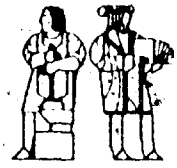


LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TR-534

ATOMIC INCREMENTAL GARBAGE COLLECTION AND RECOVERY FOR A LARGE STABLE HEAP

Elliot K. Kolodner

February 1992

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Atomic Incremental Garbage Collection and Recovery for a Large Stable Heap

by

Elliot K. Kolodner
S.M., Massachusetts Institute of Technology (1987)
B.A., B.S.E., University of Pennsylvania (1976)

Submitted to the
Department of Electrical Engineering and Computer Science
in Partial Fulfillment
of the Requirements for the Degree of

Doctor of Philosophy

at the

Massachusetts Institute of Technology

February, 1992

Signature of Author _____
Department of Electrical Engineering and Computer Science
January 10, 1992

Certified by _____
William Edward Weihl
Associate Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by _____
Campbell L. Searle
Chairman, Departmental Committee on Graduate Students

Atomic Incremental Garbage Collection and Recovery for a Large Stable Heap

by

Elliot K. Kolodner

Submitted to the Department of Electrical Engineering and Computer Science on January 10, 1992 in partial fulfillment of the requirements for the Degree of Doctor of Philosophy

Abstract

A *stable heap* is storage that is managed automatically using garbage collection, manipulated using atomic transactions, and accessed using a uniform storage model. These features enhance reliability and simplify programming by preventing errors due to explicit deallocation, by masking failures and concurrency using transactions, and by eliminating the distinction between accessing temporary storage and permanent storage. Stable heap management is useful for programming languages for reliable distributed computing, programming languages with persistent storage, and object-oriented database systems.

Many applications that could benefit from a stable heap (e.g., computer-aided design, computer-aided software engineering, and office information systems) require large amounts of storage, timely responses for transactions, and high availability. We present garbage collection and recovery algorithms for a stable heap implementation that meet these goals and are appropriate for stock hardware. The collector is incremental: it does not attempt to collect the whole heap at once. The collector is also atomic: it is coordinated with the recovery system to prevent problems when it moves and modifies objects. The time for recovery is independent of heap size, and can be shortened using checkpoints.

An object in a stable heap is stable and survives crashes if it is reachable from a root that is designated stable; other objects are volatile (e.g., objects local to procedure invocations) and do not need to survive crashes. By tracking objects as they become stable and dividing the heap into a stable area and a volatile area, our algorithms incur the costs of crash recovery and atomic garbage collection only for stable objects. Our tracking algorithm is concurrent; multiple transactions can invoke the tracking algorithm at the same time.

We present a formal specification for a stable heap and a formal description of the atomic incremental garbage collector and recovery system used to implement it. We demonstrate the correctness of the algorithms by exhibiting the abstraction function and the invariants they maintain. We also describe a prototype that we implemented to show the feasibility of our algorithms.

Thesis Supervisor: William Edward Weihl
Title: Associate Professor of Computer Science and Engineering

Keywords: garbage collection, transactions, recovery. persistence, object-oriented, database, distributed systems, formal specification

This research was supported in part by the National Science Foundation under grant CCR-8716884 and in part by the Defense Advanced Research Projects Agency (DARPA) under Contract N00014-89-J-1988.

Acknowledgments

I would like to thank my thesis advisor, Bill Weihl, for his guidance and encouragement as I worked on this thesis. I would also like to thank my readers, Butler Lampson and Barbara Liskov, for their suggestions that helped to improve the presentation.

Special thanks go to Dorothy Curtis and Paul Johnson for keeping the systems running and answering my Argus questions, Anthony Joseph and Ilhamuddin Ahmed for modifying the Argus linker to allow the linking of subroutines written in C, John Wroclawski for bringing up Mach, Wilson Hsieh for thoroughly proofreading the final draft, Eric Brewer for wading through the Spec code, Tom Cormen for drawing figures, and Lisa Kelly for proofreading these acknowledgments. I would also like to thank other members of the Principles of Computer Systems Group, past and present, who discussed various aspects of the research with me: Boaz Ben-Zvi, Adrian Colbrook, Mark Day, Sanjay Ghemawat, Bob Gruber, Debbie Hwang, Daniel Jackson, Anthony Joseph, Rivka Ladin, Sharon Perl, Mark Reinhold, Liuba Shrira, Mark Vandevoorde, and Earl Waldin.

I am grateful to my parents, Meyer and Sarah, Elana's parents, Adinah and Joe, my sisters Janet, Aviva, and Naomi, and my brothers Stuart, Mike, Dan and Gideon, for their love, support and encouragement.

Finally, I thank Elana, Kobi and Noam; without their love I could not have finished.

Contents

List of Figures

Chapter 1

Introduction

A *stable heap* is storage that is managed automatically using garbage collection, manipulated using atomic transactions, and accessed using a uniform storage model. Automatic storage management, used in modern programming languages, enhances reliability by preventing errors due to explicit deallocation (e.g., dangling references and storage leaks). Transactions, used in database and distributed systems, provide fault-tolerance by masking failures that occur while they are running. A uniform storage model simplifies programming by eliminating the distinction between accessing temporary storage and permanent storage. Stable heap management will make it easier to write reliable programs and could be useful in programming languages for reliable distributed computing [14, 29], programming languages with persistent storage [1, 3], and object-oriented database systems [11, 33, 52, 55].

In earlier research [24, 25] we designed algorithms suitable for the implementation of small stable heaps. However, many applications that could benefit from a stable heap (e.g., computer-aided design, computer-aided software engineering, and office information systems) require large amounts of storage, timely responses for transactions, and high availability. This dissertation presents algorithms to implement the large stable heaps necessary to support these applications.

A recovery system provides fault-tolerance for transactions; it manages information that ensures that the effects of successful transactions persist across failures and that unsuccessful transactions have no effect. A garbage collector typically moves and modifies objects as it collects storage that is no longer in use. It moves objects to improve locality of reference and reduce fragmentation; it modifies them in order to speed its work and reduce the amount of

additional storage it requires. In a stable heap the movement and modification of objects by the garbage collector may interfere with the work of the recovery system; yet, the recovery system must be able to recover the objects modified by the collector and find the moved ones when a failure occurs. The collector must also have access to the recovery information; objects may be reachable from this information that the collector would not otherwise retain. A collection algorithm that solves these problems and is coordinated correctly with the recovery system is called an *atomic garbage collector*.

