms.

4.4.1 The Store Header

Since we are partitioning the store, we need to modify the store header to contain information relevant to the partitioning. The information contained in the store header is described in Table 4.1. Later sections in this chapter will make it clear why we need to maintain this information in the store header.

4.4.2 Pages, Pagemaps and Paging

Partitions are divided into pages which are the unit of loading/writing to the store. Here we deal with the structure of pages, including trace- and markmaps, and the directory. Later in Section 4.4.2 we cover

`rootPID`	stores the PID of the persistent root on the store
`numPartitions`	stores the number of partitions on the store
`numPagesPerPartition`	stores the number of pages per partition
`pageSize`	stores the page size
`partition1Start`	stores the location of partition 1 on the store
`globalPhaseCounter`	stores the phase count of the global marking trace
`descriptorHashTableStart`	stores the location of the descriptor hash table on the store

Table 4.1: Partitioned PLaVa store header information.

the paging mechanism itself.

Figure 4.3 illustrates the structure of a page, containing a *pagemap* and an *object space*. We alter Maheshwari's markmaps and refer to them as pagemaps. Markmaps according to Maheshwari store mark and trace bits used for global marking and partition tracing respectively. Our pagemaps are comprised of two first-free offsets (for pagemap entries and for objects), as well as a *tracemap*, *markmap*, and a *directory*. The first-free values are used when pagemap entries and objects are allocated, and are updated according to the pagemap entry size and object size. The tracemap stores trace bits for partition tracing, and the markmap stores object marks associated with global marking. The directory is a mechanism which maps an object's number to a location within the page, so that PIDs do not require updating when objects are moved within pages. The object space is the actual page space that is allocated to objects.

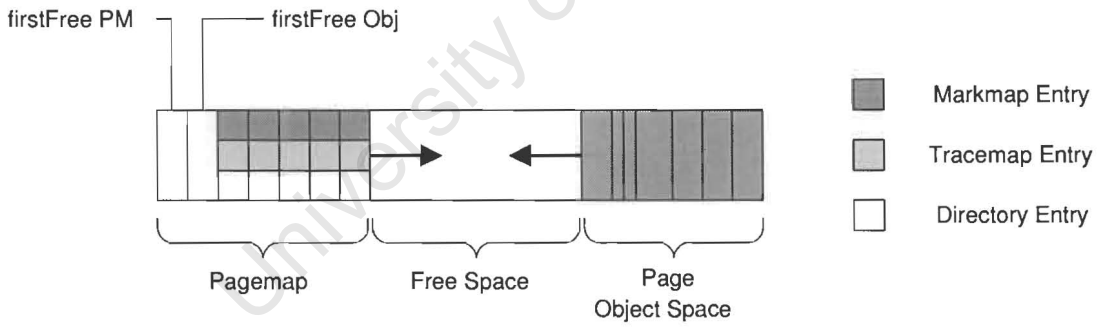


Figure 4.3: Page structure.

Note that pagemap entries are allocated from the beginning of the page towards the end, and objects are allocated from the end of the page towards the beginning. In other words, pagemap and object data grow towards each other. Since the number of pagemap entries may grow, the space usage in every page is maximized. The page is exhausted for a particular object when the difference between the `firstFree Obj` and `firstFree PM` cannot cater for the object's size.

Each pagemap entry, as shown in Figure 4.4, is a 32-bit structure containing two bits for tracing and marking, and a 30-bit integer for storing the object's offset in the object space. We chose to use a 32-bit structure instead of a 16-bit structure, because then the object space may only have a maximum size of $2^{14} - 1$ bytes, an equivalent of 16 KB. Using a 32-bit structure, the maximum object offset within pages is not a problem.

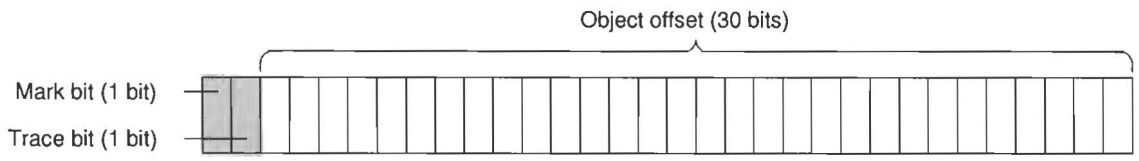


Figure 4.4: A pagemap entry.

Paging

In the original implementation of PLaVa [Tja99], store I/O occurred on a per object basis. Objects were faulted in as required and the corresponding updates to any related data structures were made, for example swizzling PIDs to memory references. In an attempt to make store I/O more efficient we implemented paging to enforce a unit of loading and writing.

Paging for the partitioned PLaVa store was implemented as follows : when the JVM requires an object from the store, its PID is used to determine the partition and page in which the object resides. The page, consisting of a pagemap and an object space, is then loaded from secondary storage into memory. The directory in the pagemap is used to determine the object's offset within the memory, and the object is copied from the memory buffer into the heap. Each object that is loaded into the heap has its reference PIDs swizzled to object references if the objects they reference reside in the heap. The procedure completes by returning a reference to the object that caused the fault. The page remains in memory so that any other objects that the JVM requires that reside in the page may be loaded into the heap. The page is memory resident until the page buffer memory is exhausted and a page is selected for eviction. The page memory buffer equals several pages and facilitates LRU functionality [Jia92].

When a (modified) object is written to the store, the corresponding page is loaded from the store into memory if it is not already in the buffer, and any modified heap-resident objects belonging to the page are written into the buffer. This process involves the swizzling of any object references to PIDs. The page is then written back to the store. Note that PLaVa does not use seperate object cache for persistent objects.

PID Generation and Object Promotion using Markmaps and the FST

When an object is promoted to the store, the Free Space Table is scanned to determine whether the given page (e.g. as per any clustering policy) or any other page contains enough free space for the object. We ensure that a page with enough free space is memory resident and its first free object number is determined. A PID is constructed using the partition number in which the page resides, the page number and the object number, and is assigned to the object. The `firstFree Obj` value in the pagemap is written to the directory using the object number as the index, and determines the offset where the object will be written. `firstFree Obj` is then updated in the pagemap to indicate the promotion of the object, and the `freeSpace` variables are updated in the FST. If no pages with enough free space exists, then a new partition is appended to the store, as discussed in Section 4.4.5 .

4.4.3 Partitions

The aim of dividing the store into partitions is so that each partition may be independently garbage collected in an attempt to reduce garbage collection pause times. The number and size of the partitions is an important issue, since many small partitions cause more inter-partition references, and as a result potentially more inter-partition cyclic garbage. On the other hand, if fewer larger partitions are considered, then more time and memory is required to garbage collect a partition [AFG95].

Partitions are structured as shown in Figure 4.5, which displays a three-paged partition. Firstly the partition's mark bit along with its local phase counter are stored at the beginning of the partition.[1] The translists to the partition (inlists), which store incoming reference information and are required for garbage collection, are written to a page allocated to translists, which is known as as *translist page*.[2] The remainder of the partition is divided into pages comprising pagemaps and object spaces as indicated.

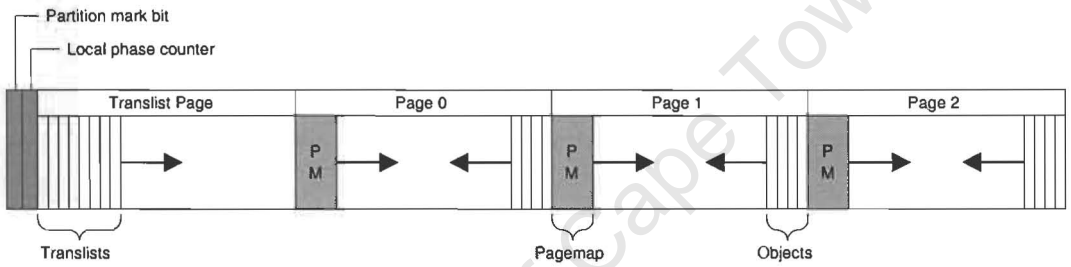


Figure 4.5: Partition structure.

4.4.4 The Store Setup Process

Store creation can optionally be parameterized to define the page size, the number of pages per partition and the number of initial partitions to create.

The store creation process begins with writing the store header information described in Section 4.4.1 to the beginning of the store. The process continues by writing the requested number of partitions to the store, using the partition structure as shown perviously in Figure 4.5. Once all partitions have been written to the store, the `descriptorHashTableStart` variable is set, and the setup process is complete. To recap, the descriptor hash table is memory-resident during execution, and is written to the store at `descriptorHashTableStart` each time the store is stablized and when the store is closed. If the hash table was located at the start of the store, the entire store would need to be moved if the hash table required a size increase. Note that this does not prevent any additions of partitions during program execution, because the hash table resides in memory.

[1] Note that partition mark bits and local phase counters, which are used for global marking, will be discussed later in Section 4.6.

[2] The structure of translists is not important at this stage. Details on the implementation of translists are discussed later in Section 4.5.2.

4.4.5 Store Exhaustion and Size Increase

The store can be exhausted when a new object is written to the store (i.e. when PIDs are allocated to objects) and no page (or no page conforming to some clustering policy) exists that contains enough free space to cater for the object. In this case, the store size is increased by appending a partition to it. The descriptor hash table is loaded into memory (if it is not already memory-resident) and a new partition is appended to the store. Once a new partition is created, the `numPartitions` is incremented to reflect the new partition. The memory-resident descriptor hash table is written to the end of the store and the `descriptorHashTableStart` variable is updated accordingly.

This is the basic approach to increasing the store size. However, matters are complicated by the fact that translists reside at the beginning of partitions. Depending on the type of translist implementation, different store size increase techniques are required. We leave these details for when we discuss translist implementation in the following section.

4.5 Inter-Partition Reference Management

The aim of partitioning the store is to allow for the independent garbage collection of individual partitions. However, partitioning naturally results in the creation of inter-partition references, which need to be considered when performing garbage collection, since they make up the root set for the partition undergoing collection. This section deals with the data structures for storing inter-partition references, as well as an efficient mechanism for maintaining them.

Moss *et. al.* [MMH96] deal with inter-partition references by associating *remsets* (or remembered sets) with each car (page) in a train (partition). The remembered set for a car contains information about incoming references into the car. Distributed environments require similar data structures for maintaining inter-site references, for example Lang *et. al* make use of entry and exit items associated with each node or site [LQP92]. Ferreira and Shapiro use scions and stubs associated with each bunch to record cross-bunch references [FS95]. Maheshwari and Liskov [Mah97, ML97] make use of *inlists, outlists, and translists* to record and maintain inter-partition references, as will be discussed in the following sections.

4.5.1 Inlists, Outlists and Translists

Translists [Mah97, ML97] are used to store information about inter-partition references. We refer to translists as *inlists* if they contain information about *incoming* inter-partition references. Similarly, *outlists* are translists that contain information about *outgoing* inter-partition references.

Figure 4.6 [3] shows how translists may be utilized to record inter-partition references among partitions with the use of inlists and outlists. The figure illustrates an example of a store organization containing three partitions with the indicated objects. Object `A` is the persistent root object and the translists are as shown, with the inlists and outlists for partitions referencing the specific translist(s). We use the following labelling convention for translists: a translist labeled $x - y$ represents the translist comprising all references from partition x to partition y. For example the translist labeled $2 - 1$ represents the set of references from partition 2 to partition 1, and contains the PIDs of objects b and c. Note that inlists and

[3] Note that this figure is apdated from an illustration in [ML97].

outlists are *not* directly represented (they simply reference specific translists) and only translists persist on the store. The figure shows how translists share information between inlists and outlists, i.e. the inlist representing the incoming references from partition 1 to 2 is also the outlist from partition 2 to 1.

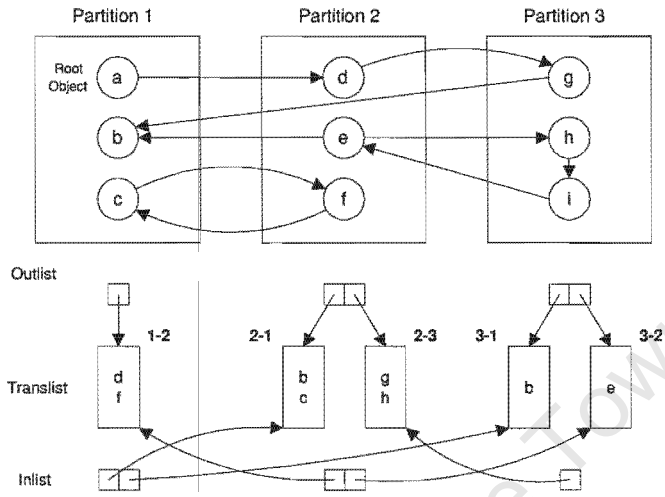


Figure 4.6: Three partitions and their respective translists.

4.5.2 Translist Implementation

We distinguish between translists that reside on the store and memory-resident translists. Memory-resident translists are used for updating stable translists and for determining inter-partition references during garbage collection. When memory-resident translists are no longer required, they are written to the store in the correct partition. The following sections deal with memory-resident and persistent (stable) translists, the updating of translists and translist overflow.

Memory-Resident Translists

Memory-resident translists are implemented as blocked linked lists [ML97], as shown in Figure 4.7. Each translist block contains an array of translist entries indicated by the shaded blocks, which contain the PIDs of the corresponding target objects. The first block of a translist, called an `InitialTranslistBlock`, contains additional information, namely the source and target partitions of the translist, the number of entries in the translist, the first free entry in the translist, and a reference to the next `TranslistBlock`. Subsequent `TranslistBlocks` only contain an additional reference to the next block. We do not require the number of blocks within a translist to be stored in the `InitialTranslistBlock`, since the number of blocks may be calculated (viz. the total number of entries divided by the number of entries per block). Blocked linked lists are dynamic data structures and therefore list sizes need not be fixed.

Translist Organization on the Store

In [ML97], Maheshwari maintains that garbage collection is speeded up by clustering translists of the form $1-x, 2-x, ..., N-x$ within partition x itself. The reason for this is that when a particular partition

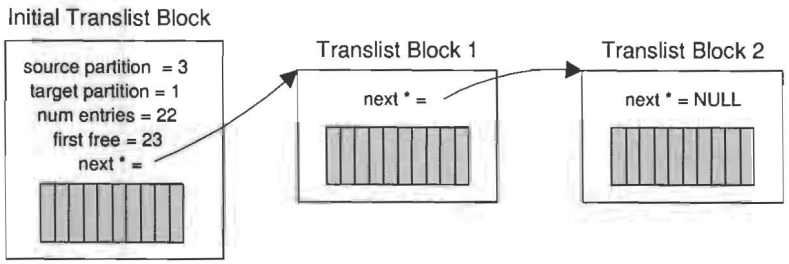


Figure 4.7: A memory-resident translist.

is garbage collected, the root set is determined by scanning all the translists to the partition. Since they are clustered together in the partition, they are all loaded from the store and scanned for potential root objects, which speeds up garbage collection. Figure 4.8 shows the clustering of the translists for Figure 4.6.

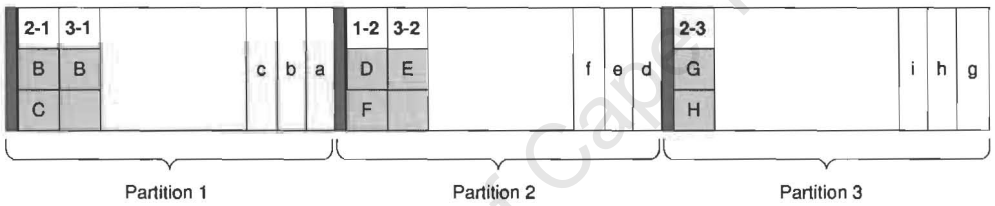


Figure 4.8: Translist organization within partitions.

The implementation of translists occurred in two phases. We began by implementing a simple approach for dealing with inter-partition reference management which avoided translist overflow. As we shall see, this technique wastes spaces on the store and is thus inefficient. The second implementation allocates less space to translists in each partition in an attempt to use store space efficiently. However, this approach requires an overflow handling mechanism, since the allocated space cannot cater for all possible incoming references. We discuss these implementations in detail below.

Translist Organization A

In our initial implementation, translist overflow is avoided by catering for the maximum number of translists into a partition. This was achieved by allocating sufficient translist space to cater for the worst case, i.e. all the objects in a single partition being referenced from all other partitions on the store.

Assuming the store contains n partitions and we are concerned with the translists pertaining to partition 1, the translist in a partition is organized as shown in Figure 4.9. The `InitialTranslistBlocks` of all the translists are written contiguously to the beginning of the partition in order (translist 2-1.0, 3-1.0, ... , n-1.0). Thereafter, the first `TranslistBlocks` of all the translists are written contiguously to the partition in order, thereafter the second `TranslistBlocks` and so on. For example translist 2-1 comprises blocks a, b, and c, and translist 3-1 comprises blocks i, ii, and iii.

With this translist organization, conversion between memory-resident and disk organizations is trivial.

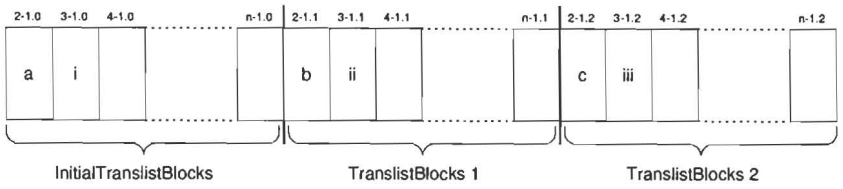


Figure 4.9: Translist page organization A.

The location of any `InitialTranslistBlock` may be calculated using the the source and target partition numbers. Thereafter, the position of the remaining `TranslistBlocks` comprising the translist may be calculated using the sizes of the `InitialTranslistBlock` and `TranslistBlock` data structures.

As explained, translist storage space is the maximum possible so that translist overflow on the store may be avoided. This could restrict the growth of the store for the following reason: when a new partition is appended to the store, the set of translists for each existing partition needs to be enlarged to cater for a larger set of potential inter-partition references from the new partition to all the other existing partitions on the store. Since sufficient translist space has been allocated to each partition before the new partition was appended, enlarging the existing translists is problematic.

A compromise is to allow the store to grow to a maximum number of partitions, say $maxP$. When a new store is created, an initial number of partitions are written to the store, which may be set to one. New partitions are appended to the store as more storage is required, ensuring that the number of partitions $P \leq maxP$. The space allocated to the translists in each partition is calculated according to the maximum number of partitions $maxP$. Therefore when a new partition is appended to the store, the space allocated to translists in existing partitions does *not* need to be enlarged, since their sizes already cater for the new partition. This avoids translist overflow but has the drawback that the store may never have more than $maxP$ partitions. Also space is wasted by catering for the worst case scenario, i.e. the maximum number of inlist references from the maximum number of partitions.

Translist Organization B

The second translist organization implementation does not reserve space to cater for all possible references into a partition, and therefore requires an overflow mechanism. In this way space on the store is used efficiently, but more work is required to deal with overflow. Instead of allocating space for all possible incoming references, each partition has a single page allocated for its inlists.

Assume we have a store with n partitions, and that incoming references from partition 2, 3 and 4 into partition 1 exist. Figure 4.10 shows the translist page for partition 1, containing translists (inlists) 2-1, 3-1 and 4-1. Each translist is preceded with a positive number indicating the "from" partition. No effort is required in storing the "to" partition information, because this may be gathered from the fact that this translist page resides in partition 1. Translist entries are allocated from the beginning of the translist page to the end.

To determine the translists to a partition, the translist page is loaded into memory and scanned from the beginning. When we reach a positive number, we recognize it to be "from" partition information. Any negative entries (PIDs) following a positive, belong to that translist, until we either reach an

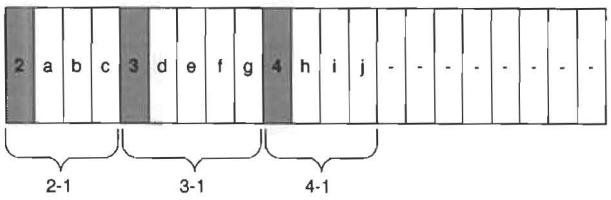


Figure 4.10: Translist page organization B.

empty entry or the beginning of another translist. When translists are written to the store, the relevant translist page is loaded into memory, and overwritten with the information stored in the memory-resident translists. When the memory-resident translist page has been updated, the page is written back to the store.

When a translist page becomes exhausted, we require a mechanism to deal with translist overflow. Moss *et. al.* [MMH96], who use remembered sets to record inter-partition references, suggest using a memory-resident overflow table, or to reference an overflow set stored elsewhere in secondary storage. In our case an entire *translist partition* is appended to the store to deal with translist overflow, as shown in Figure 4.11.