In our earlier work we introduced this notion of atomic garbage collection and presented an algorithm for it. We based our earlier atomic collector on a stop-the-world copying collector; it suspends work on all transactions while it collects and traverses the stable heap. These pauses grow longer as the heap grows larger. Similarly, the recovery system used in our earlier work requires a traversal of the whole stable object graph after a crash. For an application with a large stable graph, this traversal delays recovery and reduces the availability of the application. The exact heap size at which the pauses for garbage collection or the time for recovery become intolerable depends on the application, its response time requirements, and its availability constraints, as well as on hardware characteristics such as processor speed.

The research reported here builds on our previous research: we present the design of an integrated atomic garbage collector and recovery system appropriate for a large stable heap on stock hardware. The collector is incremental; i.e., it does not attempt to collect the whole heap in one pause. The time for recovery is independent of heap size even if a failure occurs during garbage collection, and can be shortened using a checkpointing mechanism. The design is for stock hardware with virtual memory, since these are the most common computing machines.

Our approach treats an object as stable if it is reachable from a root that is designated stable; our algorithms for atomic garbage collection and recovery ensure that stable objects survive crashes. Other objects are volatile; they do not need to survive crashes or require the functionality provided by our algorithms. We show how to avoid recovery costs for volatile objects by tracking the dynamically changing set of stable objects. The tracking algorithm is concurrent; tracking for a transaction can proceed in parallel with other tracking, and

with other transactions in the system. Also, our tracking algorithm corrects a bug in a previously published algorithm [38].

We also show how to avoid the costs of atomic garbage collection for volatile objects by dividing the heap into stable and volatile areas. Storage management in the volatile area is provided cheaply by a normal garbage collector; the more expensive atomic garbage collector is used only in the stable area.

We have implemented a stable heap prototype to show the feasibility of our algorithms. The current implementation of Argus [31] serves as the basis for the prototype; we replaced its existing storage management and recovery algorithms.

There has been other work dealing with the problem of providing fault-tolerant heap storage, but none of the solutions has been satisfactory. Some work (e.g., [9, 46]) provides persistence but not transactions, so it offers less functionality. PS-algol [3] uses a stop-the-world garbage collector and does not permit garbage collection to occur while transactions are in progress. It also uses a less general transaction model and a recovery system that imposes a high run-time overhead. Earlier work on Argus [38] uses a normal garbage collector, but treats all crashes as media failures; as a result, recovery from a system failure is relatively slow, particularly for large heaps. Our work grew out of an attempt to design a faster recovery system for Argus.

Detlefs's work [15] is the closest to ours; he has published an algorithm he calls concurrent atomic garbage collection. In his algorithm the pauses for garbage collection and the time for recovery are independent of heap size, but the pauses are too long. Each pause requires multiple synchronous writes to disk; furthermore, these writes are random. Our algorithm is better integrated with the recovery system and does not require any synchronous writes to disk.

Here is an overview of the structure of this report. Chapter 2 presents our model of a stable heap and describes an approach to recovery called repeating history [34]. Using the approach, modifications to objects in the heap follow a write-ahead log protocol. The protocol ensures that the modifications are repeatable after a failure.

In Chapters 3 and 4 we assume that every accessible object in a stable heap must survive failure. Chapter 3 describes our algorithm for atomic incremental garbage collection. It

shows how to use the write-ahead log protocol to make the steps of an incremental copying collector repeatable. It also discusses the interactions between the collector and a recovery system. Chapter 4 describes a complete recovery system for a stable heap based on repeating history and shows how to integrate the atomic garbage collector into the recovery system.

Chapter 5 shows how to use the atomic garbage collector and recovery system for a heap that contains volatile state. It describes the concurrent tracking algorithm and the division of the heap into a stable area and a volatile area.

Chapter 6 presents a formal description of a stable heap, a formal description of our algorithms, the key invariants maintained by the algorithms, and an abstraction function that explains how the algorithms work.

Chapter 7 describes our prototype implementation. It explains the changes we made to the current Argus implementation to divide the heap into a stable area and a volatile area, to use the atomic garbage collector in the stable area, and to make the time for recovery independent of heap size.

Chapter 8 surveys related work.

Finally, Chapter 9 summarizes our work and suggests directions for future research.

Chapter 2

Stable Heaps

We abstracted our model of a stable heap from the model of computation used by Argus, a programming language for reliable distributed computing, for local computation at each node in a distributed system. The stable heap model is also appropriate for object-oriented database systems and other programming languages with persistent storage. We begin this chapter by describing the model. Then we discuss recovery and failures. Finally, we discuss the hardware and operating systems for which our design is appropriate.

2.1 System Model

In the model computations on shared state run as atomic transactions [19], and storage is organized as a heap. Transactions provide concurrency control and fault tolerance; they are *serializable* and *total*. Serializability means that when transactions are executed concurrently, the effect will be as if they were run sequentially in some order. Totality means that a transaction is all-or-nothing; i.e., either it completes entirely and *commits*, or it *aborts* and is guaranteed to have no effect.

A transaction consists of a series of short low-level recoverable actions: a *read* action reads a single object, an *update* action modifies a single object, and an *allocate* action creates a new object. These actions synchronize through logical mutual exclusion locks on objects. In practice these mutual exclusion locks may be of a coarser granularity. For example, in Argus most read and update actions are indivisible. Indivisibility is enforced by allowing context switches only at low-level action boundaries.

Objects shared among transactions must be *atomic*. Atomic objects provide the synchro-

nization and recovery mechanisms necessary to ensure that transactions are serializable and total. Atomic objects can be mutable or immutable. Immutable objects are always atomic because their values never change; examples of immutable types are integer, boolean, character, string and sequence. An implementation would also provide built-in mutable atomic types and type generators; some examples are array, record and variant. In this report we assume that the heap synchronizes access to these objects using standard read/write locking, and we describe appropriate recovery mechanisms. Using the built-in types, a programmer can build objects of user-defined atomic types [49] that exhibit greater concurrency than the built-in atomic types with the aid of lazy nested top-level transactions [22, 45] and multi-level concurrency control [51].