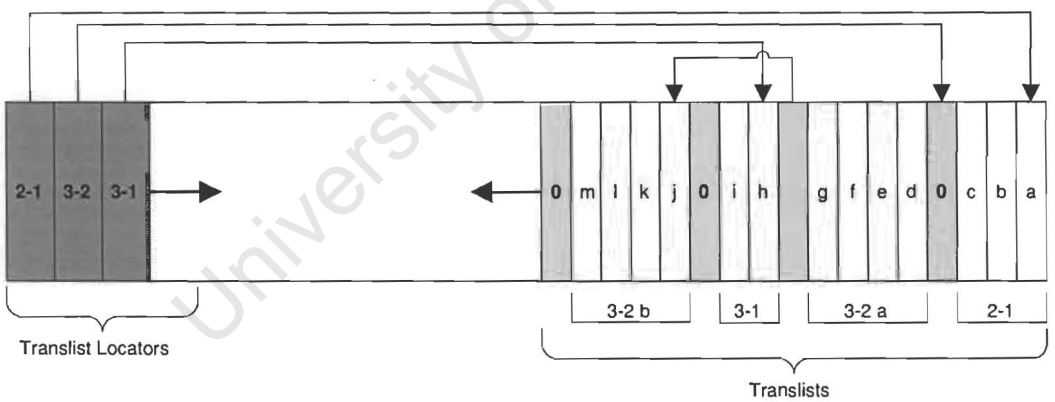


Figure 4.11: The overflow mechanism for translist page organization B.

A translist partition is used to store inlists entries of any translist page that has become exhausted. We therefore need to store both "from" and "to" partition information, in order to distinguish the translists. We make use of a structure called a *translist locator*, which stores the "from" partition, the "to" partition and an "offset" for each translist. The translist locator stores the offset of the beginning of the translist within the partition. Translist entries are allocated from the end of the partition towards the front, and *end bytes* are used to indicate the end of a translist (shown in light grey in the figure). An end byte of zero verifies that the translist has ended, and a positive end byte indicates that the translist continues elsewhere within the translist partition, as it contains the offset

of the continued translist, as shown in Figure 4.11.

The store header and each partition header are extended to incorporate translist overflow partition information. The store header stores the address of the overflow partition (if any), and the partition header stores the address of the overflow partition if the partition's translist overflowed.

If the translist overflow partition becomes exhausted (which is highly unlikely), we first attempt to reorganize the overflow partition by compacting the lists to irradicate any "holes" that might exist. If the overflow partition is still exhausted, another overflow partition is appended to the store, which ensures that there is always enough space for translists.

When objects are faulted from the store, we need to know what partition they reside in. Since we know the size of a partition, the offset of the target partition is easily calculated. To maintain this simplicity, we assume that translist partitions have the same size as ordinary partitions. This avoids the requirement of having a mechanism to maintain the starting offsets of all partitions on the store. However, if translist partitions waste too much store space, the design of a mechanism to set translist partition sizes is possible.

When the translists to a partition are required, the translist page is loaded into memory. The translists residing in the page are loaded into the memory-resident translist. If the overflow number of the partition is positive, we continue loading translists from that overflow partition. We load all the translist locators, and read the translists from those whose "to" partition information matches the partition number.

4.5.3 Delta Lists

Translist updates require the translists to be memory-resident. Considering that the translists for the entire store may become large, keeping them in memory cannot be considered. Instead, translists reside on the store until they require updating, which typically occurs when partitions are garbage collected and when the JVM executes a stabilize. We record new inter-partition references in a separate memory-resident data structure known as a delta list [ML97], so that translist updates may be deferred. Whenever translists require updating, they are loaded from the store and updated using the delta list.

Similar data structures have been implemented in other research concerning partitioned stores. Moss *et. al.* [MMH96] for example, make use of a memory-resident Δref set to record cross-car references. The Δref sets are used to update car remsets at a later time. Maheshwari and Liskov [ML97] implement delta lists as hash tables assigned to each possible translist.

We implement delta lists as blocked linked lists. Delta lists could be said to be memory-resident translists, however the sizes of delta lists depend on the number of inter-partition references created since the most recent translist update. As a result they typically require less memory than permanently keeping translists in memory. Once a partition's translists have been updated, the inter-partition information becomes persistent and the corresponding delta lists have their memory deallocated.

Write Barriers and Delta List Updates

When inter-partition references are created by the PLaVa virtual machine, they need to be recorded in a particular delta list. Only created inter-partition references are considered, since the deletion of inter-partition references occurs during garbage collection.

Write barriers were implemented in the interpreter and in the stabilizing procedure of the virtual machine. The write barriers "trap" created inter-partition references so that they can be inserted into the corresponding delta lists to maintain inter-partition references. We require both an *Interpreter Write Barrier* (IWB) and a *Stablize Write Barrier* (SWB). Since objects are allocated PIDs and are promoted during stabilizing, the SWB is required to record inter-partition references to newly-persistent objects. Together, the SWB and the IWB guarantee that *all* inter-partition reference creations are caught and inserted into the corresponding delta list.

Note that it is trivial to determine whether a reference is an inter-partition reference. This is because each PID contains the partition number of the corresponding object. Therefore we only need to check whether the target PID of a reference has the same partition number as the source object PID which is no hard task since PLaVa keeps the PIDs of objects in their headers.

The IWB need only monitor the JVM instructions that are responsible for creating references for objects and arrays. These are `putfield`, which sets references within objects, and the `xastore` family of instructions, which sets references within arrays [Tja99].

Tjasink's [Tja99] stabilizing procedure was altered to trap *newly* created persistent inter-partition references. The SWB scans an object's/array's persistent references once the object/array has been promoted. This is possible due to the nature of the stabilize procedure, which ensures that when an object promotion completes, its references have been swizzled to PIDs. Any inter-partition references encountered during the scan are caught by the SWB and inserted into the corresponding delta list (unless that delta list already contains the PID).

Delta List Overflow

If the memory-resident delta lists become too large, the largest delta lists are merged with their corresponding translists [ML97] and memory allocated to the delta lists is reclaimed. A variable `DeltaListWaterMark` defines the size to which delta lists may grow, before they are merged with their translists.

Translist Updates

The updating of translists occurs on a partition basis and the updates are performed on memory-resident translists (as shown in Figure 4.7), before they are written to the store. When the translists of a partition require updating, the translist are constructed depending on the translist page implementation as was previously discussed. Delta lists are used to update translists with newly created inter-partition references.

Memory-resident translist insertions are done *explicitly*. The translist is checked whether it contains the entry (PID) to be inserted. If this is not the case, the entry is inserted into the list at the position indicated by the value of `firstFree`. The `firstFree` value is then updated. If a translist linked list is exhausted, the operation causes a new block to be appended to the list.

On the other hand, translist deletions are *implicit*. Translist entries themselves are not deleted, but rather a new translist is constructed to replace the older, out of date translist [ML97]. In brief, before a partition is traced during garbage collection, the partition's current outlists are loaded from the store. The garbage collection trace then constructs a new set of outlists for the partition. Once complete, the current and new outlists are compared, replacing the current outlists with the newer versions if they differ. This ensures the updating of all translists, since we effectively maintain inlists by updating outlists,

according to the definitions of inlists and outlists [ML97]. The removal of translist entries will become clear when we discuss the stages of garbage collection in Section 4.8.

4.5.4 Delta Markmaps

We will see later on in Section 4.8.1 how object marks are propagated during garbage collection to ensure the collection of inter-partition cyclic garbage. Updating marks during a garbage collector trace would be inefficient, since this may involve the updating of many different markmaps (as we call the mark bits in our pagemap structures, described in Section 4.4.2). This requires heavy store I/O, so instead the propagation of object marks is deferred with the use of memory-resident *delta markmaps*.

Delta markmaps are implemented as blocked linked lists of bits. During global marking, as will be discussed in the following section, if any inter-partition references are encountered, the target object's mark bit is set in the delta markmap. Before a partition is garbage collected, its markmaps are updated using the delta markmaps [ML97]. This ensures the propagation of object marks through partitions during garbage collection.

4.6 Global Marking and Phase Counters

To ensure the collection of inter-partition cyclic garbage, Maheshwari implements a global marking strategy [Mah97, ML97]. To minimize overhead, global marking (responsible for propagating global marks) is piggybacked on partition tracing (responsible for distinguishing garbage and live objects in a partition). For this reason we briefly describe global marking before explaining the collection of a partition in Section 4.8. In Section 4.9.1 we explain the termination conditions of global marking, and in Section 4.10 we show how global marking and the termination conditions guarantee the collection of inter-partition cyclic garbage.

At the beginning of a global marking phase, only the persistent root is marked as a starting point for global marking. Tracing the partition containing the root object may cause objects in other partitions to be delta marked. Thereafter, each partition trace propagates marks through partitions. This may cause objects in other partitions to be delta marked, which need to be propagated when the partitions are traced. Global marking is known to be complete when marks have fully propagated through all partitions. This may involve many partition traces, and possibly multiple traces of some partition [Mah97, ML97].

We saw earlier that a global phase counter is maintained in the store header, and that local phase counters are associated with each partition. These phase counters are used in garbage collection as follows : every time a partition is garbage collected, we compare its local phase counter to the global phase counter. If the local phase counter is one less than the global phase counter, we know that this is the partition's first trace in the current marking phase. Any unmarked objects are known to be garbage and may be deleted.

If the local phase counter is even smaller, i.e. local phase counter < (global phase counter -1), it can be assumed that the partition was not traced in the previous phase. In this case, the entire partition may be discarded, since it can be safely assumed that the whole partition is garbage.

When a partition has been traced, its local phase counter is set to equal the global phase counter. When a new global marking phase is started, we increment the global phase counter.

This marking scheme is efficient since global marking is piggybacked on partition tracing and is guaranteed to terminate, as will be shown in Section 4.9.1. Other global marking schemes used in partitioned collection require separate traces for partition and global marking, e.g. [Hug85], and many may not terminate correctly when modifications occur, e.g. [LQP92].

4.7 Algorithm Invariants and Rules

Maheswari's partitioned garbage collection scheme [Mah97, ML97] is based on the Invariant described in Figure 4.12 and the rules shown in Figure 4.13. As we will see, the Invariant and rules ensure that partitioned garbage collection is complete and correct.

A partition is marked after it has been garbage collected, and is unmarked as described by Rule 3 in Figure 4.13. This does not mean that the target objects referenced from marked objects are marked, as delta marks are responsible for propagating object marks across partitions. The Invariant applies to both intra- and inter-partition references and ensures saftey: when the global marking phase terminates, all objects reachable from the persistent root are marked [ML97].

Invariant In a marked partition, all references contained in marked objects are marked or delta marked [ML97].

Figure 4.12: Global marking invariant.

Maheshwari's algorithm is log based, our algorithm is *not*, and so we need some other mechanism to ensure the Invariant is preserved. We use the write barriers discussed in Section 4.5.3 as follows : whenever a created or modified reference is caught by the write barriers, we check whether the partition containing the source object is marked. If it is unmarked, we do nothing, since marks will be propagated when the unmarked partition is traced. On the other hand, if the partition is marked, we delta mark the target object, which may reside in a different partition. Thus we preserve the Invariant.

Maheshwari defines the Rules shown in Figure 4.13 for the termination of a global marking phase. Later in in Section 4.9.1 we discuss proofs of safety and liveness using the Invariant and Rules.

Rule 1 A marked object is never unmarked during a phase.
Rule 2 Objects created during the current phase are marked.
Rule 3 Every time a partition is unmarked, at least one of its umarked objects is marked.

Figure 4.13: Rules for global marking termination.

4.8 Independent Garbage Collection of a Partition

We now deal with the independent garbage collection of individual partitions, which also propagates global marking. In particular, we describe the propagation of global marking with the use of inter-partition reference management data structures. The garbage collection of a single partition occurs in the stages

shown in Figure 4.14, and the following sections describe these stages in more detail. Any reference made to the garbage collection stages refer to Figure 4.14, and the following subsection numbering corresponds to these stages. To recall how global marking propagates through partitions, we reproduce our earlier global marking example in Figure 4.15.

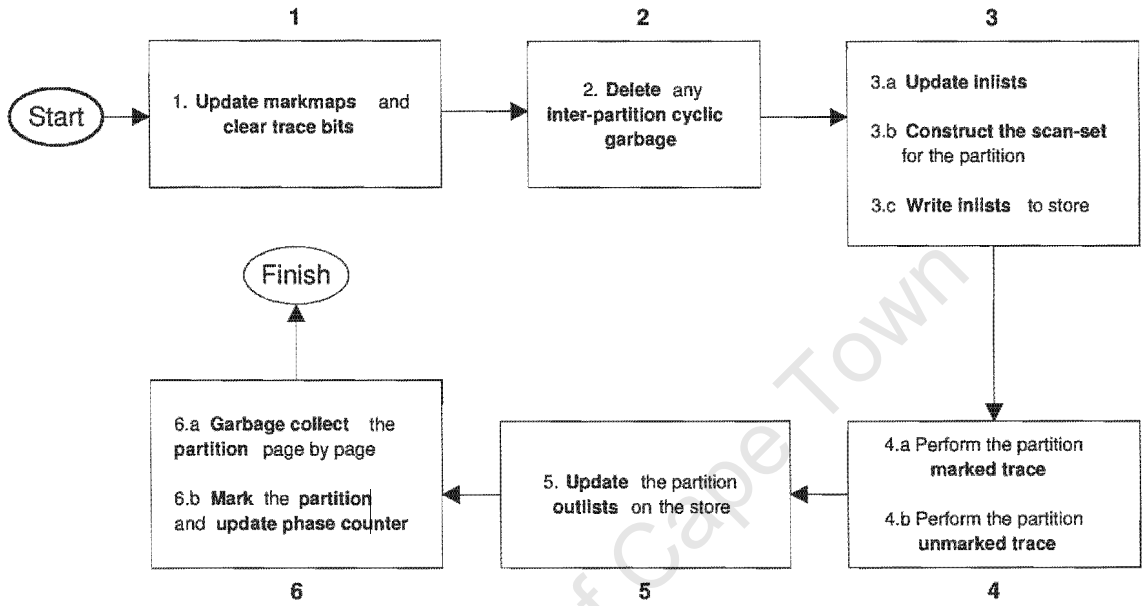


Figure 4.14: The stages of garbage collecting a partition.

4.8.1 Update Markmaps and Clear Trace Bits

Markmap updates are required to ensure that object marks propagate correctly from any earlier partition traces. We update markmaps before tracing since marks need to be correct so that their propagation is correctly done.

The pagemaps residing in the pages of the partition are loaded into an array of pagemaps, called the *PageMap Array* (PMArray). Then this is updated using the delta markmap. Once we have updated the necessary markmaps, we clear the delta markmaps used for updating. We do not write the modified PMArray back to the store. Instead the PMArray is kept in memory for later use

4.8.2 Delete Any Inter-Partition Cyclic Garbage

We use the partition local phase counter to determine whether we may delete unmarked objects. If the local phase counter is one less than the global phase counter, it is known that this is the partition's first trace in the current phase. In this case, any unmarked objects from the previous phase are known to be part of an inter-partition garbage cycle. We temporarily store the PIDs of these garbage objects in a *DelSET*, to ensure that they are removed from inlists in step 3. This makes sure that no inlist entry references a garbage object.

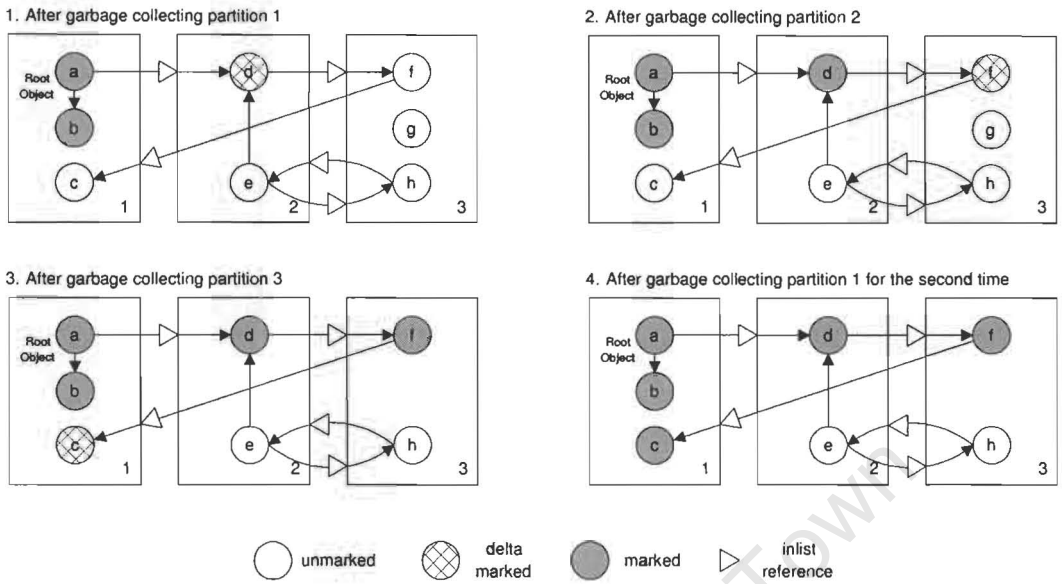


Figure 4.15: Global marking phases propagating marks from the root.

If the local phase counter is even smaller, we know that the partition was not traced in the previous phase and may discard the entire partition. We remove all the inlists and outlists to indicate that the partition is empty.

4.8.3 Update Inlists and Generate Scan-Set

This stage is responsible for updating the partition's inlists and generating the root set or *scan-set* of the partition. Once this is done, we write the partition inlists back to the store, since they are no longer required.

3.a Update Inlists

The delta lists are used to update the inlists of the partition being garbage collected. The partition's inlists are used to construct blocked linked lists as was described in Section 4.5.2 on page 55. For each existing delta list, the corresponding inlist blocked-linked list is updated. The PIDs in the DelSET are then removed from the inlists.

3.b Construct the Scan-Set for the Partition

The inlists of the partition contain object PIDs indicating incoming references that form the root set of the partition. However, inlists to the same partition may contain common entries. Therefore we construct the scan-set, which contains the unique inlist entries along with their mark and trace bits.

The procedure of creating the scan-set is as follows : the inlist entries are scanned, inserting new entries into the scan-set. For each entry in the scan-set, the corresponding mark bits are retrieved from the updated markmaps contained in the PMArray generated in step 1. If we are garbage collecting the

partition containing the store root object and its local phase counter is one less than the global phase counter, we need to mark the root object to initialize a mark starting point of a new global marking phase.

3.c Update Partition Inlists to the Store

At this stage we no longer require the inlists of the partition. We make the inlists modification persistent by writing them to the store as described in Section 4.5.2, and free the primary memory used by the inlists.

4.8.4 Tracing

We now perform the marking and tracing phases of garbage collection to determine the liveness of objects residing in the partition. Two traces are performed: the *marked trace* [ML97] is responsible for propagating global marking and reachability through partitions, by marking and tracing any objects reached. The *unmarked trace* [ML97] ensures that any reachable unmarked objects are preserved during garbage collection.

During the tracing phases, we construct updated versions of the outlists by inserting any inter-partition references reached into new, empty outlists. This ensures that the outlists of a partition are up to date when the collection of the partition completes. Other schemes compare the newly created outlists to the current outlists, replacing the current versions if they differ [LQP92, MBMM98]. We do not perform comparisons, since writing the translists of a partition to the store requires less work and time than first reading the originals and comparing them, as it is unlikely that a siginificant number of disk writes will be saved anyway.

4.a The Marked Trace

The marked trace begins by scanning the entries in the scan-set. For each *marked* entry in the scan-set, an iterative breadth first *marking* trace is performed to propagate global marking. An iterative trace using a queue was used instead of a recursive trace, since recursive procedure calls are neither time- nor space-efficient, and may cause the system stack to overflow [JL99]. Intra-partition references encountered cause mark bits to be set in the PMArray and if these were not traced already, they are added to the queue. Inter-partition references encountered are memorized in the delta markmap.

4.b The Unmarked Trace

The unmarked trace is exactly the same as the marked trace described above, except that the trace uses the *unmarked* entries in the scan-set as starting points and sets only trace bits. For each unmarked entry in the scan-set, an iterative breadth first trace is performed.

It is possible for the unmarked trace to reach marked objects that were not traced during the marked trace. Such objects are known as *marked orphans* [ML97]. Invariant 1, as discussed in Section 4.7 stipulates that references in marked objects are marked or delta marked. However, marked orphans can reference objects that have been traced but not marked, thus violating the Invariant.

Maheshwari and Liskov [ML97] solve this problem by delta marking any unmarked references contained in marked orphans. However, the target object will become an orphan in the next trace. [ML97] suggest speeding up the propagation of marks through orphans, by recursively tracing and delta marking any unmarked reference in an orphan, even if it has previously been traced. We now provide an example to illustrate the marked and unmarked traces, as well as the difficulties cause by marked orphans.

Marked and Unmarked Tracing Example

Figure 4.16 shows an initial partition configuration containing inter-partition references. Figure 4.17 [4] (stages 1 - 4) illustrates the marked and unmarked traces, and how marks are propagated through partitions by global marking.

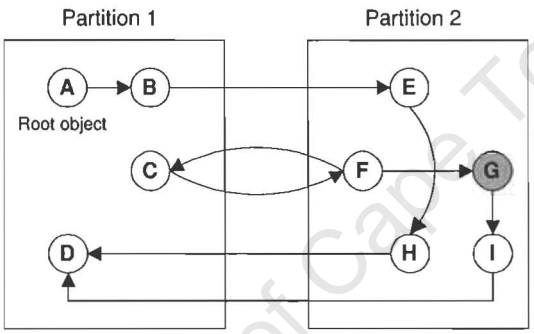


Figure 4.16: Partition configuration before tracing with object G marked.

Partition 1 is traced in Figure 4.17 (1). The marked trace propagates marks and traces from object A to object B, which causes object E to become delta marked. The unmarked trace propogates traces from objects C and D. However, object F is *not* delta marked, since we are performing the unmarked trace.

At some later stage Partition 2 is traced, as shown in Figure 4.17 (2). The marked trace propagates marks from object E, as it was marked in the delta markmap, to object H. This causes object D to become delta marked. The unmarked trace propagates from object F through to I. The result is shown in stage 3. Note that marks were propagated from object G to I. Object G is a marked orphan, since G was not traced in the marked trace. The Invariant is broken, since we have a marked object referencing an unmarked object. As already described, the reference to I is marked, and the marking is propagated recursively, to ensure consistency with the Invariant. Finally in step 4, Partition 1 is retraced to complete global marking. As a result object D is marked in the marked trace, since it was delta marked, and C is traced in the unmarked trace.

The example highlights how marks are propagated through partitions (object D is marked when performing the unmarked trace on partition 1 for the second time). It also indicates the problems caused by marked orphans and a solution to solving inconsistencies produced by marked orphans. The inter-partition garbage cycle comprising objects C and F will be detected and reclaimed in the following global marking phase, as they are left unmarked by the completed marking phase.

[4] Note that Figure 4.17 is apdated from an illustration in [ML97]

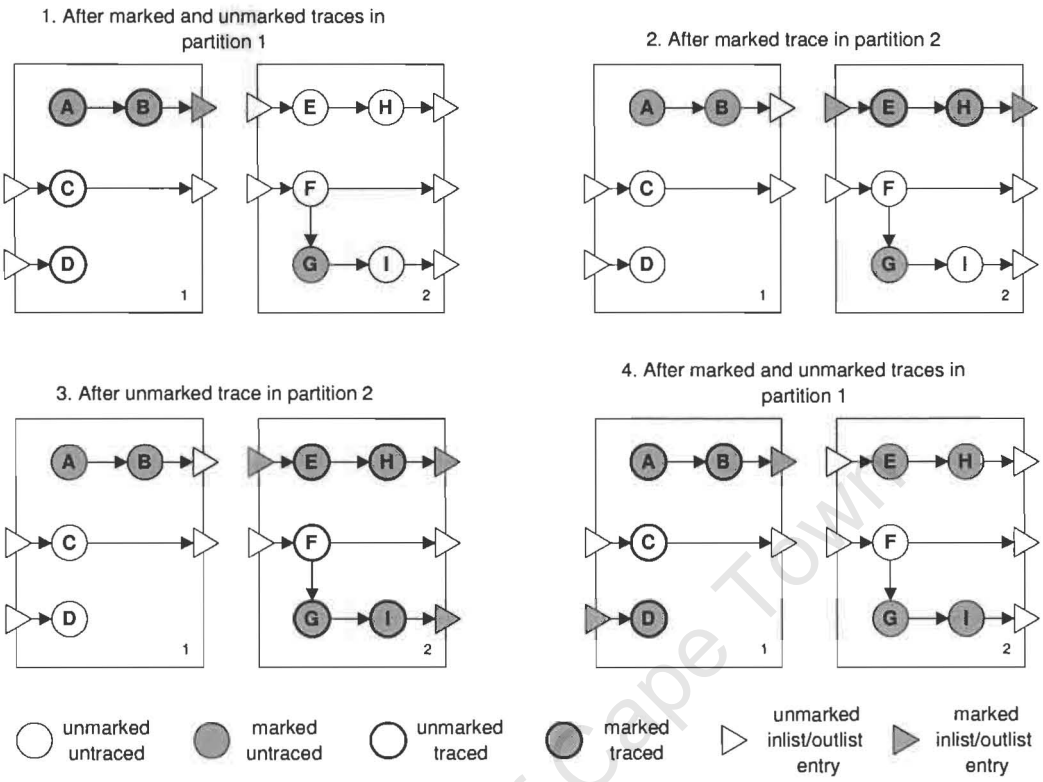


Figure 4.17: Marked and unmarked tracing.

4.8.5 Update the Outlists

Once both the marked and unmarked traces have been performed, we need to update modified outlists on the store. The partition inlists have already been updated in step 3.b, so here we are concerned with writing the newly constructed outlists on the store.

We update the partition's outlists by writing the outlists constructed by the partitions traces. We are required to update the partition's outlists, since by doing so we are effectively updating the inlists of other partition [ML97], according to the definitions of inlists and outlists defined in Section 4.5.1.

4.8.6 Garbage Collect and Mark the Partition

At this stage, the partition markmaps and tracemaps in the PMArray, and the partition inlists and outlists are up to date. All that remains to be done is to update the partition's pagemaps, garbage collect the pages of the partition, mark the partition, and update its local phase counter.

6.a Garbage Collect the Pages in the Partition

By this stage, the markmaps and tracemaps of the partition have been marked and/or traced in the PMArray indicating which objects are live (the untraced objects are known to be garbage). The process of garbage collecting the partition, page by page, may now proceed. Any suitable garbage collection

technique may be used to collect a page, so compaction was chosen. Although compaction is expensive, applying it to small pages within partitions is efficient. Each page in the partition is loaded, its pagemap is updated using the PMArray, and the compaction algorithm makes use of the mark and trace bits and offset information in the updated pagemap to compact the live objects to the beginning of the page. The compaction scheme calculates the amount of free page space. The Free Space Table is then updated to reflect this. Once complete, the garbage collected page is written back to the store.

6.b Mark the Partition and Update Phase Counter

Once the pages within the partition have been garbage collected, the partition is marked to indicate that all the references contained in marked objects are marked or delta marked, according to the Invariant. Finally, the local phase counter is set to the value of the global phase counter, to indicate that is was last traced in the current phase.

4.9 Algorithm Correctness

We have seen in the previous section how global marking is performed incrementally by garbage collecting a partition at a time. However, to prove the correctness of the algorithm, we need to prove safety and liveness. We first specify global marking termination conditions as presented by Maheshwari [ML97], followed by proofs of safety and liveness [Mah97, ML97]. Section 4.10 then demonstrates how global marking and the partitioned garbage collection algorithm guarantee the collection of inter-partition cyclic garbage [Mah97, ML97].

4.9.1 Conditions for Global Marking Phase Termination

Maheshwari defines four global marking conditions that guarantee termination [ML97]. One condition is based on the log to ensure the preseravtion of the Invariant. We discussed how we ensure the preservation of the Invariant eariler in Section 4.7 with the use of write barriers. We summarize the three conditions in the context of the PLaVa garbage collection implementation in Figure 4.18 below.

Condition 1 All partitions are marked.
Condition 2 All delta markmaps have been merged.
Condition 3 All application roots reference marked objects.

Figure 4.18: Global marking termination conditions.

Before a partition is garbage collected a termination check is performed as follows : we check if all partitions are marked, and if so proceed to merge all delta markmaps with the markmaps residing on the store. Finally we check whether all application roots reference marked objects. If application roots reference unmarked objects, we mark the unmarked objects and unmark the partitions they reside in (Rule 3). If any of these condition checks cause a partition to be unmarked, we continue garbage collection until all partitions are marked, and repeat the termination check process. If all the partitions are marked after all termination conditions are checked, the global marking phase is complete.

4.9.2 Safety

To prove safety, it is sufficient to show that all objects reachable from the persistent root are marked when the global marking phase terminates.

Maheshwari [Mah97, ML97] proves safety by using his log-based termination conditions indicated in the previous section. We assume that all partitions are marked (Condition 1). The write barriers ensure that the Invariant is preserved, since the barriers ensure that objects referenced from marked partitions are delta marked. Condition 2 guarantees that there are no references that are delta marked but not marked. Condition 3 then implies that all objects reachable from the persistent root and application roots are marked (Invariant), and therefore preserved by the garbage collector.

4.9.3 Liveness

To guarantee liveness, it is sufficient to show that a partition can be unmarked only a finite number of times. Therefore each partition will eventually become marked, and so global phase termination is guaranteed.

Maheshwari proves liveness [ML97] as follows : Rule 3, as stated in 4.7, stipulates that whenever a partition is unmarked, at least one of its unmarked objects is marked. Rule 2, which states that objects created during the current phase are marked, implies that such an unmarked object was created during the previous phase. Since marked objects are never unmarked (Rule 1) and the partition only has a finite number of unmarked objects, partitions can be unmarked only a finite number of times. This guarantees the termination of global marking. This proof holds for our partitioned garbage collector, since the implementation adheres to the Invariant and the specified rules.

4.10 Inter-Partition Cyclic Garbage

Having discussed the garbage collection of individual partitions, we proceed to show how the garbage collection scheme ensures the collection of inter-partition cyclic garbage.

The Invariant implies that all objects reachable from the persistent root(s) are marked when the global marking phase terminates. Since inter-partition cyclic garbage is not reachable from the persistent root(s), it will remain unmarked and therefore be garbage collected when a new global marking phase starts. However, inter-partition garbage cycles may be marked when global marking terminates for the following reason. Rules 1 and 2 stipulate that marked objects are never unmarked during a phase, and that objects created during the current phase are marked. Therefore objects that were created during the current phase and have been modified to create inter-partition cyclic garbage will be marked. Marked objects comprising inter-partition cyclic garbage will be unmarked in the next phase, and garbage collected when a new phase begins, since they are unmarked when the phase terminates.

4.11 Garbage Collection Decisions

Having discussed the full implementation of a partitioned garbage collector for the PLaVa store in the previous sections, we are now concerned with garbage collection decisions. These include partition selection

policies and when to garbage collect.

4.11.1 Partition Selection Policies

Cook *et. al.* [CWZ94] have studied garbage collection selection policies for object databases in some detail. Their policies are based on the notion that over-written references provide good hints on where to find garbage. Policies 4 and 5 below are two suggested by [CWZ94]. We implemented the following partition selection policies :

1. **First UNMARKED** : selects the first unmarked partition for garbage collection.
2. **UNMARKED Round Robin** : this policy maintains that the required (i.e. unmarked) partitions should be garbage collection in partition-number order. This implementation involved keeping track of the last garbage collected partition. The next partition is selected by finding the next unmarked partition in round robin order.
3. **Random UNMARKED** : randomly selects an unmarked partition for garbage collection. This policy is based on the reasoning that all required (i.e. unmarked) partitions should be given an equal chance to be garbage collected, no matter when they were last garbage collected.
4. **Mutated Partition** : according to Cook *et. al.*, this policy selects the partition in which the most pointers have been updated since the last collection. The reasoning here is that this partition is also likely to contain the most garbage.

   This policy was implemented by associating a mutation counter with each partition. The write-barriers, as discussed in Section 4.5.3, were implemented to increment a partition's mutation counter, when an object residing in the partition has its references mutated. The partition with the highest mutation count is selected.
5. **Updated Pointer** : Cook *et. al* [CWZ94] observe that whenever a reference is overwritten, the object it pointed to is more likely to become garbage. This policy selects the partition with the highest number of overwritten pointers that pointed into that partition. The reasoning is that when a pointer is overwritten, the object that was pointed at may have become garbage.

   The updated pointer policy was implemented using counters for each partition. The Populator is run to populate the store, then is run several times to modify references/pointers in each partition. Only then is garbage collection initiated. As with the mutated partition policy, the PLaVa machine write-barriers were adapted to increment the partition's counter, when an incoming reference is deleted. The partition with the highest count is selected.

The weighted pointer policy [CWZ94] takes into account the weight of over-written references, since over-written pointers nearer to the root of an object graph may produce more garbage than over-written reference closer to the leaves. This is because the database connectivity is potentially greater nearer the root. It was considered too much overhead to supply each reference with a weight, and so this was not implemented.

For interest sake, the policies suggested by Cook *et. al.* were tested on simulation runs over a synthetic database. Munro and Brown [Dav00] tested the same selection policies on a working implementation

using the PMOS Garbage Collector [MMH96], and found that the policies that performed best in Cook's simulations performed badly in their applications.

4.11.2 When to Garbage Collect a Partition?

Stop/start garbage collectors cause severe pause times, and copying and generational garbage collectors need to carefully decide when collection should take place. For generational garbage collectors, Jones [JL99] suggests hiding garbage collections where the user is least likely to notice pauses, or to trigger efficient collections when there is likely to be most garbage to collect. The simulations run by Cook *et. al.* [CWZ94] invoked their garbage collector according to the number of pointer overwrites, which varied between 150 and 300. Their reasoning is that over-written references create garbage.

Further research was done by Cook *et al.* [CKWZ96] concerning how often to perform garbage collection. Two semi-automatic, self-adaptive collection policies for controlling collection rates are described. The policies are based on user preferences. The Semi-Automatic I/O policy attempts to limit garbage collector I/O, while the Semi-Automatic Garbage policy attempts to limit the amount of garbage in the database. These policies were tested using trace-driven simulations. The results show that the choice of collection rate can have a significant impact on application performance and that the "best" rate depends on the dynamic behaviour of the application [CKWZ96]. Implementing and testing such collection policies is beyond the scope of this thesis, and is therefore left to further work.

Incremental garbage collection algorithms attempt to avoid pauses incurred by stop-and-collect techniques [JL99]. In our implementation, if we are dealing with partition sizes that do not require a great deal of memory during garbage collection, then drastic pause times are avoided. Therefore decisions on when to garbage collect are not as crucial.

4.12 Summary

This chapter has presented our implementation of Maheshwari's partitioned garbage collection scheme. We have discussed in detail how the store is partitioned and how inter-partition references are managed, as they are required for partitioned garbage collection. We have explained Maheshwari's algorithm and how global marking ensures the collection of inter-partition cyclic garbage. To give the reader a good idea of how the algorithm works, we explained in detail how partitions are garbage collected independently. Finally we described our implemented partition selection policies.

We are done with issues concerning the implementation the PLaVa garbage collectors, namely a copying and a partition collector. In the following chapter we presents our evaluation result of these collectors.

Chapter 5

Evaluation

Having described the implementation of semispace copying collection and Maheshwari's partitioned garbage collector in the previous two chapters, we now provide an evalutation of the garbage collectors. We describe the test applications that are used for the evaluation, and provide reasons as to why the test application results are useful. The chapter is organized as follows :

- Section 5.1 discusses the test applications used for the evaluation.
- Section 5.2 evaluates the performance of the semispace copying collector.
- Section 5.3 evaluates our implementation of Maheshwari's partitioned garbage collection scheme.

5.1 Test Applications

Here we discuss the test applications used to evaluate the implemented garbage collection algorithms. The evaluation makes use of a specifically-designed, synthetic application. Garbage collections were run on stores of 2MB, 4MB, 8MB and 16MB in size. These were the largest stores possible on our equipment.

5.1.1 Test Database Application

Specifically for the evaluation of the semispace and partitioned garbage collectors, we designed a database application called the *Populator*, written in Java and run over the PLaVa VM. The database application is purpose-made to allow the independent control of different parameters concerning the structure of the object graph residing on the store. With a real application, we are unable to fine-tune these parameters. The Populator exhaustively populates both semispace and partitioned PLaVa stores with homogeneous objects. The stores are populated and modified according to different configurations, for example the amount of garbage or the number of inter-partition references in a partitioned store. We then garbage collect the store and measure the effect of the resulting store characteristic(s) on garbage collection.

For semispace PLaVa stores, the Populator exhausts stores of different sizes with linked lists. The Populator is able to delete elements of linked lists, thus creating non-cyclic garbage. The amount of garbage created is controlled by varying the number of elements that are deleted from each list. In this way the store may be configured to contain varying degress of garbage. In the evaluation of semispace

copying, we do not consider cyclic garbage because copying collection does not differentiate between non-cyclic and cyclic garbage. Cyclic garbage is considered as regular garbage since it is not reachable from the root object(s) and therefore will not be copied during collection.

The Populator is further used to populate partitioned PLaVa stores, containing any number of partitions. Each page in every partition is populated with a linked list of homogeneous objects. The linked lists are setup deliberately to avoid thrashing withing partitions during tracing, thus minimizing I/O that is unrelated to garbage collection mechanisms. The linked lists in each page are modified in a particular manner to create a store with the desired configuration. We intend to determine how garbage affects collection, and the impact of inter-partition references and inter-partition cyclic garbage on collection. The Populator is equipped with the following functionality :

- **create intra-partition garbage**: intra-partition garbage [1] is generated by deleting certain elements from the linked list in every page. For uniform distriubutions, we remove the same number of elements from each list throughout the store. For uneven distributions we vary the quantity of garbage for different pages. The amount of garbage created in each page is determined by setting the specified `PercentageGarbage` variable, which may vary throughout the store depending on the distribution.

- **generate inter-partition references**: by referencing specific elements in a linked list from specific elements in another linked list residing in a different partition, we are able to generate inter-partition references. Each node in a linked list contains two pointers: one references the next node in the linked list, and the other is used to reference nodes residing in other partitions, i.e. for inter-partition reference generation. The amount of inter-partition references into a page are controlled by setting the specified `PercentageIPReferences` variable. The Populator is able to generate both a constant number of inter-partition references on every page and a variable distribution of inter-partition references across the pages of the store.

- **create inter-partition cyclic garbage**: we create cycles by removing specific elements from linked lists residing in different partitions, and create references to link the elements into a cycle. The cycle is generated so that it is reachable from only a single object. Once the cycle is created, the reference to the cycle is deleted thus creating an inter-partition garbage cycle. Assume we remove x elements from each linked list without breaking the references among them. We then join the linked lists in order to generate the cycle. x is known to be the *chain size* of the garbage cycle. We vary the chain size and the number of inter-partition garbage cycles to determine how garbage collection overhead is affected by inter-partition garbage cycles.

Note that the Populator may be configured to create the above conditions uniformly across all pages of the store or unevenly distributed over the store. This allows us to manipulate the clustering or density of intra-partition garbage, inter-partition references and inter-partition cyclic garbage. This can thus determine whether uneven distributions of garbage, inter-partition references and inter-partition garbage cycles have any impact on garbage collection.

We chose to design the distribution types shown in Figure 5.1, so that the Populator functionality categories above could be tested under these seven distribution patterns. The height of the bars in

[1] Intra-partition garbage comprises objects that are unreachable from the root, that do not reference other garbage objects residing in other partitions, and that were not referenced from any object outside their own partition.