A heap consists of a set of root objects and all the objects accessible from them. Objects vary in size and may contain pointers to other objects. In a stable heap, some programmer specified roots are stable; the rest are volatile. The stable roots are global. The *stable state* is the part of the heap that must survive crashes; it consists of all objects accessible from the stable roots. The objects in the stable state must be atomic. The *volatile state* does not necessarily survive crashes; it consists of all objects that are accessible from the volatile roots, but are not part of the stable state, e.g., objects local to a procedure invocation, objects created by a transaction that has not yet completed, and global objects that do not have to survive crashes.

The programmer sees one heap containing both stable and volatile objects. He can store pointers to stable objects in volatile objects, and can cause volatile objects to become stable by storing pointers to them in an object that is already stable. (A volatile object actually becomes stable when a transaction that makes it accessible from a stable object commits.) Transactions share a single address space that contains both shared global objects and objects local to a single transaction; the programmer does not need to move objects between secondary storage and a transaction's local memory, or distinguish between local and global objects.

2.2 Recovery and Failure Model

Below we describe the storage architecture for a stable heap, the failure model, how to do recovery, and how to optimize recovery.

In this report we assume that transactions are not distributed. Our recovery algorithms can be extended to support distributed transactions with the addition of a two phase commit protocol. The changes necessary to support two phase commit should be obvious.

2.2.1 Storage Architecture

A recovery system provides fault-tolerance by controlling the movement of data between the levels of a storage hierarchy. In a typical database there are four components in the hierarchy: (1) main memory, (2) disk, (3) log, and (4) archive. We assume a similar hierarchy for the design of our algorithms.

A database keeps its data on disk, which is non-volatile, and uses main memory, which is volatile, as a cache or buffer pool. A buffer manager decides which pages to keep in the cache; it reads pages from disk into main memory and writes modified pages back to disk. The recovery system may constrain the buffer manager by *pinning* a page in main memory; a pinned page may not be written back to disk until recovery unpins it. For a stable heap, the main memory and disk together implement a one-level store or virtual memory.

The log is a sequential file, usually kept on a stable storage device, to which the recovery system writes information that it needs in order to redo the effects of a committed transaction or undo the effects of an aborted transaction. A stable storage device [26] is often implemented using a pair of disks; with very high probability, it avoids the loss of information due to failure. The recovery system does not directly write to the log on stable storage; rather, it spools information to a log buffer. When a buffer fills, recovery writes it to disk asynchronously and begins spooling to the next buffer. A well designed recovery system synchronously writes a buffer to stable storage, or *forces* the log, only at transaction commit when it must ensure that the effects of the transaction survive failure.[1] In this dissertation when we say *write to the log*, we mean spool to the log buffer. If we want to

[1] A high performance transaction system will use *group commit* [18] instead of forcing the log for every transaction; this allows the buffer to fill before writing it to stable storage, and commits many transactions at the same time.

describe a synchronous write, we use the phrase *force the log*. To distinguish the part of the log on stable storage from the part in the log buffer, we call the former the *stable log* and the latter the *volatile log*. When we use the word log without qualification, we mean the whole log, both its stable and volatile parts.

The archive is an out-of-date copy of the database; it may be on disk or some cheaper non-volatile medium such as magnetic tape.

2.2.2 Failure Model

A recovery system deals with three kinds of failure: (1) transaction, (2) system, and (3) media. A transaction fails when it aborts; the recovery system may use information in main memory, on the disk, or in the log to ensure that the transaction has no effect.

A system failure can be caused by software (e.g., inconsistent data structures in the operating system) or hardware (e.g., power failure). When the system fails main memory is lost, but the disk and log survive. A system failure also aborts transactions that are active when it occurs. The recovery system uses information in the log and on the disk to recover the state of the heap. The recovered heap reflects the cumulative effects of all the transactions that committed before the failure, and none of the effects of aborted transactions. We also call a system failure a crash.

A media failure occurs when a page or several pages of the disk get corrupted. The recovery system uses the log together with the archive to recover the pages. In this dissertation we are concerned primarily with transaction and system failure. Our recovery system writes enough information to the log to recover from a total media failure; but we do not discuss archives or how to optimize the recovery of a single page. Many of the standard techniques for constructing an archive and dealing with partial media failure are applicable to our work [21, 34].

2.2.3 Recovery

Given the storage architecture and failure model described above, we describe a way to do recovery called repeating history due to Mohan [34]. Repeating history is easy to understand and easy to optimize.

The key to repeating history is the *write-ahead log protocol*, which we also call the

redo protocol. For any modification to an object, the recovery system follows the ensuing protocol:

1. It pins the page on which the object resides in main memory. The buffer manager may not write the pinned page to disk.
2. It modifies the object on the page.
3. It spools a record containing redo information to the log buffer. The record contains the address at which the modification occurred and the new value.
4. At this point, the modification is complete and the protocol returns to its invoker.
5. After the redo record is in the stable log, the page is unpinned. The buffer manager is then free to write the unpinned page back to disk.

The write-ahead log protocol ensures the following property: if a modification is on disk, the redo record describing the modification is in the stable log. The *repeating history invariant* follows directly from this property:

Invariant 2.1 (Repeating History) *The disk state that would be produced by applying the stable part of the redo log to the disk (i.e., carrying out each of the redo actions in the order they are recorded in the log) is a state that actually occurred at some previous point in the computation.*[2]

The action of redoing the log is called *repeating history*.

The repeating history invariant simplifies recovery after a crash. The repeating history invariant is also useful for our atomic garbage collector. In the design of our collector, we depend on the invariant to bring the heap to a state from which the collector can complete its work after a crash.