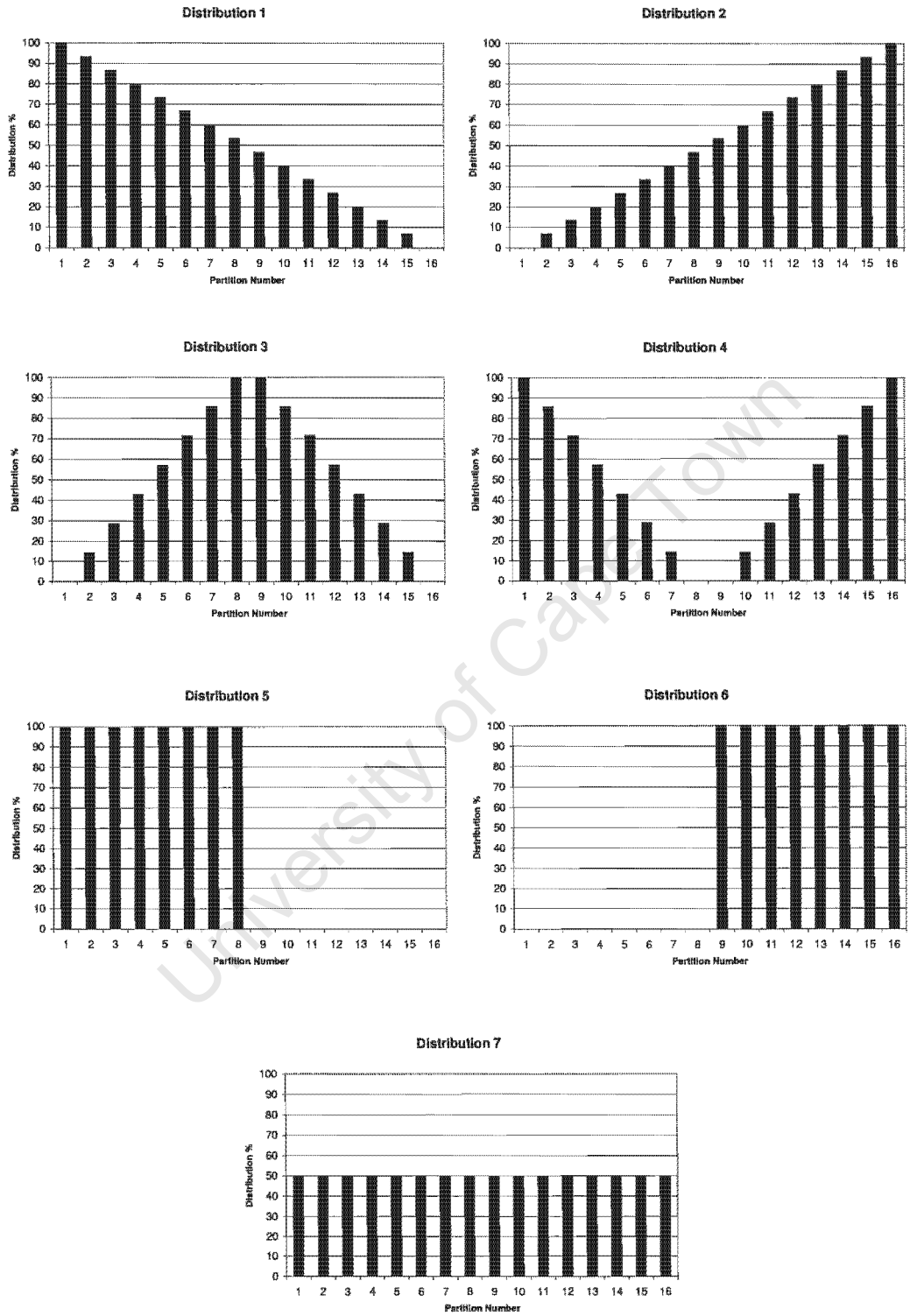


Figure 5.1: Distribution types for a store with 16 partitions.

Figure 5.1 indicate the amount (of intra-partition garbage or inter-partition references or inter-partition cyclic garbage). Distributions 1 and 2 allow us to determine the effect of descreasing and increasing amounts respectively. Distribution 3 ensures the bulk resides in the middle of the store, whereas in distribution 4 this lies at the beginning and the end of the store. Distributions 5 and 6 respectively allow us to determine whether the bulk at the beginning and the end of the store affects garbage collection. Finally, distribution 7 is an even distribution.

Further evaluation results of interest may be to see what effect to the overall store performance some of these distributions have. For example, speeding up garbage collection by 10% might not be desirable if it slows down object-faulting by a factor of two. These types of evaluations are beyond the scope of this thesis, which concentrates on partitioned garbage collection and not object distribution or clustering per se.

As well as implementing these different distributions, we developed three page allocation mechanisms to determine how the clustering of objects affects garbage collection. One of our schemes is to allocate objects according to their type, i.e. an object's location is determined by its type. For this we therefore altered the database application to generate linked lists containing objects of different types. The page allocation schemes are as follows :

- **first-free page allocation**: this page allocation scheme finds the first page on the store that contains enough free space and a free object number, by scanning the Free Space Table, when promoting an object. Descriptors are shared among objects of the same type, so that only one descriptor per object type resides on the store. Any objects that reside in different partitions to their corresponding descriptors naturally create inter-partition references between the objects and their descriptors.

- **partition-based descriptors**: we minimize the amount of inter-partition references by allocating descriptors to each partition, so that all objects of the same type reference the corresponding descriptor in the same partition. Descriptors are shared among objects of the same type that reside in the same partition, and descriptor sharing does *not* cross partitions. As a result, references from objects to their descriptors never form inter-partition references, thereby reducing the number of inter-partition references on the store.

- **object-type page allocation**: the basis for this scheme is to allocate objects according to their type, i.e. an object's location on the store is determined by its type. The scheme incorporates partition based descriptors to minimize inter-partition references and allows us to determine the effect of clustering objects of the same type.

  This scheme required modifications to the PLaVa stablizing procedure and was implemented as follows : assume we have X partitions in a PLaVa partitioned store. We define two categories of object-type page allocation:

  Type A: the database creates linked lists containing X number of different node types of the form `Node01 -> Node02 ->, ..., -> NodeX -> Node01, -> Node 02, ....` The stablizing function was altered to determine the type of a Node object, extract the type number, and allocate the node object to that particular partition. As a result, nodes of type `Node01` are promoted to partition 1, nodes of type `Node02` are promoted to partition 2, and so on. This

scheme will generate large amounts of inter-partition references, as a single linked list is spanned across however many partitions are on the store.

Type B: to minimize inter-partition references, we generate lists of the form `Node01 -> Node01 - > ... -> Node01 - > Node02 -> Node02 - > ... -> Node02 - > Node03 -> Node03 - > ... -> Node03 - > ....` The *cluster size* is the number of objects of the same type that are linked together, which is configurable in our test application.

As a result, we are able to determine whether clustering affects garbage collection by considering the two extremes - Type A is a worst-case scenario and Type B is a best-case scenario. We can also set the clustering to an intermediate case by altering the cluster size accordingly.

Timing Mechanisms

Along with the implementation of the Populator, we incorporated timing mechanisms in order to take measurements for evaluation purposes. We record times spent performing specific garbage collection tasks individually, to see which have a significant impact on overall garbage collection performance.

5.1.2 Hardware Specifications

We briefly describe the hardware specifications of the computer which was used for the evaluation of our PLaVa garbage collectors. The testing computer contained an Intel Pentium II 400 MHz processor accompanied by 128 MB RAM. The motherboard contained a NL440BX/T440BX Chipset (Production Release 9.0 - Build 37) and 512 K onboard cache. The hard drive was a Ultra-SCSI Seagate ST34572N with the following characteristics : 4.55 GB disk size, 7200 RPM in speed, a seek time of 9.4 ms, an average latency of 4.17 ms, and a buffer of 512 KB.

The operating system which was used for programming and testing was Red Hat Linux 7.1 (Seawolf) with kernel version 2.4.2-2.

5.1.3 Correctness

Evaluating garbage collection schemes can produce meaningful results only if we are sure that the scheme is correct and complete. We use a simple yet effective method for ensuring correctness.

For the semispace copying collector, we use the PLaVa Object Graph Traverser (POGT) to determine the number of live objects (and the values of them) in the tospace of the store. Since only reachable, i.e. live, data is copied during collection, we are guaranteed that all garbage is left behind in the fromspace.

For the partitioned garbage collector the POGT is used to determine the live objects (and the values of them) residing on a store before garbage collection. Because of our implementation of PIDs, we are able to determine in which page in which partition the traced objects reside. We maintain object counts per page, so that once the liveness trace is complete, we know the number of live objects residing in each page in each partition. The store is then garbage collected until all garbage is collected. Once collection is complete, we need to ensure that all garbage has been successfully reclaimed, and that all live objects are still accessible, i.e. live objects have not been recognized as garbage. We therefore scan through all the pagemaps on the store, determining the number of live objects in each page by checking object

directories in the pagemaps. If these figures match the liveness trace results prior to collection, then we know that garbage collection was performed correctly and completely. We then run the POGT and print the store contents to determine that the correct objects are live on the store.

We also implemented a simple database application to maintain personal records. The database is accessed by a simple text-based menu, allowing users to query the database, add records, delete records, and modify records. To ensure that all the database functionality works correctly, the user is equipped with a further function to determine the number of records in the database. Therefore we may for example delete 100 records, and ensure that the deletion was performed correctly by checking that the total number of records has decreased by 100, and that the correct data persists.

5.2 Semispace Copying Garbage Collection Evaluation

This section provides the evaluation results for the semispace copying garbage collector. At first we describe the measurement parameters used in the evaluation, as well as explaining why the specific parameters were chosen. We make use of the review on evaluating garbage collectors for persistent storage due to Amsaleg *et al* [AFFS95]. Note that the results were obtained by multiple garbage collection runs and time-averaging to avoid unexpected behaviour.

5.2.1 Evaluation Parameters

For the evaluation of the semispace copying garbage collector, we test the following parameters :

1. **Percentage of Garbage**: the homogeneous database application may be configured to generate any amount of garbage on stores that are populated with objects. This parameter allows us to determine what effect the amount of garbage has on garbage collection.

2. **Store Size**: recall that the `PStore` class contains functionality for specifying store sizes when they are created. We implemented the Populator so that it was able to exhaust stores of a given size; we chose to create stores of sizes of 2, 4, 8, and 16 MB.

   According to Amsaleg *et al.*, the size of a store directly affects the duration of each collection cycle required to garbage collect it. Therefore by garbage collecting stores of different sizes with varying degrees of garbage, we may determine the effect of store size on garbage collection.

3. **Store I/O Time**: this is the time that the garbage collector spends performing store I/O. We regard store I/O as the overhead incurred by the loading and writing of objects to and from the store, as well as time spent performing handle table I/O.

4. **Remaining Time**: this comprises any garbage collection time that is not concerned with store I/O. The remaining time is made up of time used for overheads like reference identification within loaded objects, the tracing of object references, the queue maintenance in breadth-first copying etc.
   .

5. **Garbage Collection Time**: this is the total time required to fully garbage collect a specific store, so that we may compare garbage collection times under different configurations. It comprises the store I/O time plus the remaining time. Since the semispace copying algorithm is closely linked

with the tracing algorithm used to determine the liveness of objects, we do not distinguish between garbage collection trace time and collection reclaim time, as suggested by Amsaleg *et al.*

5.2.2 Depth-first and Breadth-first Semispace Copying

To evaluate the effect of depth- versus breadth-first copying on garbage collection, we used the Populator to exhaust semispace stores of differing sizes with 128 byte objects. The stores were altered to contain varying degrees of garbage, as previously described. We then garbage collected the stores using both depth-first and breadth-first copying algorithms, recording the garbage collection times in each case.

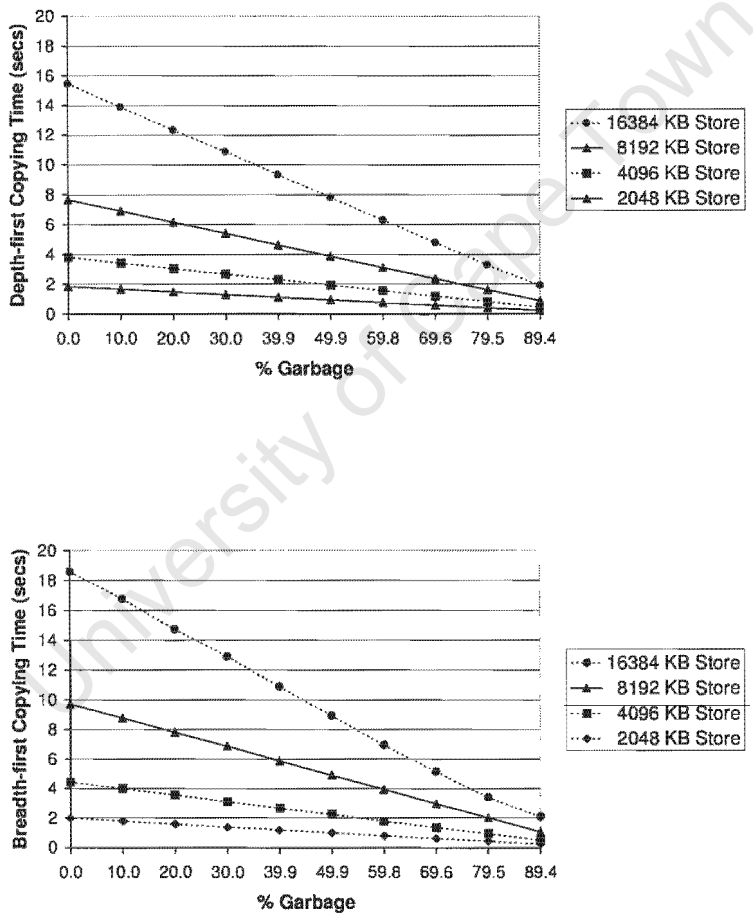


Figure 5.2: The effect of garbage on depth- and breadth-first copying garbage collection.

Figure 5.2 shows the garbage collection times for depth-first and breadth-first copying invoked on stores with different sizes, containing different percentages of garbage. The results show that in both cases the garbage collection time decreases linearly as the percentage of garbage increases for all store sizes. This is because the copying overhead is proportional to the amount of live data, i.e. the work

involved in copying collection decreases linearly as the amount of garbage increases, since less live objects are copied.

Another interesting point to mention is the fact that all readings collection times are significantly reduced when large degrees of garbage are present. This indicates that the mechanisms for maintaining the store for garbage collection are minimal, and that copying depends on the amount of live data and not on the size of the store. We will see later in Sections 5.3.2 and 5.3.3 that partitioned stores of given sizes still perform a substantial workload during garbage collection, even when large amounts of garbage are present.

Figure 5.2 further indicates that garbage collection for breadth-first copying requires more time than depth-first copying when garbage collecting the same store with the same amount of garbage. This is due to the additional overhead required in maintaining the breadth-first copy queue. Another reason is the spatial locality of objects: the PLaVa stabilizing procedure operates in a depth-first fashion and since we populate the store with large linked lists, two adjacent objects in a list will reside at adjacent locations on the store after stabilization.

The linearly decreasing nature of both algorithms is further observed in Figure 5.3. The figure shows depth-first and breadth-first copying times according to collection overhead break-downs for a 16 MB store. The results indicate that breadth-first copying requires more time than depth-first copying when collecting the same amount of garbage on the same store. Store I/O for breadth-first copying is larger as explained above, and store I/O decreases linearly since the number of object loads and writes decreases as the amount of live objects decreases.

The remaining time is due to the work required to identify references within objects and the marking of reached objects during tracing. We observe that the breadth-first remaining time is larger than the remaining time for depth-first copying. This is due to the fact that breadth-first copying has the additional overhead of maintaining the copy queue. Note that in both cases, the store I/O contributes a greater proportional to the total collection time compared to the remaining time. This indicates that the overhead in loading and writing objects and the maintenance of the handle table is larger than the work required for reference identification, object tracing, and queue maintenance in the case of breadth-first copying .

Finally in Figure 5.4 we directly compare depth-first and breadth-first copying on a 16 MB store with varying percentages of garbage. We include these results to indicate the difference in depth-first and breadth-first copying times.

5.3 Partitioned Garbage Collection Evaluation

In [ML97], Maheshwari and Liskov present a comprehensive design for garbage collection in partitioned stores. They implement and evaluate the data structures and algorithms that are responsible for inter-partition reference management. Their evaluation concentrates on the performance of inter-partition reference maintenence mechanisms. We evaluate the partitioned garbage collection scheme as a whole and first describe the measurement parameters relevant to partitioned garbage collection, before proceeding with the evaluation. Note that the results were obtained by multiple garbage collection runs and time-averaging to avoid unexpected behaviour.

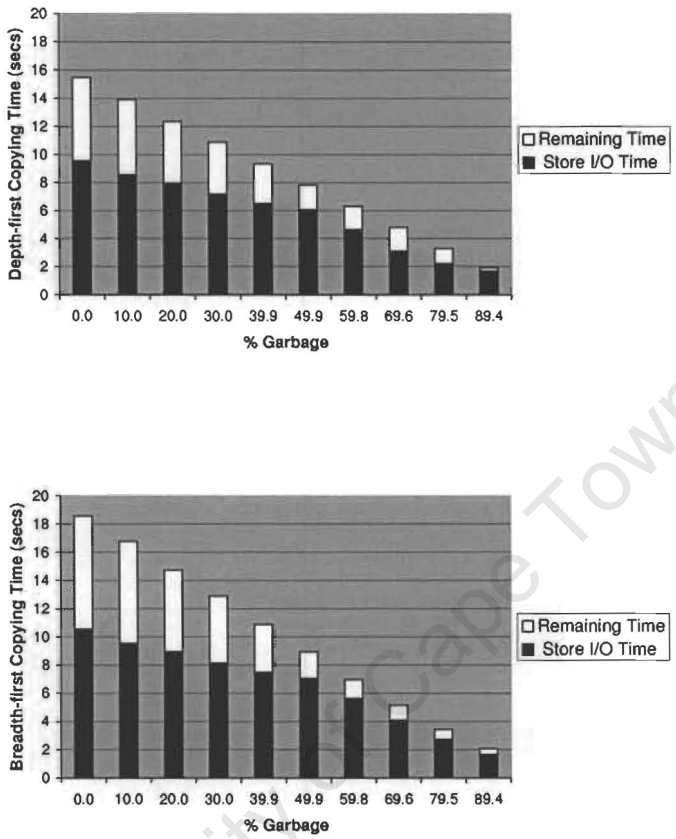


Figure 5.3: Depth- and breadth-first copying collection overhead break-down for a 16 MB store.

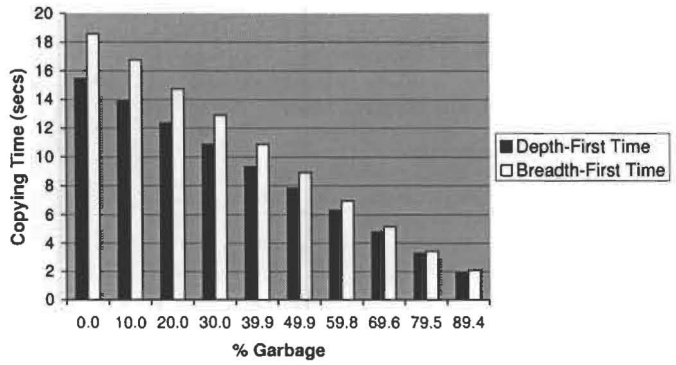


Figure 5.4: Depth- and breadth-first copying collection overhead for a 16 MB store.

5.3.1 Evaluation Parameters

For partitioned garbage collection, we test for the following parameters :

1. **Translist Management Time**: is the time the garbage collector spends performing translist management required during garbage collection. This includes the reading/writing of translists between the disk buffer and the memory heap, as well as the time used to update the outlists of a partition. Note that this time does not include the loading and writing of the translist memory buffer to and from the store. The work required for those operations is considered as store I/O, since we are loading and writing data from and to the store.

2. **Pagemap Management Time**: is the time the collector spends performing pagemap management required during garbage collection. It includes the clearing of pagemaps before collection, the updating of pagemaps using delta lists before garbage collection (to ensure the propagation of global marking), and the updating of pagemaps during the collector's marked and unmarked traces. Note that we regard the I/O concerned with the loading and writing of pagemaps to and from the store as store I/O, and not pagemap management time.

   This evaluation parameter will assess the use of pagemaps for maintaining object mark and trace bits (used for global marking and partition tracing respectively).

3. **Trace Time**: since the partitioned garbage collection algorithm distinguishes between tracing and garbage reclamation phases, we measure the time taken for tracing. Trace time comprises the time required for identifying references within objects and the tracing of object references.

   We record this parameter to determine how the overhead of tracing is affected by the amount and type of garbage. Note that the overhead for pagemap maintenance during tracing is regarded as pagemap management time, and the overhead of loading and writing pages to the store is considered store I/O.

4. **Reclaim Time**: once the partition has been traced, the pages in the partition are individually garbage collected. The reclaim time therefore refers to the time taken to reclaim garbage within pages on the store. Note that the loading and writing of pages is considered store I/O, and that the reclaim time includes scanning through pages compacting objects within the page.

   Measuring trace and reclaim times may highlight bottlenecks in the implementation of low-level mechanisms [AFFS95]. We may also determine how reclaim time is affected by the amount and type of garbage.

5. **Store I/O Time**: this involves all the store I/O used for partitioned garbage collection. This comprises translist memory buffer I/O, the I/O concerned with loading and writing pagemaps to and from the store, the I/O required in paging, and Free Space Table I/O.

   We measure store I/O to determine how garbage amounts and the number of inter-partition references affects store I/O.

6. **Remaining Time**: this comprises the remaining time, i.e. the difference between the total time and the time used for translist management, pagemap management and store I/O, and for tracing and reclaiming (i.e parameters 1-5). The remaining time comprises overheads like delta markmap access, the construction of the DelSET, etc.

7. **Garbage Collection Time**: refers to the total time required to rid a partitioned store of garbage.

Our evaluation is performed on stores under different configurations and sizes. Unless otherwise stated, each partition is configured to be .5 MB in size. To create larger stores we simply increase the number of partitions.

We have described the measurement parameters used for evaluating the partitioned collection scheme. We now present the results for partitioned garbage collection, which indicate how garbage collection is affected by intra-partition, inter-partition and inter-partition cyclic garbage, and the effect of page size, partition size, partition selection policy and object placement strategy on collection. The next few sections give measurements for a store using translist organization A (with fixed sizes catering for the maximum possible number of inlist references); thereafter the performance using a store with translist organization B (single-page translists) is supplied.

5.3.2 Intra-Partition Garbage

The test application was configured to populate partitioned stores of sizes 2, 4, 8, and 16 MB, where each partition contained 64 pages of 8 KB in size, and each page contained 62 objects of 128 bytes in size. The Populator exhausts each page on the store with a linked list, which is then modified to setup a particular store configuration. We generate garbage within each page by removing objects from the linked lists. The generated garbage is known as *intra-partition garbage*, since the objects do not contain any references to garbage objects residing in other partitions and were not referenced from outside their partition. Recall that translist implementation A was used in this test case.

Evenly Distributed Intra-Partition Garbage

Figure 5.5 shows that garbage collection time decreases linearly with an increase in evenly distributed intra-partition garbage. Since fewer objects are traced and compacted during collection when more garbage is present, less time is required for garbage collection. Maheshwari's results show that garbage collection time increases exponentially with an increase in *inter-partition* references [ML97]. Because the number of inter-partition references is insignificant under this configuration, time spent performing translist management may be disregarded thus eliminating the exponential factor. Not only is translist management overhead insignificant, it is constant.[2]

Unlike semispace copying, garbage collection time is *not* proportional to the store size as the large store requires longer garbage collection time than expected. This is due to the fact that translist management and store I/O increase with store size. Note that the results in Figure 5.5 do *not* converge as the amount of live data approaches zero (cf. semispace copying). This indicates that the collection algorithms and related mechanisms perform a large amount of work, even when high amounts of garbage are present. This overhead is amplified as the store size increases, due to the fact that pagemap and translist maintenance becomes increasingly heavier. This is true since we increase the store size by increasing the number of partitions, and therefore translists maintained on the store (using translist implementation A) inherently become larger, thus requiring greater overhead to maintain them.

[2]Each page contains a single linked list and the list head object is never deleted when creating garbage. Since the references to head objects are the only inter-partition references, the number of inter-partition references remains constant.

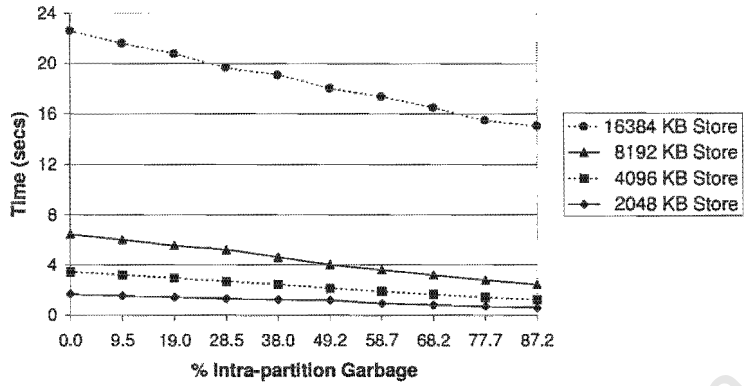


Figure 5.5: The effect of intra-partition garbage on partitioned garbage collection.

In Figure 5.6 we show garbage collection overhead break-downs for the 2 and 16 MB partitioned stores in Figure 5.5 (note the different time scales). Translist management time is insignificant as the stores are configured to contain very few inter-partition references, and therefore little work is required to maintain the inter-partition reference state of the store. Pagemap management is minor and decreases by the slightest degree as the amount of garbage increases. This is true since fewer objects are traced when more garbage is present, thus requiring less work to set trace bits. Little overhead is required for propagating marks and updating translists, since in this case very few inter-partition references exist.

The trace times in both cases decrease linearly as the amount of intra-partition garbage increases, since fewer objects are traced when the amount of garbage inceases. This means that fewer objects are loaded when large amounts of garbage exist, which explains the decreasing nature of the store I/O time show in Figure 5.6.

Lastly we note that the times recorded are all greater than semispace copying garbage collection times (cf. Figure 5.2).

Unevenly Distributed Intra-Partition Garbage

The test database application was used to exhaust stores of 2, 4, 8, and 16 MB in size with linked lists. The store was then altered to generate unevenly distributed intra-partition garbage, according to the distribution types discussed previously in Section 5.1.1.

Figure 5.7 shows the results. Garbage collection times are similar across the distribution types. Since all distribution types contain the same amount of live objects and garbage objects, overheads like reclaiming, tracing, and pagemap management remain constant. The impact of translist management is insignificant since very few inter-partition references exist on the store.

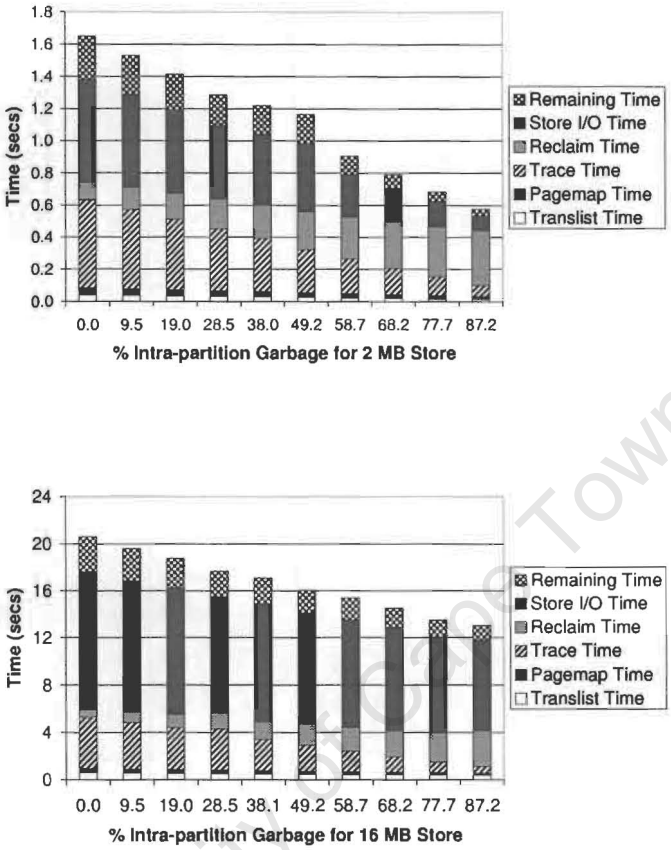


Figure 5.6: Partitioned garbage collection overhead break-down for a 2 MB and 16 MB store containing varying degrees of intra-partition garbage.

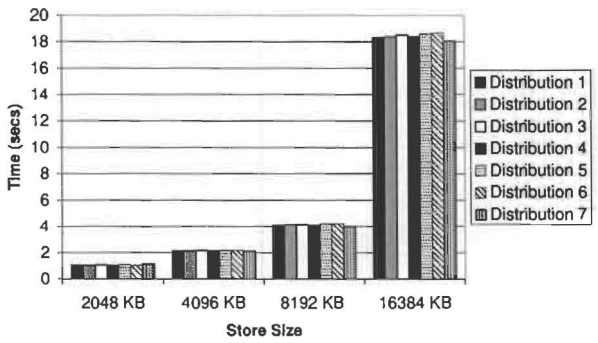


Figure 5.7: The effect of unevenly distributed intra-partition garbage on partitioned garbage collection.

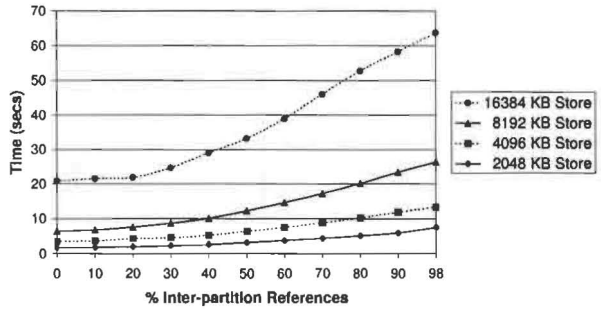


Figure 5.8: The effect of inter-partition references on partitioned garbage collection for the worst case scenario (no garbage costs).

5.3.3 Inter-Partition References

The foundation for Maheshwari's partitioned garbage collection scheme are the mechanisms used to deal with inter-partition references [ML97]. Therefore it is important to determine how inter-partition references affect garbage collection. The Populator was used to exhaust 2, 4, 8, and 16 MB store with varying degrees of evenly and unevenly distributed inter-partition references. Each partition on the store contained 64 pages of 8 KB in size, and each page contained 62 objects of 128 bytes in size. This was accomplished by linking objects residing in a page in one partition with objects residing in a page in another partition. We consider the worst case scenario when the stores contain no garbage, since we are purely determining the effect of inter-partition references on garbage collection. Note that translist implementation A was used in this test case.

Evenly Distributed Inter-Partition References

Figure 5.8 indicates that garbage collection time increases as the amount of inter-partition references increases. The greater the amount of inter-partition references, the more overhead is incurred in maintaining the inter-partition state of the store with the use of translists, and as a result the longer the garbage collection time. The results indicate that with small translist sizes the increase is slow, whereas with large translist sizes the increase is steeper. This is because translist management and maintenance becomes heavier with an increase in translist size. Maheshwari's results are similar [ML97].

In Figure 5.9 we show the garbage collection overhead break-down for the 2 and 16 MB stores shown in Figure 5.8 (note the different time scales). Of special note is the increase in translist management time for both stores, while all other times increase very slightly or remain relatively constant for each store. As explained earlier, more inter-partition references incur more overhead to maintain translists. All other readings remain constant as the stores contain no garbage. Therefore the same amount of pagemap management and store I/O occurs in all cases, and the trace time increases slightly, as expected since the inter-partition references incur extra maintenance costs. The reclaim time is minimal as no compaction overhead is incurred since the store contains no garbage. Note that in each case, the store I/O remains

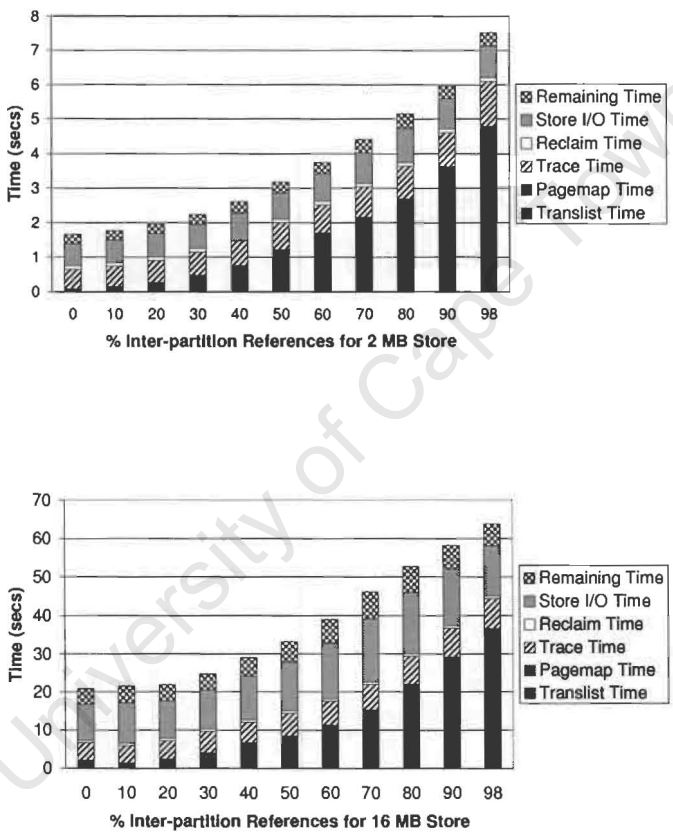


Figure 5.9: Partitioned garbage collection overhead break-down for a 2 MB and 16 MB store containing varying degrees of inter-partition references.

constant, since the same number of translists of the same size are loaded in a complete collection phase.

Lastly, a comparison with the values recorded for stores with zero percent garbage in Figure 5.2, shows that collection times for the partitioned store are noticeably greater than for semispace copying.

Unevenly Distributed Inter-Partition References

The test application was used to exhaust stores of 2, 4, 8, and 16 MB in size with linked lists. The store was then altered to generate unevenly distributed inter-partition references, according to the distribution types discussed previously in Section 5.1.1.

Figure 5.10 shows that distributions 1 - 4 and distribution 7 remain relatively similar for all store sizes, while distributions 5 and 6 require longer garbage collection times. Although all distributions contain the same amount of inter-partition references in total, distributions 5 and 6 contain partitions that are completely exhausted by objects reachable via inter-partition references. This concentration of inter-partition references causes translist overhead to increase. This is because if inlists are very large then collection performance is worse, even though the total number of inter-partition references on the store is unchanged.

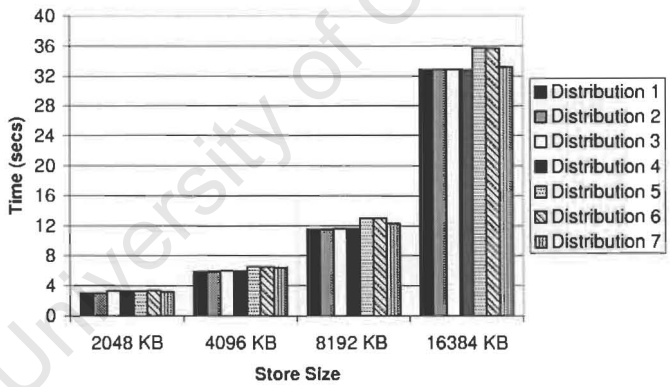


Figure 5.10: The effect of unevenly distributed inter-partition references on partitioned garbage collection.

5.3.4 Inter-Partition Cyclic Garbage

Maheshwari's partitioned garbage collection makes use of a global marking scheme to ensure the reclamation of inter-partition cyclic garbage. To evaluate the effect of inter-partition cyclic garbage, we created stores of different sizes and used the Populator to generate varying degrees of inter-partition cyclic garbage, as described earlier in Section 5.1.1. In this case, we only generate inter-partition cyclic garbage, i.e. no other garbage is created, and the stores were then garbage collected until all garbage was reclaimed, and the times recorded. Recall that translist implementation A was used in this test case.

Evenly Distributed Inter-Partition Cyclic Garbage

Figure 5.11 shows how inter-partition cyclic garbage affects collection overhead. The amount of inter-partition cyclic garbage was specified by varying the number of inter-partition garbage cycles while all the cycle chain sizes were kept constant at 1. This means that as we create a cycle of inter-partition references (that we subsequently turn into garbage), we link in just one object on each page we visit.

The results indicate that garbage collection overhead increases proportionally to the amount of inter-partition garbage cycles. The overhead for smaller stores increases slightly, whereas the overhead involved in garbage collecting the 16 MB store increases steeply. Since no other inter-partition references are present besides those created due to inter-partition cyclic garbage, the number of inter-partition references increases as the number of inter-partition garbage cycles increases.

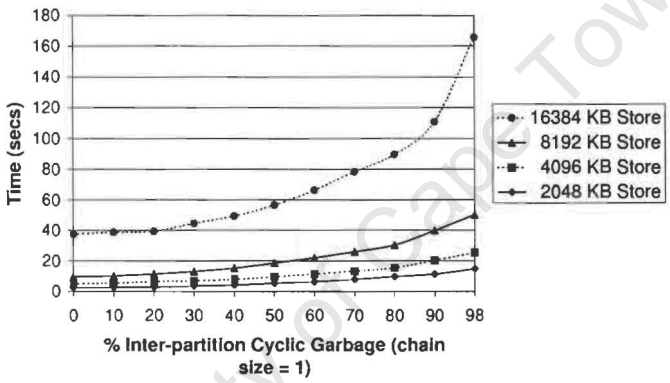


Figure 5.11: The effect of inter-partition cyclic garbage with a chain size = 1 on partitioned garbage collection.

The garbage collection overhead times in Figure 5.11 are approximately double of those recorded for inter-partition references in Figure 5.8 for the following reason : the store requires two full global marking phases to successfully reclaim inter-partition cyclic garbage. This is true since inter-partition cyclic garbage is recognized as live during the first phase, since inlist entries for the cycles exist. Only once entries are left unmarked during the next phase will the inter-partition cyclic garbage be detected and reclaimed.

In Figure 5.12 we show the garbage collection break-downs for the 2 and 16 MB stores shown in Figure 5.11 (note the different time scales). In both cases translist management time increases significantly as the number of inter-partition references caused by the inter-partition cyclic garbage increases. The first phase of collection is expensive, as many inter-partition references must be traced during collection. The second phase essentially removes the inter-partition references as the inter-partition garbage cycles are reclaimed. Therefore the translist management time shown here is slightly less than double the translist management readings for inter-partition references shown in Figure 5.9.

We further observe in Figure 5.12 that pagemap management time increases as the number of inter-partition garbage cycles increases. This is due to the fact that pagemap management increases when

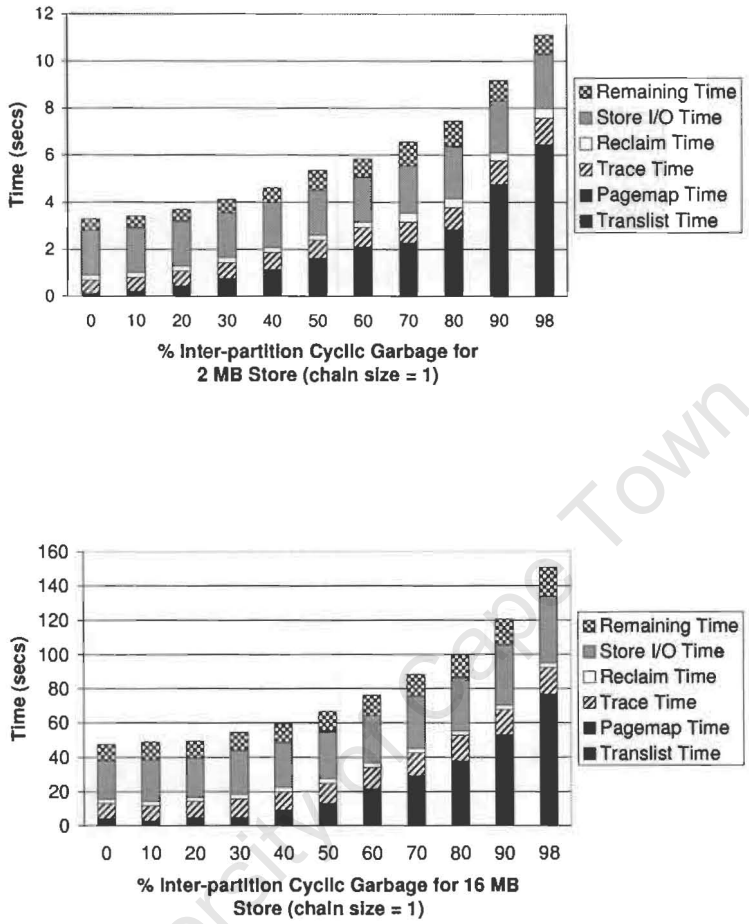


Figure 5.12: Partitioned garbage collection overhead break-down for a 2 MB and 16 MB store containing varying degrees of inter-partition cyclic garbage with the chain size = 1.

inter-partition garbage cycles are traced in the first global marking phase, since more objects are involved. The trace and reclaim times decrease slightly for the following reason: in the first global marking phase the same number of objects is traced, since the inter-partition cyclic garbage has not been recognized as garbage yet. Also no objects are reclaimed in the first phase. However, in the second marking phase fewer objects are traced and fewer objects are moved during reclamation, causing a decrease in trace and reclaim overhead.