We present an approach to abort for a system in which transactions update objects in place; the recovery system described in Chapter 4 uses this approach. (Our prototype implementation uses a different approach to abort, which we describe in Chapter 7.) To deal with abort, the recovery system includes undo information together with the redo information in the record it writes during the write-ahead log protocol. The undo information

[2]Formally, an invariant is a predicate on state; at first glance this statement of the repeating history invariant may not appear to be such a predicate. However, we can formalize it by defining the redo function determined by the log, and defining a history variable that captures the sequence of states through which the computation passes.

may be *logical*, the name of an operation and arguments to the operation, or *physical*, the previous state for the part of the object that is modified. Usually logical information takes up less space in the log than physical information. To abort a transaction, the recovery system undoes the transaction's updates in reverse order. Undoing an update is a modification so it follows the write-ahead log protocol and writes a redo record describing the undo. A redo record written by undo is called a *compensation log record* or CLR. There is no undo information in a CLR; undo never has to be undone.

Employing the above approach to abort, recovery after a system failure begins by repeating history, i.e., applying the redo information in the stable log to the disk. According to the invariant, this brings the database to a state from which it is valid to abort the transactions that were active before the crash.[3] Then recovery completes by using its normal method for transaction abort to abort the active transactions. This is much simpler than previous recovery algorithms that required more complicated redo and undo passes [28].

2.2.4 Optimizations

It is easy to optimize a recovery system based on repeating history, and to understand why the optimizations work. Logical undo is one such optimization. We briefly describe two other optimizations below; both shorten recovery times after a system failure. In Chapter 4 we describe how the optimizations fit into a full recovery system.

First, the buffer manager writes a *page-fetch record* to the log each time it fetches a page from disk into main memory, and an *end-write record* just after an updated page of main memory reaches disk. These records contain the number of the page that was read or written. Using these records, the recovery system can deduce a superset of the pages that were dirty at the time of a system failure. When it repeats history, it only has to apply redo records to the pages in this set.

Second, the recovery system checkpoints at regular intervals to keep the time for recovery short. To checkpoint, it stops the system in a low-level quiescent state, a state for which no thread is in the middle of the write-ahead log protocol (steps 1 – 4 of the protocol discussed

[3]Recovery is a bit more complicated for an update to an object that spans multiple pages. For such an update the write-ahead log protocol pins all updated pages until individual redo records describing the change to each page are in the stable log. After a crash, recovery applies the redo records for a multi-page update only if all of its records are in the stable log.

in Section 2.2.3). Then it constructs and writes a *checkpoint record* to the log. The record contains a list of the dirty pages at the time of the checkpoint and for each page the log address of its last page-fetch record. Using this information after a system failure, recovery deduces a point in the log from which it starts repeating history. The record also contains other information which we describe in Section 4.6. These checkpoints are cheap; they do not require any synchronous writes, and they halt the system for very brief periods.

2.3 Implementation Platform

Our design is for stock uniprocessors with virtual memory; these are the most common computing machines. No special-purpose hardware to support recovery or garbage collection is assumed.

The design requires an operating system that allows a program some control over the virtual memory system. Primitives are needed to control when a page of virtual memory can be written to the backing store and to set protections on pages. The ability to preserve the backing store for virtual memory after a crash is also required. Mach [39] satisfies these requirements and is used for the prototype implementation.

Chapter 3

Atomic Incremental Garbage Collection

Many algorithms for garbage collection have been described in the literature, but all are based on one of three basic types: (1) reference counting, (2) mark-sweep, or (3) copying. A reference counting collector associates a count with each object; the count contains the number of references to the object. The counts are updated as objects are modified. When an object's count reaches zero, its storage can be reclaimed. The major disadvantage of reference counting is that it cannot collect circular structures. A mark-sweep collector associates a mark bit with every object. To collect garbage, it begins by tracing from the roots of the heap and setting the mark bits of the objects reachable from the roots. Then it sweeps through memory and reclaims the storage of unmarked objects. A copying collector traces from the roots and copies the reachable objects to a new area of memory. Then it reclaims all of the storage in the old area.

Applying system requirements helps us choose an appropriate algorithm on which to base our atomic garbage collector. The principal requirement is that the algorithm be suitable for collecting a large heap. There are two implications: (1) the pauses associated with garbage collection must be short enough to support interactive response times, and (2) the collector must interact well with virtual memory.

A mark-sweep or copying collector that halts all other computations while it works is called a stop-the-world collector. Such a collector is not suitable for large heaps because the pauses associated with its collections are too long. Two general techniques have been used to shorten garbage collection pauses: (1) incremental garbage collection [4], and (2)

dividing the heap into independently collectible areas [6]. Steps of an incremental collector are interleaved with normal program steps such that the pause due to each incremental step is small. An incremental collector is also called real-time if there is a bound on the longest possible pause. To date most incremental collectors have been based on Baker's algorithm [4], which is a copying collector.

When dividing the heap into areas, either stop-the-world or incremental collection can be used in each area. A good division of the heap into areas leaves few inter-area references and places objects with similar lifetime characteristics into the same area. For programs without persistent storage, one automatic way of dividing the heap without programmer intervention is based on the age of objects; this is called generational collection [27, 35, 47]. Generational collection depends on an observed behavior of program heaps that new objects are more likely to become garbage than old objects; it concentrates its work on the areas containing the youngest objects, where the most storage will be reclaimed for the least amount of effort. Generational collection might also be appropriate for persistent heaps; but this can be determined only by studying real workloads. In Chapter 5 we discuss how to divide a stable heap into stable and volatile areas; determining whether generational collection is appropriate for either area is left for future work.

For large heaps implemented in virtual memory, one important purpose of garbage collection is to reorganize the heap to provide good paging performance. Reorganizing the heap requires a collector that can move objects, e.g., a copying collector. Copying collectors can increase locality of reference and reduce paging by moving objects that are referenced together to the same page [13, 35, 53].[1]

The other requirement is that the algorithm work well on stock hardware. Without hardware assists, Baker's incremental garbage collector is expensive on stock hardware–it requires a comparison on every heap reference. A variant of Baker's collector [8] substitutes a memory indirection for the comparison, but is still too expensive. Ellis, Li and Appel [17] and Zorn [56] have shown how the virtual memory hardware on stock hardware can be used to facilitate incremental copying garbage collection with lower overhead.

[1]The actual performance of these specific techniques (i.e., a measure of how much they actually increase locality) requires further investigation. Better ways to increase locality may be discovered, but they will also require copying.