To determine the effect of inter-partition cyclic garbage chain size on collection overhead, we kept the number of inter-partition garbage cycles on the store constant, but varied the number of objects from each page that make up each section of an inter-partition garbage cycle. We thus keep the number of inter-partition references formed by inter-partition cyclic garbage constant, but increase the number of garbage objects that comprise the inter-partition garbage cycles. The results for the collection overhead

under these parameters is shown in Figure 5.13. As before, two global marking phases are required to fully collect all garbage.

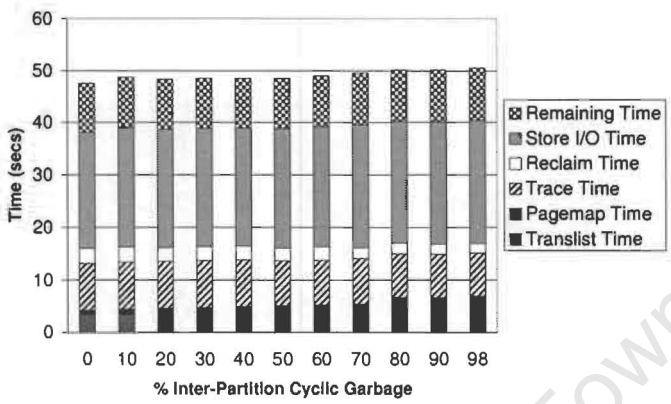


Figure 5.13: Partitioned garbage collection overhead break-down for a 16 MB store containing inter-partition cyclic garbage with varying chain sizes.

The figure suggests that garbage collection overhead does not depend that much on the amount of inter-partition garbage, but is strongly influenced by the number of inter-partition references that inter-partition cyclic garbage comprises. This also explains the virtually constant readings for translist management.

The pagemap management time increases because the amount of garbage detected and reclaimed in the second global marking phase increases. The trace times decrease slightly, as fewer objects are traced in the second global marking phase. Finally, the store I/O readings remain constant because the same number of pages are read and written to the store each time.

Comparing Figures 5.12 and 5.13 with Figure 5.2 we again see that partitioned garbage collection time exceeds that of semispace copying, with the difference becoming increasingly evident as the amount of garbage increases.

5.3.5 Page Size and Partition Size

Our partitioned stores are configurable in page size, page number and number of partitions, so that we can determine how these characteristics affect garbage collection. Stores with particular page and partition configurations were created and exhausted using the Populator. We then modified object references to ensure that the stores contained 50 percent garbage, without generating any inter-partition references. Note that translist implementation A was used in this test case.

Figure 5.14 shows the results of testing the effect of page size and the number of pages per partition on garbage collection time, keeping the partition size constant The results show that garbage collection time increases as the store is configured to contain more pages that are smaller in size. This is true since the overhead in loading fewer larger pages is less than the overhead involved in loading a larger amount

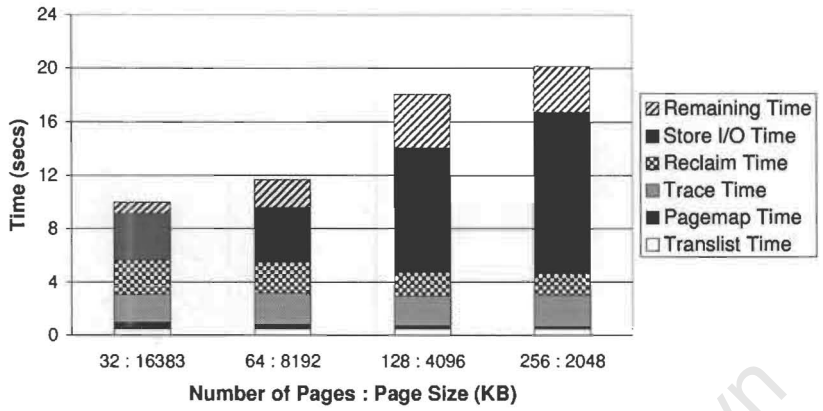


Figure 5.14: The effect of page size and number of pages on partitioned garbage collection overhead for a 16 MB store.

of pages that are smaller in size. Translist management time remains constant because the store contains few inter-partition references and the number of partitions is constant. This is true since translists apply to partitions and not pages.

Figure 5.15 indicates how the number of partitions and the partition size affect garbage collection for the same store configuration. The store size remains constant by maintaining the same page size and number of pages on the store. Translist management time increases with an increase in the number of partitions and becomes significant for the store with 64 partitions. More partitions inherently require more translists on the store requiring more overhead to maintain them. The partitioned scheme copes with 32 partitions, but becomes more of a bottle-neck with 64 partitions. Stores with more partitions that are smaller in size require more store I/O to complete garbage collection since more translists are read/written during collection.

All other measurements are expected to remain relatively constant for the following reason. We keep the total number of pages in each store configuration constant. Since pagemaps are associated with pages, pagemap management overhead should be minimally affected because the same number of pagemaps are being maintained during a garbage collection phase. The trace times are relatively unaffected since the same total number of objects is traced during a collection phase. Similarly for reclaim time, since each loads an equal number of pages in a collection and the amount of garbage is kept constant.

Of special note is the drop in time for the store configuration with 16 partitions and 128 pages per partition. This measurement was taken several times and always yielded a similar result. Possible explanations are that the store configuration is highly suited to partitioned garbage collection, thus allowing garbage collection to perform efficiently, or (more likely) that this page size is best for the performance of disk I/O, particularly since I/O is less.

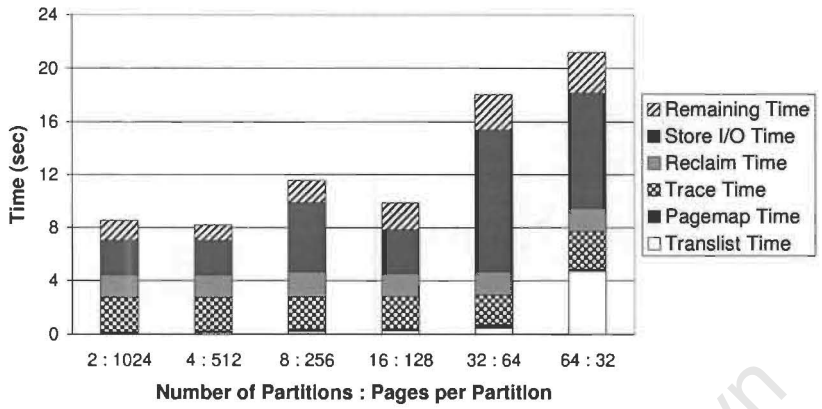


Figure 5.15: The effect of partition size and number of pages per partition on partitioned garbage collection overhead for a 16 MB store.

5.3.6 Translist Evaluation

Our initial translist implementation catered for the maximum number of possible inter-partition references into a partition. This was done to avoid translist overflow and was referred to as Translist Organization A (TOA). However, space on the store is wasted when catering for this worst case scenario. The evaluation results thus far presented made use of this translist organization.

We now present evaluation results for our second translist organization, as dicussed in Section 4.5.2 in the previous chapter, and referred to as translist organization B (TOB). In each case the identical store configuration was used as in the evaluation for TOA, thus allowing us to compare the two translist implementations.

To recap, TOB allocates a single page to each partition to store inter-partition reference information. This page is referred to as a translist page. We cater for overflow by allocating a translist overflow partition when a partition's translist page is exhausted. Any other partitions whose translist pages are exhausted make use of the same translist overflow partition for storing inter-partition reference information. In the case that an overflow partition is exhausted, another is allocated to deal with further overflow. In this way space on the store is used conservatively.

The problems with TOA are that a large $maxP$ (the maximum number of partitions the store can grow to) will increase the translist size on the store, as might the fact that it reserves space for the maximum possible number of inter-partition references. The former problem can clearly have a big impact on garbage collection time; to gauge the affect of the latter problem we ran our test cases using TOB on identical store configurations to the tests run with TOA. That is, the page size, partition size and store size (with $maxP$ set to 32, there being 32 partitions of .5 MB in size on our 16 MB store) were retained for our tests of translist organization B. Note that the translist time in each organization should be equal, since this measurement concerns translist manipulations when the translists are memory resident. Hence

it is only store I/O time that can potentially be affected.

Intra-Partition Garbage

Figure 5.16 shows the results for determining the effect of intra-partition garbage on partitioned garbage collection using TOB.

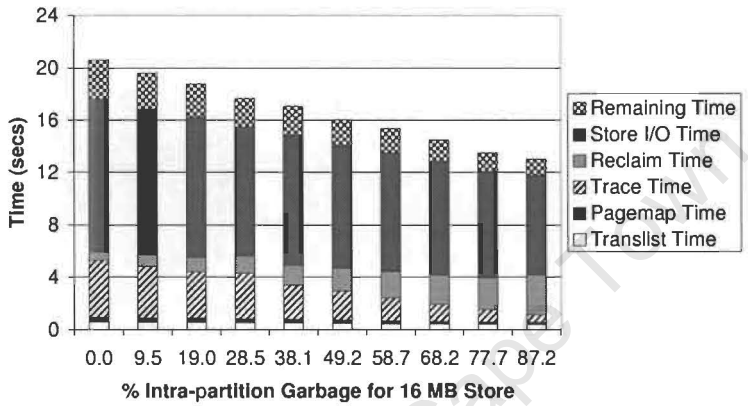


Figure 5.16: Partitioned garbage collection overhead break-down for a 16 MB store containing varying degrees of intra-partition garbage using translist organization B.

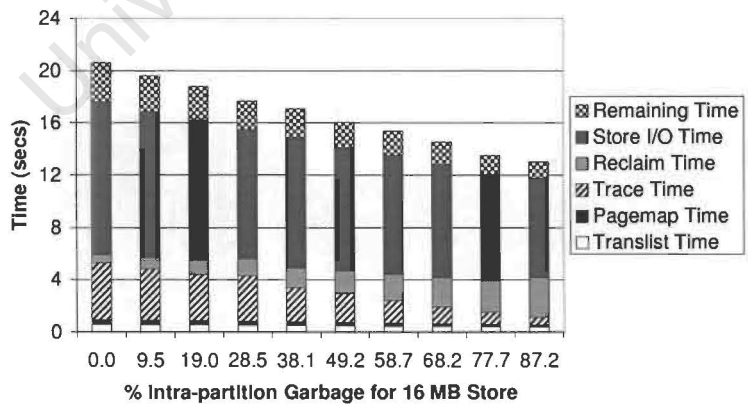


Figure 5.17: Partitioned garbage collection overhead break-down for a 16 MB store containing varying degrees of intra-partition garbage using translist organization A.

Compared to the same tests run with TOA (Figure 5.17), the total garbage collection times are similar.

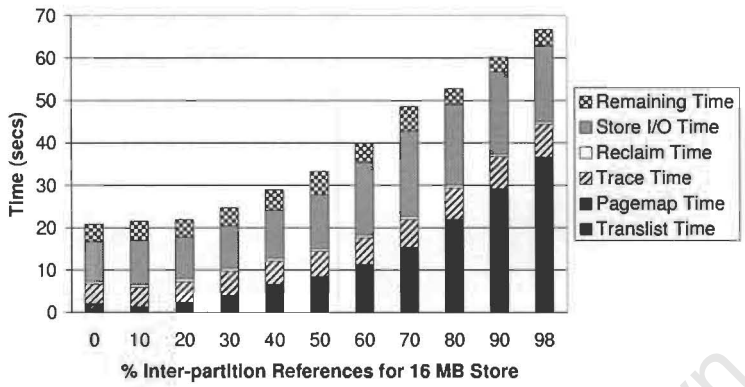


Figure 5.18: Partitioned garbage collection overhead break-down for a 16 MB store containing varying degrees of inter-partition references using translist implementation B.

Most measurements for the two implementations are similar. This is because very few inter-partition references exist on the store. Therefore this test case is largely independant of translist implementation. Both organizations require the same amount of store I/O because the same number of pages is loaded from the store.

Inter-Partition References

The evaluation results for TOB with regard to various degrees of inter-partition references on a 16 MB store is shown in Figure 5.18. The store contained no garbage, as we are determining the effect of inter-partition references on garbage collection.

Since the store contains varying degrees of inter-partition references, TOA (shown in Figure 5.19) and TOB should exhibit different evaluations results. TOB performs better where the proportion of inter-partition references is reasonable (half or less). Stores containing 60 percent inter-partition references or more required more time to garbage collect when using TOB. This is because an overflow partition is used for storing overflowing inter-partition reference information. As a result more pages are loaded from and written to the store.

Inter-Partition Cyclic Garbage

We determined the affect of inter-partition cyclic garbage using TOB on partitioned garbage collection. The results are shown in Figure 5.20.

TOA (shown in Figure 5.21) and TOB require the same amount of translist management time, since the organization affects translists on the store, and not in memory where they are manipulated, which is the translist management time. Again we have the characteristic that store I/O increases slightly for

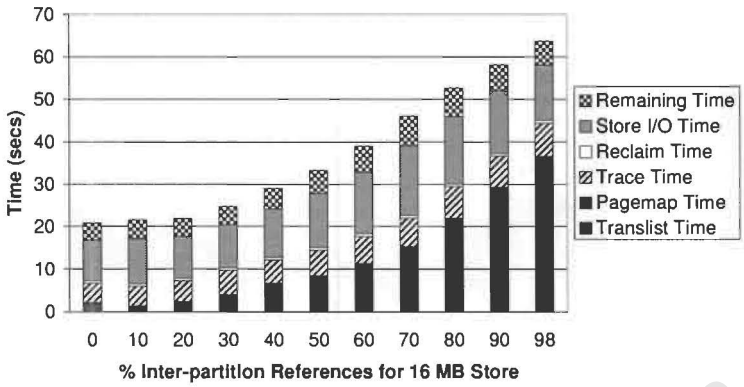


Figure 5.19: Partitioned garbage collection overhead break-down for a 16 MB store containing varying degrees of inter-partition references using translist implementation A.

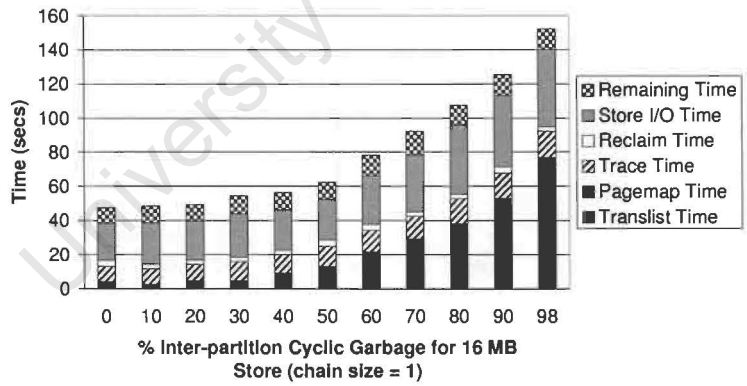


Figure 5.20: Partitioned garbage collection overhead break-down for a 16 MB store containing inter-partition cyclic garbage with varying chain sizes using translist implementation B.

TOB when we have more than 60 percent garbage. This is due to the store I/O required to access the translist overflow partition. The remaining measurements are similar in both organizations since they are independant of translist organization on the store.

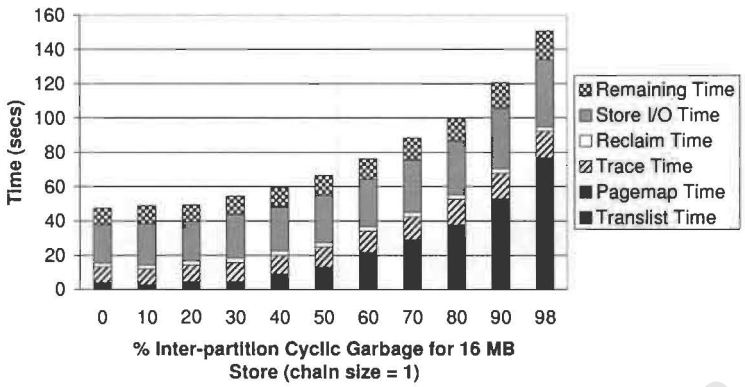


Figure 5.21: Partitioned garbage collection overhead break-down for a 16 MB store containing inter-partition cyclic garbage with varying chain sizes using translist implementation A.

Partition Size

We tested the effect of TOB and partition size on partitioned garbage collection, and we show the result in Figure 5.22.

As the figures shows, all measurements remain similar for both translist organizations. The only significant difference in an increase in store I/O using TOB for stores with 32 and 64 partitions. This is due to the fact that more partitions inherently create more inter-partition references, thus requiring translist overflow partitions. More partitions require more partition garbage collections and more I/O to load/write translists to and from the store.

5.3.7 Partition Selection Policies

Another part of our evaluation examined the effect of different partition selection policies on garbage collection overhead. We evaluate the selection policies discussed in the previous chapter in Section 4.11.1. Translist implementation A was used in this test case.

Partition Selection Policies on Evenly Distributed Stores

A 16 MB store was exhausted and the Populator was used to generate 30 percent garbage. The store was also altered to contain 20 percent of inter-partition references.

The results in Figure 5.24 show that, besides the random selection policy, the mutated partition selection policy performs the worst. This confirms the results of Cook *et. al.* [CWZ94] since this heuristic is based on the partition with the most reference modifications, and not the partition with the most amount of garbage. The flaw with this scheme is that both reference creations and modifications are considered as mutations. However, reference creations are not correlated to garbage creation. The

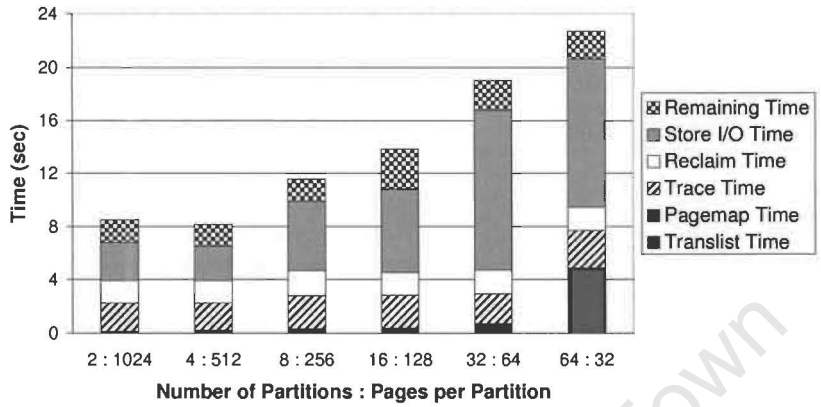


Figure 5.22: The effect of partition size and number of pages per partition on partitioned garbage collection overhead for a 16 MB store using translist implementation B.

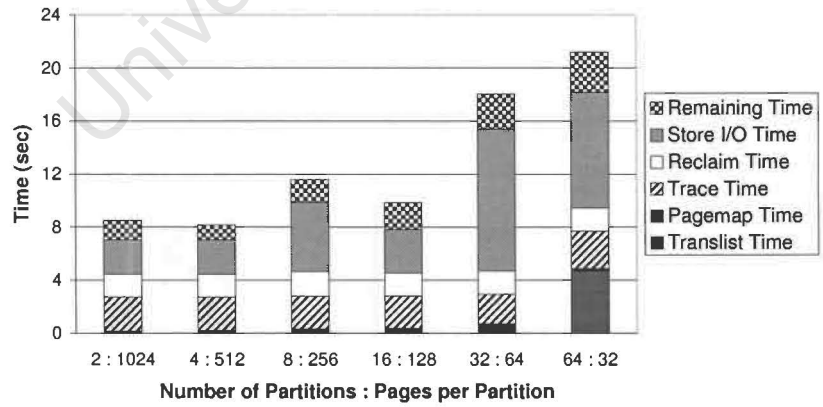


Figure 5.23: The effect of partition size and number of pages per partition on partitioned garbage collection overhead for a 16 MB store using translist implementation A.