We base our atomic incremental garbage collector on the collector of Ellis, Li and Appel (hereafter attributed to Ellis) because it is an incremental, copying collector that runs on stock hardware. Before describing our collector, we review copying collection and incremental copying collection, and we show how a copying collector interferes with recovery. After describing our collector, we discuss its performance and its other interactions with recovery.

3.1 Copying Collection

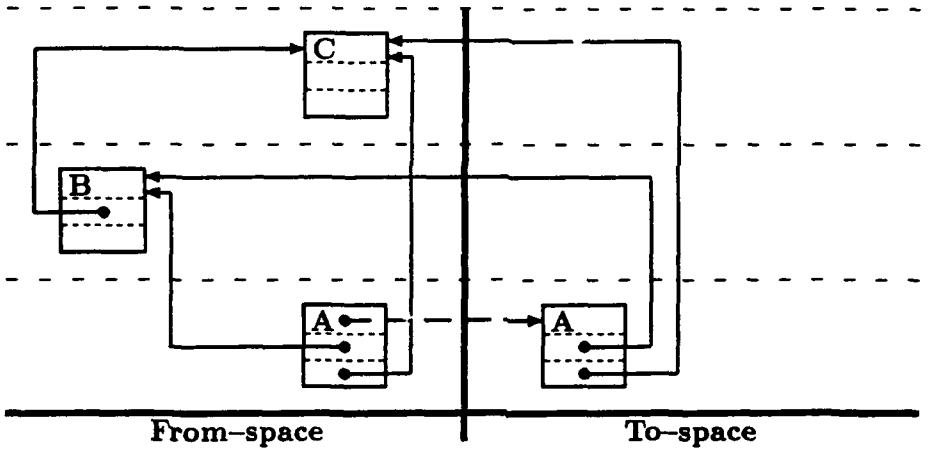
In a system that uses copying garbage collection, memory is divided into spaces. The program allocates new objects in its current space until the memory in the space is exhausted or the paging behavior of the program needs to be improved. At that point, a collection is initiated.

The collection begins with a *flip*: the space in which the program had been allocating is designated *from-space*, and the collector obtains a fresh area of memory called *to-space*. After the flip, the garbage collector copies the accessible or live objects from from-space to to-space. At the completion of the collection, from-space is freed and to-space becomes the current allocation space for the program.

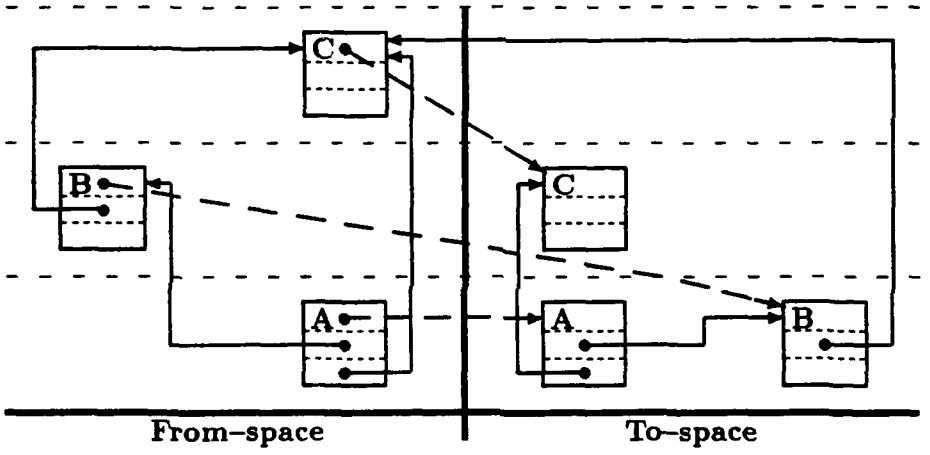
During a collection, as each object is copied, a forwarding pointer is inserted in its from-space copy. The purpose of forwarding pointers is to preserve sharing in the object graph; they prevent an object from being copied into to-space more than once.

The first step of a collection is to copy the root objects to to-space. Then to-space is scanned sequentially for pointers into from-space. As each such pointer is found, it is dereferenced to find the from-space object it references. If the from-space object has a forwarding pointer, then that object has already been copied to to-space, so the pointer in to-space is changed to point to the to-space copy. If there is no forwarding pointer, the object is copied to to-space, a forwarding pointer is left behind, and the pointer in to-space is updated to point to the copy. A collection ends when all of to-space has been scanned. At that point all of the accessible objects have been copied and compacted into to-space.

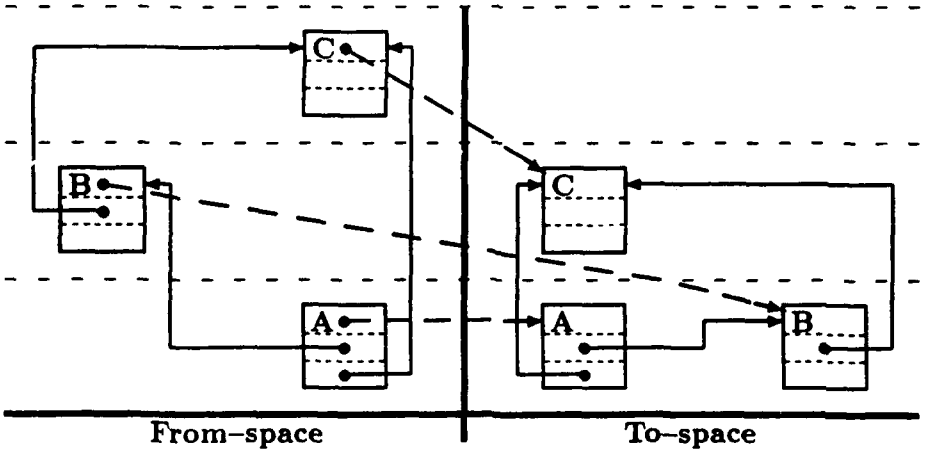
An example of copying collection can be seen in Figure 3.1. The figure only shows live objects. In Figure 3.1.a object A, the root object, is copied to to-space. A forwarding pointer is placed in the from-space copy of object A. Then to-space is scanned sequentially



3.1.a: Root Object is Copied



3.1.b: To-space is Scanned



3.1.c: Forwarding Pointers Preserve Sharing

Figure 3.1: Example of Copying Garbage Collection

for pointers into from-space. Pointers to objects B and C are found in object A. Objects B and C are copied to to-space to the next free locations in Figure 3.1.b. As the sequential scan of to-space continues, a pointer to object C is found in object B. The forwarding pointer in object C indicates that it has already been copied, so object B is updated to point to object C in Figure 3.1.c.

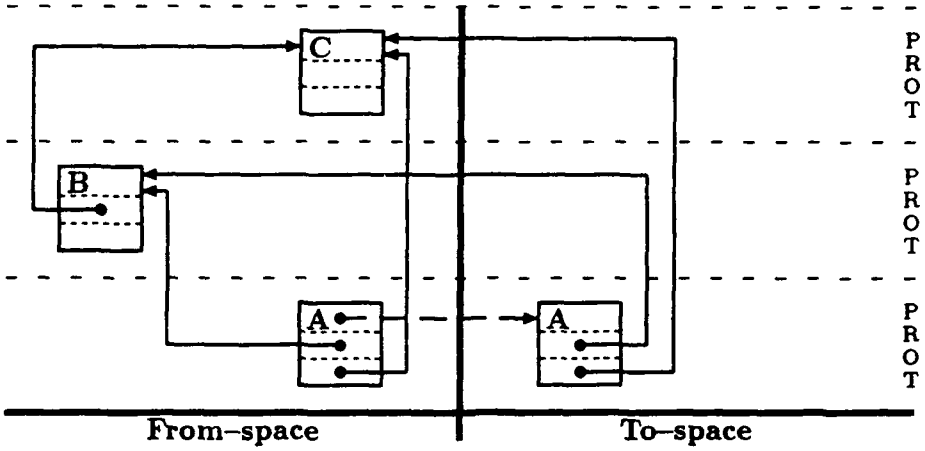
3.2 Incremental Collection

Most incremental garbage collectors have been based on Baker's algorithm [4], which is a copying collector. In Baker's algorithm the program doing useful computation (often called the mutator [16]) and the collector run as coroutines subject to a synchronization constraint that we discuss below. The mutator calls the collector to do some work each time it allocates a new object. When the garbage collector runs, it either scans a fixed number of locations in to-space or it flips. At a flip to-space becomes from-space, a new to-space is allocated, and the collector copies the root objects to the new to-space. Baker's algorithm can be extended in the obvious way to allow the collector and multiple mutators to run in separate threads.

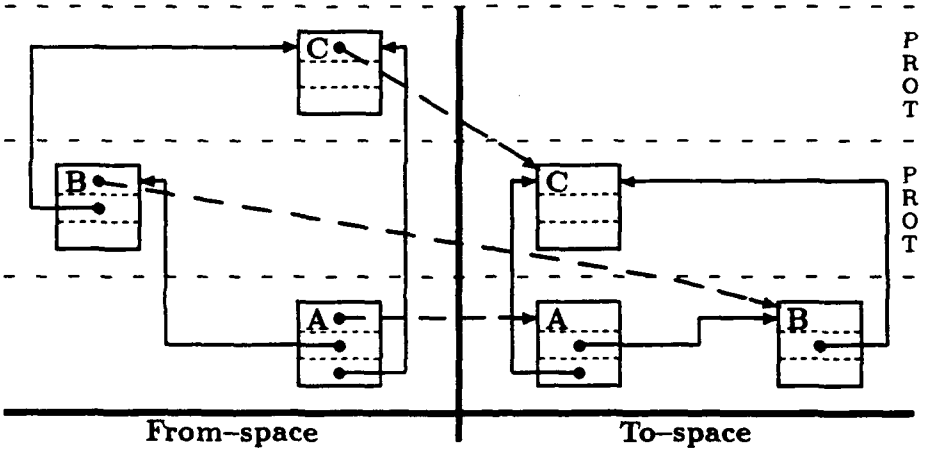
3.2.1 Read Barrier

Synchronization between the mutator and the collector in Baker's algorithm depends on the invariant that the mutator, which in the case of a transaction system includes transactions, never sees a pointer into from-space. This invariant is established at a flip: the root objects (i.e., those objects referenced by a register, a stack, or an own variable) are copied to to-space, and the corresponding registers, stack locations and own variables are updated to point to the to-space copies.

The invariant is enforced during the collection by the so-called "read barrier". The read barrier prevents the program from seeing pointers into from-space. Baker suggested the following implementation of the read barrier: every time the processor fetches the contents of a memory cell, it checks to see whether the cell contains a from-space address. If it does, the collector returns the corresponding to-space address, transporting the appropriate object to to-space as necessary.



3.2.a: After Flip



3.2.b: Page is Scanned

Figure 3.2: Ellis's Read Barrier

Ellis suggests a cheap implementation of the read barrier. After the root set is copied to to-space at a flip, the collector uses the virtual memory hardware to protect the unscanned pages of to-space against both reads and writes. When the program tries to access an unscanned page, the collector fields the resulting trap and scans the page, translating all from-space addresses on the page to the corresponding to-space addresses. Scanned pages do not contain from-space addresses; the program never accesses an unscanned page, so it never sees a from-space address.

Figure 3.2 illustrates Ellis's read barrier. In Figure 3.2.a all the pages of to-space have

been protected at the flip. In Figure 3.2.b, the first page of to-space has been scanned and unprotected. The scanned page does not contain any from-space pointers.

Ellis's approach is cheap; it adds little to the overall garbage collection time since there will be at most one trap per page of to-space. However, Ellis's collector might not be as incremental as a Baker-style collector. The distribution of read barrier traps for both collectors will be skewed to be very frequent just after a flip. In the case of Ellis, each trap requires that a whole page be scanned, whereas a Baker trap only requires that a single location be scanned. Thus, the pauses for garbage collection just after a flip might be long and frequent, thereby defeating the purpose of incremental collection.