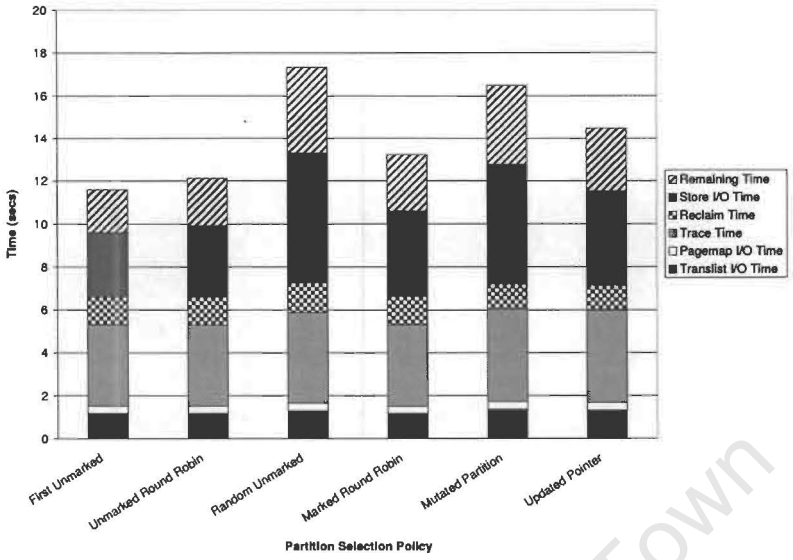


Figure 5.24: The effect of partition selection policy on partitioned garbage collection overhead for a 16 MB store.

updated pointer policy performs slighty better because this policy keeps track of the number of reference modifications for each partition, excluding the creation of new objects.

The remaining policies are not based on reference modifications. The results indicate that the first unmarked selection policy performs best. This is due to the fact that the inter-partition reference propagation in our test application object graph is mainly forward, with very few backward pointers.[3] The first unmarked policy adheres to this forward characteristic and begins garbage collection from the "earliest" unmarked partition.

The unmarked round robin selection policy requires more time than first unmarked, since the tracing of a partition X may cause the unmarking of a partition Y, where $X < Y$. The first unmarked selection policy would collect the first unmarked partition, but the round robin will continue tracing partitions in round robin fashion, which requires more partition traces. Finally, the random unmarked policy performs relatively well, considering partitions are selected according to a random heuristic. Randomly selecting partitions does not provide any useful insight into garbage collection overhead. However, using a random partition selection policy ensures that all partitions are given an equal chance of being garbage collected [CWZ94]. It also provides a point of comparison when evaluating the other policies.

Partition Selection Policies on Unvenly Distributed Stores

We investigated the effect of unevenly distributed intra-partition garbage, inter-partition references and inter-partition cyclic garbage on garbage collection time using the defined partition selection policies.

Our test application was used to exhaust 16 MB stores with linked lists. The store was then altered to generate unevenly distributed intra-partition garbage, inter-partition references, and inter-partition

[3] A forward inter-partition reference is one where a pointer residing in an object in partition X points to an object in partition Y, and $X < Y$.

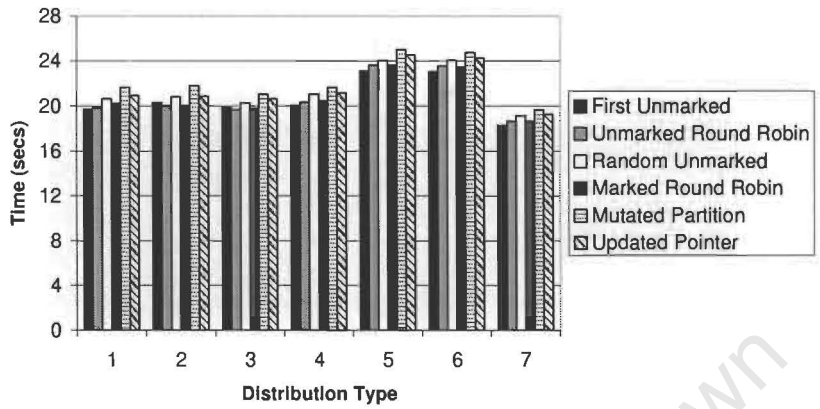


Figure 5.25: The effect of unevenly distributed intra-partition garbage on garbage collection based on partition selection policies for a 16 MB store.

cyclic garbage, according to the distribution types discussed previously in Section 5.1.1. We then garbage collected the stores using the defined partition selection policies.

Unevenly Distributed Intra-Partition Garbage and Partition Selection Policies

Figure 5.25 shows the results of garbage collecting the different distributions of intra-partition garbage using the defined partition selection policies on a 16 MB store. Observing each distribution individually, we notice that the relative performance of the partition selection policies correspond to our results for even distributions (Figure 5.24).

Unevenly Distributed Inter-Partition References and Partition Selection Policies

Figure 5.26 shows the results of performing garbage collection using the partition selection policies on stores with different distributions of inter-partition references. As with our results for partition selection policies on stores containing intra-partition garbage in the previous section, garbage collection times have similar relative performance across the distributions.

Unevenly Distributed Inter-Partition Cyclic Garbage and Partition Selection Policies

Finally we investigate the affect of distributed inter-partition cyclic garbage on garbage collection using the defined partition selection policies. The results for distributed inter-partition cyclic garbage are shown in Figure 5.27.

When comparing distributions, distrbutions 5 and 6 again perform the worst because they contain partitions which are completely exhausted by garbage and distribution 7 (even distribution) again performs the best.

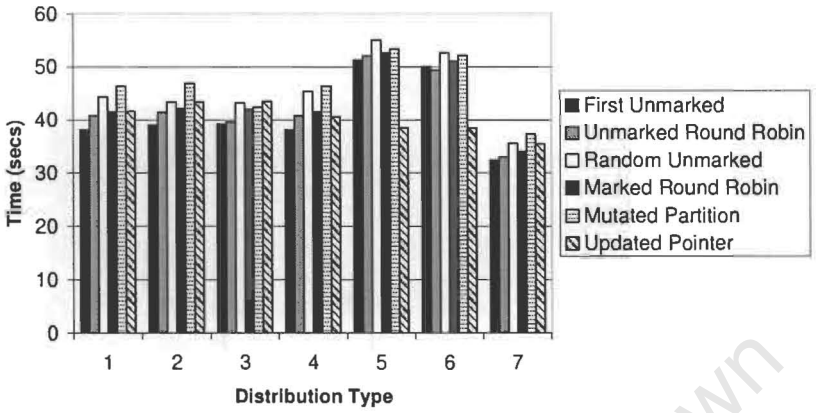


Figure 5.26: The effect of unevenly distributed inter-partition references on garbage collection based on partition selection policies for a 16 MB store.

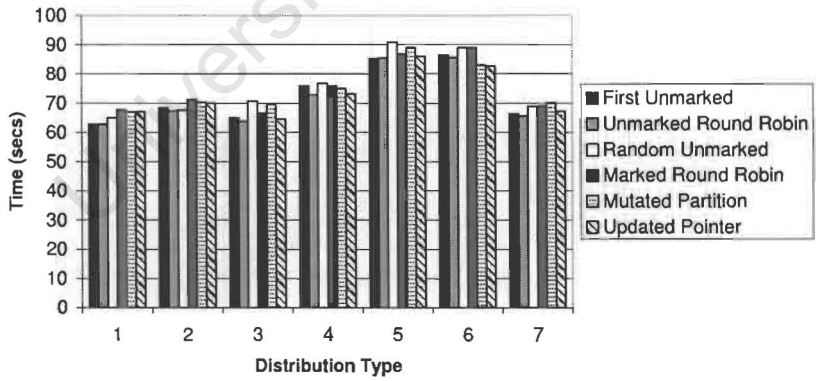


Figure 5.27: The effect of unevenly distributed inter-partition cyclic garbage on garbage collection based on partition selection policies for a 16 MB store.

When comparing the partition selection policies for each distribution, we notice that the results conform to those for the evenly distributed case.

5.3.8 Page Allocation Configurations

Finally, we evaluate the affect of our page allocation schemes, described in Section 5.1.1, used to adjust the clustering of objects on the partitioned store. The test application was used to exhaust 16 MB stores with linked lists using the different page allocation techniques. The store was then altered to contain 30 percent garbage and 20 percent inter-partition references. The stores were then garbage collected and the results are shown in Figure 5.28. Note that translist implementation A was used in this case.

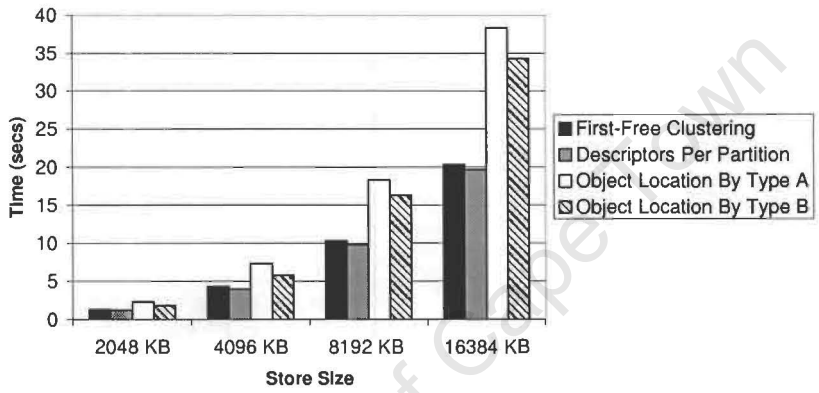


Figure 5.28: The effect of page allocation configurations on partitioned garbage collection for varying store sizes.

The results show that first-free and descriptors-per-partition page allocation schemes (refer to Section 5.1.1 for descriptions of page allocation schemes) require similar times. When descriptors are allocated to each partition, the garbage collection time is slightly less. This is because the number of inter-partition references is decreased by this allocation scheme, thus requiring less garbage collection time.

When objects are allocated by type the garbage collection times increase significantly. This is because when objects are allocated according to their type, the linked lists span numerous partitions. We saw earlier in Section 5.3.3 that large amounts of inter-partition references cause a bottleneck in garbage collection. Thus the results are almost double compared to first-free and descriptor-per-partition page allocation techniques.

5.4 Summary

We have evaluated the performance of semispace copying and Maheshwari's partitioned garbage collection scheme on PLaVa stores under different configurations. The evaluation results are summarized here :

Semispace Copying

The results for semispace copying are much as would be expected: garbage collection overhead is proportional to the amount of live data. Unlike with the partitioned garbage collector, this applies to all store sizes, since there is no other overhead apart from the handling of live data. Depth-first copying requires less time than breadth-first copying in all cases, because a depth-first object promotion scheme is used in the virtual machine and because this requires no queue maintenance by the collector.

Intra-partition Garbage with Partitioned Collection

The effect of evenly distributed intra-partition garbage on partitioned garbage collection overhead behaves in a similar fashion to semispace copying: collection overhead decreases linearly as the amount of garbage increases. However, garbage collection time increases with an increase in store size, since larger stores enforce a larger relative overhead to collect, and little overhead is required to maintain translists since very few inter-partition references exist.

The effect of different distributions of intra-partition garbage on garbage collection is small, but shows that high concentrations of garbage are more costly to collect than balanced distributions.

Inter-partition References with Partitioned Collection

As the number of inter-partition references increase, so the collection overhead increases exponentially. We see that the time allocated to translist management causes the exponential characteristic in the results, since all other parameters remain relatively constant. The 2, 4, and 8 MB stores seem to handle large degrees of inter-partition references, but the 16 MB store requires a relatively large collection overhead.

Again, high concentrations of inter-partition references incur the greatest garbage collection costs requiring more translist management. Other distributions of inter-partition references require similar time to garbage collect.

Inter-partition Cyclic Garbage with Partitioned Collection

With an increase in the number of inter-partition garbage cycles, the number of inter-partition references increases. Therefore for an even distribution our results are expected to have a similar exponential characteristic as the results for inter-partition references. However, the garbage collection times are approxiamately double, since two complete marking phases are required to fully collect the store. Translist management time becomes prominant, since the store is required to maintain the inter-partition reference state of all cycles, since they are assumed to be live in the first phase.

Once again, highly concentrated inter-partition garbage cycles incur longer garbage collection times, while other distributions have similar times.

Page and Partition Size with Partitioned Collection

We altered the number of pages and the page size to maintain the partition size. The results show that fewer, larger pages require more collection overhead than more smaller pages. Most parameters remain relatively constant, except store I/O increased as the number of pages increases and the page size decreases.

We then altered the number of pages per partition and the number of partitions to maintain the same store size. As the number of partitions increases, so the greater the collection time becomes, since more translists are required to maintain the inter-partition state of the store. The more partitions a store store is divided into, the more of a bottleneck the partitioned collection scheme becomes.

Translist Implementations

We implemented two translist store organizations; TOA catered for the maximum number of possible inter-partition references into a partition, and TOB allocated less space to a partition's translist but dealt with overflow by using a translist overflow partition.

In our evaluation for intra-partition garbage, the translist organization did not effect the results, i.e. all measurements were similar. This is because the store did not contain inter-partition references. However, with the other evaluation results, the usual case was that the results remained similar until about the 50 percent mark, then an increase in store I/O would appear for TOB. This is because TOB requires an overflow partition, which requires more store I/O to load/write translists.

TOB is better when the distribution percentages are under 50 percent, while TOA, even though store space is wasted, out performs TOA.

Partition Selection Policies

Our results indicate that the different partition selection policies effect the costs involved in garbage collection. In the evenly distributed case, the updated pointer policy performs the worst followed by the random mutated partition selection policy. The unmarked scheme performs the best, whereas the unmarked round robin and marked round robin lie in the middle.

When applying different selection policies when garbage collecting unevenly distributed intra-partition garbage, the policies perform in a similar fashion across the distribution types, since we are garbage collecting intra-partition garbage. When comparing partition selection policies, the ratio of the differences seem to correspond with our results in the evenly distributed case.

When garbage collecting unevenly distributed inter-partition references using the defined selection policies, distributions 5 and 6 require the longest collection time. This is because of the bulk nature of the distribution types, i.e. partitions completely exhausted with inter-partition references incurrs a greater collection cost. The proportions of the actual partition policy times is again similar to the results in the unevenly distributed case.

Finally we evaluated the effect of unevenly distributing inter-partition cyclic garbage using the defined selection polices. The results show that distributions 5 and 6 require the longest collection time, because of their bulk nature. Again distribution 7 seems to require the least collection time, showing that even distributions incur less collection cost. Distribution 4 seems to require slightly more garbage collection time than distributions 1 - 3.

Page Allocation Configurations

We compared three page allocation configurations and the results show that when descriptors are allocated to each partition, less garbage collection cost is involved. This is because this scheme naturally eliminates the inter-partition references caused between an object and its reference. When allocating objects by type, i.e. objects of the same type are promoted to the same partition, the garbage collection time increases significantly. This is because these page allocation scheme

generate inter-partition references, since linked lists will span every partition at least once. The lowest garbage collection costs were incurred when we increased the number of objects of the same type that reference one another. This is because we are effectively decreasing the amount of inter-partition references.

Chapter 6

Conclusion

6.1 Summary

This thesis has described the design, implementation and evaluation of two garbage collection schemes for the store used by UCT's persistent Java Virtual Machine called PLaVa. Semispace copying was implemented in an attempt to provide the PLaVa store with a simple garbage collection functionality. Maheshwari and Liskov's partitioned garbage collection scheme was implemented to facilitate incremental garbage collection of the PLaVa store.

Semispace copying time is proportional to the amount of live data, and store size has very little impact on collection time when there is a large percentage of garbage. However, double the amount of secondary storage is required for such a mechanism. Depth-first copying performs better that breadth-first copying because is corresponds to the depth-first nature of the stabilizing procedure and requires no queue maintenence for storing branch points within the object graph.

Maheshwari's partitioned garbage collection scheme incurs greater garbage collection costs than semispace copying since the inter-partition reference state of the store needs to be maintained, in order to successfully and completely garbage collect the partitioned store. The evaluation results show that intra-partition garbage has little effect on garbage collection. However, large amounts of inter-partition references incur greater collection costs, an effect which is particularly evident when they are highly concentrated in pockets on the store. Even with collection of inter-partition garbage cycles, performance is mainly affected by the number of inter-partition references and is much less sensitive to the amount of garbage in these cycles. As a result, any store or paging configuration that affects the amount of inter-partition references has the greatest effect on garbage collection overhead.

Our evaluation results indicate that translist organizations affect garbage collection when significant degrees of inter-partition references are present on the store. Translist organzition B (TOB) performs well when low amounts of inter-partition references exist on the store, since less space is allocated to translists. However, when an overflow partition is required due to translist overflow, garbage collection time using TOB increases due to the I/O involved in maintaining the inter-partition state of the overflow partition. Depsite requiring more space, TOA performs better when large amounts of inter-partition references are present, since all relevant translists are clustered together and no additional I/O is required for an overflow partition.

The structure of the store, i.e. the page size, the number of pages per partition, and the number of partitions, affects garbage collection since the structure determines the amount of inter-partition references.

Our evaluation results indicate that partition selection policies do affect the time needed to complete garbage collection. This is mainly due to the impact of partition selection policy on the time taken to complete the detection of inter-partition cyclic garbage. The global marking scheme propagates marks through all partitions, and so by choosing different partitions in different sequences we affect the propagation of global marking. It is important to define relevant partition selection policies for each individual partitioned garbage collector, which takes into account the subtleties of the particular partitioned garbage collection technique. For our implementation of Maheshwari and Liskov's collector [ML97], our test runs indicate that round robin is more effective than an updated pointer policy, which a mutated poiter policy performing worse than both of these.

The different placement schemes we implemented affect garbage collection since they determine the amount of inter-partition references on the store. Allocating descriptors per partition (rather than having a single descriptor per class for the entire store) was shown to be a simple yet efficient technique for reducing inter-partition references, thus speeding up garbage collection. The page allocation scheme which determines an object's location by its type was shown to noticeably degrade garbage collection performance, since it increases the amount of inter-partition references on the store.

6.2 Future Work

This research has resulted in the successful implementation of two garbage collection schemes for the PLaVa store. Future work includes the following :

Garbage Collection of the Store Handle Table

We made use of a store handle table in the implementation of semispace copying to map object PIDs to their locations on the store. Therefore when an object was copied from one semispace to the other, only the handle table entry corresponding to the object requires updating. However, when objects become garbage due to reference modifications or deletions, their handle table entries are not reused. As a result the handle table may become large and may contain large degrees of redundant information.

To solve this problem we require a "garbage collection" mechanism for the handle table, i.e. a scheme of detecting handle table entries that are no longer used, so that they may be reused by other objects. Recall that objects PIDs are allocated as negative integers starting at -1, decreasing by a decrement of 1, and that it is these PIDs that are used as keys into the handle table. One solution would be to use a hash table. Each entry contains the location of a particular object on the store, along with its PID and a mark bit. During a copying collection, each object that is copied, i.e. each object that is live, has its hash table entry marked. This is achieved with little effort since an object's hash table entry requires access during the copying traversal anyway. Once the collection is complete, we scan the hash table, deleting all entries that are left unmarked.

Inter-Partition Reference Considerations

Since large amounts of inter-partition references cause greater garbage collection overhead, schemes that minimize such inter-partition references would make collection more efficient. One such technique is to allocate descriptors per partition, thus descreasing the amount of inter-partition references, since each object references its descriptor in the same partition. Another method could be to adapt the stabilizing algorithm in an attempt to minimize inter-partition references during stablizing.

Other techniques include altering the store configuration to suit particular applications. For example fewer larger partitions would naturally result in fewer inter-partition references on the store, however larger partitions would take longer to garbage collect. Mechanisms for monitoring inter-partition references and for tuning the store configuration would be useful in this context.

Studying Actual Applications

Other partition selection policies could be investigated, and the effect of different selection policies on real stores (being used in real-world applications) should be measured. The effect of other garbage collection and store parameters, such as partition size, clustering mechanisms, etc. should also be measured in the context of real application usage. Measurements of the amount and locality of garbage, inter-partition references and inter-partition garbage cycles in the stores used by real applications are also needed.

6.3 Closing Comments

Prior to this research, I had only made use of the Java programming language and had little experience in using persistent programming languages. I also had a vague idea of the theory of garbage collectors and how they are used and implemented, especially in persistent environments. During the implementation of the garbage collectors for the PLaVa store I gained experience and knowledge of the architecture and operation of Java Virtual Machines and of persistent stores.

Upon the completion of this research I have realized the importance of persistence to the vast expanding field of computer science, not to mention the importance of space reclamation in persistent environments. Research into garbage collection should be given more attention, as space reclamation is always a requirement, even though in some cases and in some environments the cost of garbage collection may be quite large.

In conclusion, I would like to thank all those people that made the completion of this research possible.

Bibliography

[ABC+83] M. P. Atkinson, P. J. Bailey, K. J. Chrisholm, P. W. Cockshott, and R. Morrison. An Approach to Persistence Programming. *The Computer Journal*, 26(4):360–365, 1983.

[ACCM83] M.P. Atkinson, K. J. Chrisholm, W. P. Cockshott, and R. M. Marshall. Algorithms for a Persistent Heap. *IEEE Sofware, Practice and Engineering*, 13(3):259–272, March 1983.

[AFFS95] L. Amsaleg, P. Ferreira, M. Franklin, and M. Shapiro. Evaluating Garbage Collectors for Large Persistent Stores. In *OOPSLA '95 Workshop on Object Database Behavior, Benchmarks, and Performance*, Austin, Texas, October 1995.

[AFG95] L. Amsaleg, M. Franklin, and O. Grüber. Efficient Incremental Garbage Collection for Client-server Object Database Systems. In *Proceedings of the 21th VLDB International Conference*, Zürich, Switzterland, September 1995.

[AJDS96] M. P. Atkinson, M. J. Jordan, L. Daynès, and S. Spence. Design Issues for Persistent Java: a type-safe , object-oriented, orthogonally persistent system. In *Proceedings of the 7th Workshop on Persistent Object Systems*, Cape May, May 1996.

[AM91] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In *3rd International Symposium on Programming Language Implementation and Logic Programming*, number 528 in *Lecture Notes in Computer Science*, pages 1–13, Passau, Germany, August 1991.

[AM95] M. P. Atkinson and R. Morrison. Orthogonal Persistent Object Systems. *VLDB Journal*, 4(3):319–401, 1995.

[AP87] S. Abraham and J. Patel. Parallel Garbage Collection on a Virtual Memory System. In E. Chiricozzi and A. D'Amato, editors, *International Conference on Parallel Processing and Applications*, pages 243–6, L'Aquila, Italy, September 1987. Elsevier, North-Holland.

[App89] Andrew A. Appel. Simple Generational Garbage Collection and Fast Allocation. *Software Practice and Experience*, 19(2):171–183, 1989.

[App92] Andrew W. Appel. *Compiling with Continuations*, chapter 16, pages 205–214. Cambridge University Press, 1992.

[Bak78] Henry G. Baker. List Processing in Real-time on a Serial Computer. *Communications of the ACM*, 21(4):280–294, 1978. Also published as AI Laboratory Working Paper 139, 1977.

[Bak92] Henry G. Baker. The Treadmill, Real-time Garbage Collection Without Motion Sickness. *ACM SIGPLAN Notices*, 27(3), March 1992.

[Bar90] Joel F. Bartlett. A Generational, Compacting Collector for C++. In *OOPSLA/ECOOP '90 Workshop on Garbage Collection in Object-Oriented Systems*, Ottowa, October 1990.

[BB74] E. C. Berkely and Daniel G. Bobrow, editors. *The Programming Language LISP: Its Operation and Applications.* Information International Inc., Cambridge, MA, fourth edition, 1974.

[BCW85] Brenda Baker, E. G. Coffman, and D. E. Willard. Algorithms for Resolving Conflicts in Dynamic Storage Allocation. *Journal of the ACM*, 32(2):327–343, April 1985.

[Bis77] P.B. Bishop. Computer systems with a very large address space and garbage collection. Technical Report MIT-LCS-TR-178, MIT Lab for Computer Science, May 1977.

[Bis98] Judy Bishop. *Java Gently.* Addison-Wesley, second edition, 1998.

[Boe93] Hans-Juergen Boehm. Space Efficient Conservative Garbage Collection. In *Proceedings of SIGPLAN '93 Conference on Programming Languages Design and Implementation*, pages 197–206, Albuquerque, NM, June 1993. ACM Press.

[Bro84] Rodney A. Brooks. Trading Data Space for Reduced Time and Code Space in Real-Time Garbage Collection on Stock Hardware. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pages 256–262, Austin, Texas, August 1984.

[Bro89] A. L. Brown. *Persistent Object Stores.* PhD thesis, University of St Andrews, Scotland, October 1989.

[But87] Magaret H. Butler. Storage Reclamation in Object Oriented Database Systems. In *Proceedings of the 1987 ACM SIGMOD International Conference on the Management of Data*, pages 410–425, San Fransisco, May 1987.

[BW88] Hans-Juergen Boehm and Mark Weiser. Garbage Collection in an Uncooperative Environment. *Software Practice and Experience*, 18(9):807–820, 1988.

[BZ93] David A. Barett and Benjamin G. Zorn. Using Lifetime Predictors to Improve Memory Allocation Performance. *Proceedings of SIGPLAN'94 Conference on Programming Languages Design and Implementation*, volume 28(6) of *ACM SIGPLAN Notices*, Albuquerque, NM, June 1993. ACM Press.

[CDG+90] M. J. Carey, D. J. DeWitt, G. Graefe, D. M. Haight, J. E. Richardson, D. T. Schuh, E. J. Shekita, and S. L. Vandenberg. The Exodus Extensible DBMS Project: An Overview. *Readings in Object-Oriented Database Systems*, pages 474–499, San Mateo, California, 1990. Morgan Kaufmann.

[CG77] Douglas W. Clark and C. Cordell Green. An Empirical Study of List Structure in Lisp. *Communications of the ACM*, 20(2):78–87, February 1977.

[Che70] C. J. Cheney. A Nonrecursive List Compacting Algorithm. *Communications of the ACM*, 13(11):677–8, November 1970.

[CK87] T. Chikayama and Y. Kimura. Multiple Reference Management in Flat GHC. In 4^{th} *International Conference on Logic Programming*, pages 276–293, University of Melbourne, May 1987. MIT Press.

[CKWZ96] Jonathan E. Cook, Artur W. Klauser, Alexander L. Wolf, and Benjamin G. Zorn. Semi-automatic, Self-adaptive Control of Garbage Collection Rates in Object Databases. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, Montreal, Canada, June 1996.

[Cla79] Douglas W. Clark. Measurements of Dynamic List Structure Use in Lisp. *IEEE Transactions on Software Engineering*, 5(1):51–9, January 1979.

[Col60] C. E. Collins. A Method for Overlapping and Erasure of Lists. *Communications of the ACM*, 3(12):655–7, 1960.

[Col89] R. M. Coloumb. Issues in the Implementation of Persistent Prolog. In *Proceedings of the 3^{rd} International Workshop on Persistent Object Stores*, pages 67–79, Newcastle, N.S.W., January 1989.

[CWZ94] Jonathan E. Cook, Alexander L. Wolf, and Benjamin G. Zorn. Partition Selection Policies in Object Database Garbage Collection. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, Minneapolis, MN, May 25 - 27 1994.

[Dav00] David S. Munro and Alfred L. Brown. Evaluating Partition Selection Policies using the PMOS Garbage Collector. In *9^{th} International Workshop on Persistent Object Systems*, Norway, September 2000.

[DB76] L. Peter Deutsch and Daniel G. Bobrow. An Efficient Incremental Automatic Garbage Collector. *Communications of the ACM*, 19(9):522–6, September 1976.

[DeT90] John DeTreville. Experience with Concurrent Garbage Collectors for Modula-2+. Technical Report 64, Digital Equipment Corporation Systems Research Center, Palo Alto, California, August 1990.

[Det91] David L. Detlefs. *Concurrent, Atomic Garbage Collection.* PhD thesis, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 15213, November 1991.

[DLM+78] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly Garbage Collection : An Exercise in Cooperation. *Communications of the ACM*, 21(11):966–975, November 1978.

[FCW89] M. J. Franklin, G. Copeland, and G. Weikum. What's Different About Garbage Collection for Persistent Programming Languages? Technical Report ACA-ST-062-89, MCC, Austin, Texas, February 1989.

[FH92] Mary F. Fernandez and David R. Hanson. Garbage Collection Alternatives for Icon. *Software Practice and Experience*, 22(8):659–672, August 1992.

[Fre99] Alan Freedman. *The Computer Desktop Encyclopedia.* Amacom, second edition, 1999.

[FS95] Paulo Ferreira and Marc Shapiro. Garbage Collection in the Larchant Persistent Distributed Shared Store. In *Proceedings of the 5^{th} Workshop on Future Trends in Distributed Computing Systems*, Cheju Island, Republic of Korea, August 28-30 1995.

[FS96] Paulo Ferreira and Marc Shapiro. Larchant: Persistence by Reachability in Distributed Shared Memory through Garbage Collection. In *Proceedings of the 16th International Conference on Distributed Computer Systems*, Hong Kong, 1996.

[FY69] Robert R. Fenichel and Jerome C. Yochelson. A Lisp Garbage-collector for Virtual-memory Computer Systems. *Communications of the ACM*, 12(11):611–2, November 1969.

[GR83] Adele Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan-Kaufman, San Mateo, CA, 1993.

[Hag87] R. Hagmann. Reimplementing the Cedar File System Using Group Commit. In *Proceedings of 11th Symposium on Operating System Principles*, pages 155–162, Austin, Texas, November 1987.

[Hay91] Barry Hayes. Using Key Object Opportunism to Collect Old Objects. In *ACM SIGPLAN 1991 Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA '91)*, pages 33–46, Phoenix, Arizona, October 1991. ACM Press.

[HD90] Richard L. Hudson and Amer Diwan. Adaptive Garbage Collection for Modula-3 and Smalltalk. In *OOPSLA '90 Workshop on Garbage Collection*, pages 273–282, Phoenix, Arizona, 1990.

[HE74] Timothy P. Hart and Thomas G. Evans. Notes on Implementing LISP for the M-460 Computer. In Berkely and Bobrow [BB74], pages 191-203, 1974.

[HM92] Richard L. Hudson and J. Eliot B. Moss. Incremental Garbage Collection for Mature Objects. In *Proceedings of the International Workshop on Memory Management*, volume 637 of *Lecture Notes in Computer Science*, St Malo, France, 16-18 September 1992. Springer-Verlag.

[HMMM97] Richard L. Hudson, Ron Morrison, J. Eliot B. Moss, and David S. Munro. Garbage Collecting the World: One Car at a Time. In *Proceddings of the 1997 Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '97)*, volume 32, pages 162–175. ACM Press, 1997.

[Hug85] R. J. M. Hughes. A Distributed Garbage Collection Algorithm. In J. P. Jouannoud, editor, *Functional Programming Language and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 256–272. Springer-Verlag, 1985.

[HW67] B. K. Haddon and W. M. Waite. A Compaction Procedure for Variable Length Storage Elements. *Computer Journal*, 10:162–5, August 1967.

[Jia92] B. Jiang. DFS-traversing graphs in a paging environment, LRU or MRU? *Information Processing Letters*, 40(4):193–196, 1992.

[JL99] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.* John Wiley & Sons, 1999.

[KCP] kissme-classpath - A package of the GNU Classpath Java class libraries for the kissme JVM. John Leuner.
http://freshmeat.net/users/jewel/ [03/10/2001].

[Kis] kissme - Free Java Virtual Machine. John Leuner.
http://freshmeat.net/users/jewel/ [03/10/2001].

[KLW89] Elliot K. Kolodner, Barbara Liskov, and William E. Weihl. Atomic Garbage Collection: Managing a Stable Heap. In *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data*, pages 15–25, Portland, Oregon, June 1989.

[Knu73] D. E. Knuth. *The Art of Computer Programming*, volume 1: Fundamental Algorithms. Addison-Wesley, 1973.

[Kol92] Elliot K. Kolodner. Atomic Incremental Garbage Collection and Recovery for a Large Stable Heap. Technical Report MIT-LCS-TR-534, Massachusetts Institute of Technology, February 1992.

[LAC+96] B. Liskov, A. Adya, M. Castro, M. Day, S. Ghemawat, R. Gruber, U. Maheshwari, A. Myers, and L. Shrira. Safe and Efficient Sharing of Persistent Objects in Thor. In *Proceedings of '96 SIGMOD*, pages 318–329. ACM Press, 1996.

[Ler93] Steven R. Lerman. *Problem Solving And Computation For Scientists And Engineers: An Introduction Using C.* Prentice-Hall, 1993.

[LFP84] *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, Austin, Texas, August 1984. ACM Press.

[LH83] Henry Lieberman and Carl Hewitt. A Real-time Garbage Collector based on the Lifetimes of Objects. *Communications of the ACM*, 26(6):419–429, June 1983.

[LQP92] Bernard Lang, Christian Queinnec, and José Piquer. Garbage Collecting the World. In *Proceedings of Principles of Programming Languages (POPL) '92*, pages 39–50. ACM Press, 1992.

[LY96] Tim Lindholm and Frank Yellin. *The Java™ Virtual Machine Specification.* Addison-Wesley, 1996.

[MA90] R. Morrison and M. P. Atkinson. Persistent Lanaguages and Architectures. In J. Rosenberg and J. L. Keedy, editors, *Security and Persistence*, pages 9–28. Springer-Verlag, 1990.

[MABD88] R. Morrison, M. P. Atkinson, A. L. Brown, and A. Dearle. Bindings in persistent programming languages. *SIGPLAN Notices*, 23(4):27–34, April 1988.

[Mah97] Umesh Maheshwari. Garbage Collection in a Large, Distributed Object Store. Technical Report MIT-LCS-TR-727, MIT Laboratory for Computer Science, Cambridge, Massachusetts, September 1997.

[MBCD89] R. Morrison, A. L. Brown, R. Connor, and A. Dearle. *The Napier88 Reference Manual.* The University of Glasgow and St Andrews PPRR-77, Scotland, 1989.

[MBMM98] David S. Munro, Alfred L. Brown, Ron Morrison, and J. Eliot B. Moss. Incremental Garbage Collection of a Persistent Object Store using PMOS. In *Proceedings of the 8th International Workshop on Persistent Object Systems*, Tiburon, California, 30 August - 1 September 1998.

[McB63] J. Harold McBeth. On the Reference Counter Method. *Communications of the ACM*, 6(9):575, September 1963.

[McC60] John McCarthy. Recursive Functions of Symbolic Expressions and their Computation by Machine. *Communications of the ACM*, 3(4):184–195, April 1960.

[Min63] Marvin L. Minsky. A Lisp Garbage Collector Algorithm using Serial Secondary Storage. Technical Report Memo 58 (rev.), Project MAC, MIT, Cambridge, MA, December 1963.

[ML97] Umesh Maheshwari and Barbara Liskov. Partitioned Garbage Collection of a Large Object Store. In *ACM SIGMOD International Conference on Management of Data*, 1997.

[MMH96] J. Eliot B. Moss, D. S. Munro, and R. L. Hudson. PMOS: A Complete and Course-Grained Incremental Garbage Collector for Persistent Object Stores. In *Proceedings of the 7th Workshop on Persistent Object Systems*, pages 140–150. Morgan Kaufmann, 1996.

[Moo84] David Moon. Garbage Collection in a Large Lisp System. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 235–246, Austin, Texas, August 1984. ACM Press.

[MS89] F. Matthes and J. W. Schmidt. The Type System of DBPL. In *Proceedings of the 2nd International Workshop on Database Programming Languages*, pages 219–225, Salishan, Oregon, June 1989.

[MTH89] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1989.

[NS97] Peter Naughton and Herbert Schildt. *Java: The Complete Reference*. Osborne McGraw-Hill, 1997.

[ONG93] James W. O'Toole, Scott M. Nettles, and David Gifford. Concurrent Compacting Garbage Collection of a Persistent Heap. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, Asheville, North Carolina, December 1993.

[OPJ] OPJGC - The PJama Disk Garbage Collector. http://www.dcs.gla.ac.uk/pjava/tutorial/opj.tutorial.opjgc.html [18/01/2001].

[PAD+97] Tony Printezis, Malcolm Atkinson, Laurent Daynès, Susan Spence, and Pete Bailey. The Design of a new Persistent Object Store for PJama. In *Second International Workshop on Persistent and Java*, San Francisco Bay Area, California, August 1997.

[Pri00] Tony Printezis. *Management of Long-Running High-Performance Persistent Object Stores*. PhD thesis, University of Glasgow, May 2000.

[RCS90] J. Richardson, M. Carey, and D. Schuh. The Design of the E Programming Language. Technical Report 824, Computer Science Department, University of Wisconsin, February 1990.

[Rov85] Paul Rovner. On Adding Garbage Collection and Runtime Types to a Strongly-typed, Statically-checked, Concurrent Language. Technical Report CSL-84-7, Xerox PARC, Paulo Alto, California, July 1985.

[Sau74] Robert A. Saunders. The LISP System for the Q-32 Computer. In Berkely and Bobrow [BB74], pages 220-231, 1974.

[Sch95a] Herbert Schildt. *C++: The Complete Reference.* Osborne McGraw-Hill, second edition, 1995.

[Sch95b] H. Schwetman. Object-oriented Simulation Modeling with C++/CSIM. In *Proceedings of 1995 Winter Simulation Conference*, pages 529–533, December 1995.

[SCN84] Will R. Stoye, T. J. W. Clarke, and Arthur C. Norman. Some Practical Methods for Rapid Combinator Reduction. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 159–166, Austin, Texas, August 1984. ACM Press.

[Sha98] Robert A. Shaw. *Empirical Analysis of a Lisp System.* PhD thesis, Stanford University, 1998. Technical Report CSL-TR-88-351.

[Sob88] Patrick G. Sobalvarro. A Lifetime-based Garbage Collector for Lisp Systems on General-purpose Computers. Technical Report AITR-1417, MIT AI Lab, February 1988. Bachelor of Science thesis.

[SV97] Marcin Skubiszewski and Patrick Valduriez. Concurrent Garbage Collection in O_2. In *Proceedings of the 23rd VLDB Conference*, Athens, Greece, 1997.

[TB98] Stephan J. Tjasink and S. Berman. Providing Persistence on Small Machines. In R. Morrison and M. Atkinson, editors, *Eighth International Workshop on Persistent Object Systems*, Tiburon, CA, U.S.A., August 30 - September 1 1998.

[Tja99] Stephen Tjasink. PLaVa: A Persistent, Lightweight Java™ Virtual Machine. Master's thesis, University of Cape Town, February 1999.

[UJ88] David M. Ungar and Frank Jackson. Tenuring Policies for Generation-based Storage Reclamation. *CM SIGPLAN Notices*, 23(11):1–17, 1988.

[Ung84] David M. Ungar. Generation Scavenging : A Non-disruptive High Performance Storage Reclaimation Algorithm. *ACM SIGPLAN Notices*, 19(5):157–167, April 1984.

[Wei63] J. Weizenbaum. Symmetric List Processor. *Communications of the ACM*, 6(9):524–544, September 1963.

[Wei94] Mark Allen Weiss. *Data Structures and Algorithm Analysis in C++.* The Benjamin / Cummings Publishing Company, Inc., 1994.

[WF77] David S. Wise and Daniel P. Friedman. The One-bit Reference Count. *BIT*, 17(3):351–9, 1977.

[Whi80] Jon L. White. Address/Memory Management for a Gigantic Lisp Environment. In *Conference Record of the 1980 Lisp Conference*, pages 119–127, Redwood Estates, CA, August 1980. Also published under the title "GC Considered Harmful".

[Wil92] Paul R. Wilson. Uniprocessor Garbage Collection Techniques. In *Proceedings of the 1992 International Workshop on Memory Management*, pages 1–42. ACM Springer-Verlag, September 1992.

[Wil94] Paul R. Wilson. Uniprocessor Garbage Collection Techniques. Technical report, University of Texas, January 1994. Revised version of [Wil92].

[Wis79] David S. Wise. Morris' Garbage Compaction Algorithm Restores Reference Counts. *ACM Transactions on Programming Languages and Systems*, 1:115–120, July 1979.

[Wis93] David S. Wise. Stop and One-bit Reference Counting. Technical Report 360, Indiana University, Computer Science Department, March 1993.

[WJ93] Paul R. Wilson and Mark S. Johnstone. Truly real-time non-copying garbage collection. In *OOPSLA/ECOOP '93 Workshop on Garbage Collection in Object-Oriented Systems*, October 1993.

[WM89] Paul R. Wilson and Thomas G. Moher. Design of the Opportunistic Garbage Collector. *ACM SIGPLAN Notices*, 24(10):23–35, 1989.

[YNY97] V. Yong, J. Naughton, and J. Yu. Storage Reclamation and Reorganization in Client-Server Persistent Object Stores. In *Proceedings of Data Engineering*, pages 120–133. IEEE Press, 1997.

[Yua90] Taichi Yuasa. Real-time Garbage Collection on General-Purpose Machines. *Journal of Software and Systems*, 11(3):181–198, 1990.

[Zor89] Benjamin G. Zorn. *Comparative Performance Evaluation of Garbage Colleciton Algorithms.* PhD thesis, University of California at Berkeley, March 1989. Technical Report UCB/CSD 89/544.

[Zor90] Benjamin Zorn. Comparing Mark-and-Sweep and Stop-and-Copy Garbage Collection. In *1990 ACM Conference on Lisp and Functional Programming*, pages 87–98, Nice, France, June 1990.