Ellis's algorithm requires the ability to scan an arbitrary page of to-space. (Baker's requires that an arbitrary location in to-space be scanned.) If the contents of every memory location are tagged to indicate whether or not it is a pointer, the collector can scan an arbitrary page. If some location is not tagged, some other mechanism is needed. One such mechanism is to construct objects such that the first cell contains a descriptor giving the object's low-level type and length. The descriptor also says where the pointers in the object are located. In addition, the collector also creates a data structure during the scan, called the Last Object Table [40]. This table is an array indexed by to-space page number; the entry for a page contains the location of the last object on that page. To scan an arbitrary page, the collector uses the Last Object Table to find the last object on the previous page. Then it uses the object descriptors to parse the objects on the page, and find and scan their pointers.[2]

Zorn [56] describes a different way of using page protection to implement garbage collection. His method is more incremental than Ellis's, but it increases the overall garbage collection time because it leads to many more traps. Zorn proposes a weaker read barrier. The barrier is weaker because it allows the program to read from-space addresses into its registers, but it does not allow the program to store from-space addresses in memory. To implement the weaker barrier, the pages of from-space are protected and all stores into

[2]In place of the Last Object Table, Ellis suggests using a crossing map, a bitmap that contains a bit for each page of to-space. The bit is set if an object crosses the boundary at the beginning of a page. To make sure that there are pages for which the bit is not set, the allocator may waste space at the end of some pages. Using the Last Object Table, the first object on an arbitrary page can be found faster, and the space taken by the table is likely less than the wasted space in Ellis's scheme.

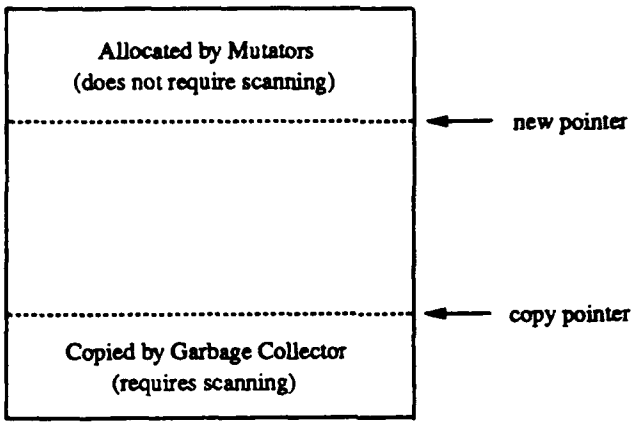


Figure 3.3: Layout of To-space

to-space are checked. Zorn's approach could require a trap per location of to-space; it makes all stores, including those to the stack and to initialize a newly allocated object, more expensive; and it complicates pointer comparisons.

Since Ellis's approach is simpler and cheaper than Zorn's, we use it as the basis for atomic incremental garbage collection. We built a prototype that will enable us to measure the length and frequency of the pauses attributable to the read barrier; we discuss the prototype in Chapter 7.

3.2.2 Allocation

Both the collector and the mutator allocate space for objects in to-space: the collector for the objects it copies and the mutator for the new objects it creates. The read barrier ensures that the new objects allocated by the mutator never contain pointers into from-space. Thus, these objects do not have to be scanned by the collector; Baker proposed a layout for to-space, shown in Figure 3.3, that allows the collector to avoid scanning them. The collector copies objects contiguously to the low part of to-space and allocates by adding to the *copy pointer*. The mutator allocates objects contiguously in the high part of to-space and allocates by subtracting from the *allocation pointer*. We assume the same layout of to-space for our atomic incremental garbage collector.

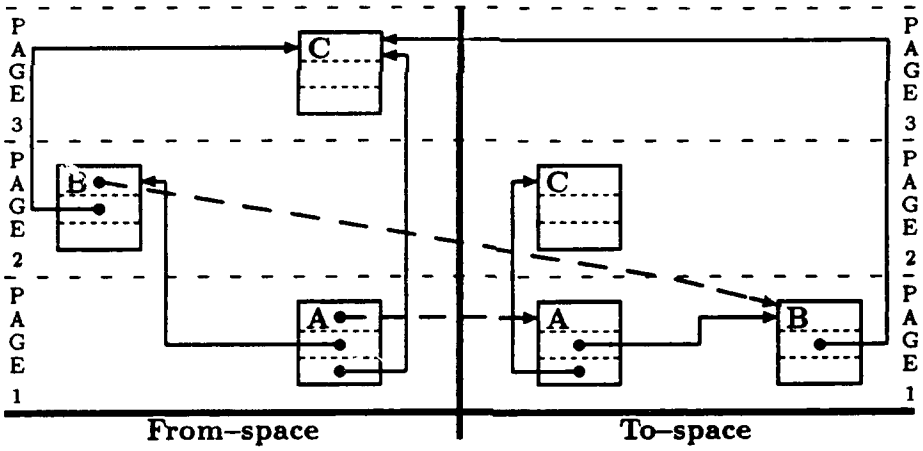


Figure 3.4: Lost Forwarding Pointer

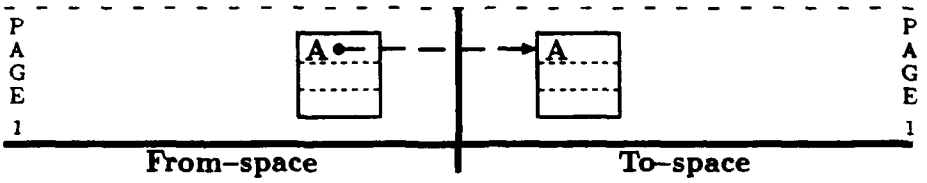
3.3 Why Copying GC is not Atomic

Copying garbage collection is not atomic for several reasons. First, a copying collector modifies from-space by inserting forwarding pointers in objects, and to-space by copying and scanning objects. Second, it moves objects, changing their addresses. We discuss these two problems below; they were first described in a report on our previous research [24, 25]. We describe solutions to the problems in Section 3.4. There are also other interactions between the collector and the recovery system; we describe them in Section 3.5.

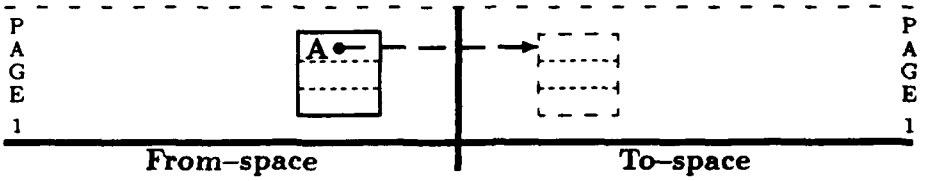
3.3.1 Modifications to Objects

As a copying collector modifies from-space and to-space, the pages of virtual memory are paged into volatile main memory and out to disk by the buffer manager. A crash can easily leave the contents of the disk in an inconsistent state. The following two examples show the kinds of problems that can occur and must be avoided by an atomic collector.

The first example, illustrated in Figure 3.4, shows that forwarding pointers can be lost. If an object is copied, but its forwarding pointer does not survive the crash, then recovering on the basis of information already copied to to-space would not preserve the sharing in the graph of accessible objects. Suppose the collection illustrated in Figure 3.1 were interrupted by a crash after objects A, B, and C had been copied to to-space, but before the pointer to object C from object B had been replaced by a to-space pointer. Figure 3.1.b shows virtual memory just before the crash. Figure 3.4 shows a possible state of the disk after the crash.



3.5.a: Virtual Memory Just Before Crash



3.5.b: Backing Store After Crash

Figure 3.5: Lost Object Descriptor

The crash occurred after pages 1 and 2 of from-space and pages 1 and 2 of to-space had been written to disk, but before page 3 of from-space had been written. The forwarding pointer for object C has been lost, even though the object has already been copied to to-space.

The second example, illustrated in Figure 3.5, shows that the contents of the cell overwritten by the forwarding pointer can be lost. Figure 3.5.a shows an object copied from from-space to to-space; a forwarding pointer was placed in the from-space copy. The forwarding pointer overwrites a cell of the from-space copy. The page of from-space on which the old object version resides is then written to disk. Figure 3.5.b shows what happens if the system crashes before the new version in to-space reaches the disk. The disk will not contain a valid version of the object after the crash. A cell of the object has been overwritten with a forwarding pointer, and is not available on the backing store for from-space. Neither is it available on the backing store for to-space.

The example in Figure 3.5 also shows that not all forwarding pointers are valid after a crash; in Figure 3.5.b there is no object at the forwarding address. If the collector were to be restarted after a crash, object A would never be recopied to to-space.

Approach

The recovery system already solves the modification problem for transactions; so, we could run the garbage collector as a single long user-level transaction or a series of short transac-

tions. However, using a single transaction is inconsistent with our goal of short pauses for garbage collection: the collector would eventually acquire a lock on every object in the heap, thereby preventing other transactions from running. Running the collector as a series of short transactions could easily lead to deadlock or a situation where a long user transaction could prevent the garbage collector from making progress. Deadlocks could occur because the collector would likely acquire locks in a different order than user transactions. A long user transaction could prevent progress by holding a lock on an object that needed to be copied by the collector.

Therefore the garbage collector has to run below the level of user transactions and provide for its own recovery. A user transaction already consists of low-level read, update and allocate actions. To allow atomic garbage collection, we add two additional types of low-level actions: *copy* actions and *scan* actions. A copy action copies a single object from from-space to to-space. A scan action scans a single page of to-space. In Section 3.4 we show how to make these new actions recoverable and in Section 3.5.1 how to handle their synchronization with read, update, and allocate actions.

3.3.2 Movement of Objects

A copying garbage collector moves objects. However, the recovery system records values for objects in the log and these values contain the names of other objects. This leads to a naming problem: what names should the recovery system use in its log to refer to objects? There are two requirements on the solution: (1) the identity of an object must be preserved across garbage collections, and (2) the recovery system needs to find objects on disk after a failure.

Approach

There are two approaches to solving the naming problem: (1) the unique object identifier (UID) approach and (2) the virtual address approach. When an object value is written to the log using the UID approach, the pointers in it to other objects are replaced by the UIDs of those objects. For fast recovery from system crashes, the UID approach requires that a map of UIDs to virtual addresses also be available (either in the log or in virtual memory)

so that objects on disk can be found.

In contrast, when an object value is written to the log using the virtual address approach, pointers in it to other objects remain unchanged. For recovery from media failure, the virtual address approach requires that translation information relating object addresses before a garbage collection to addresses after a collection be written to the log. Depending on the recovery algorithms, some of this translation information may also be required for system crashes.

The information written to the log to solve the modification problems discussed in Section 3.3.1 contains the translation information mentioned above. Since this also solves the naming problem at no extra cost, we choose the virtual address approach.

3.4 Making Copying Garbage Collection Atomic

After a failure during garbage collection, it is sufficient that the system be able to complete the interrupted garbage collection. This requires that the recovery system be able to restore the heap to a state from which the garbage collection can be completed. This is exactly what the redo protocol can do; its repeating history invariant guarantees that the state reached by applying the redo log to the disk is an actual state from the history of the system. We designed our atomic garbage collector to work with the recovery system based on write-ahead logging and repeating history, which we describe in Chapter 4. However, the collector can also be used with other recovery systems as well, such as the recovery system based on versions and intentions lists that we used in our implementation, which is described in Chapter 7.

A copying collection consists of copy and scan steps. We show how to apply the redo protocol to the copy and scan steps.

3.4.1 Copy Step

First we show how to make the copy step recoverable using the redo protocol. A naive application of the protocol is correct, i.e., it makes the copy step repeatable. However, it writes a large amount of information to the log. We show how to optimize the protocol for copy to reduce the amount of information it logs.

Redo Protocol

A copy step makes two modifications to memory: it inserts a forwarding pointer in an object in from-space, and it copies that object to to-space. This modification affects more than one page, yet it must be redoable as a unit. Handling a multi-page modification requires a small change to the redo protocol described in Section 2.2.3: all pages to be modified are pinned in the first step, and they are kept pinned until redo records for all modified pages are in the stable log.

In from-space only the pages containing the cell overwritten by the forwarding pointer are modified. It is easy to ensure that this cell resides entirely on a single page — always overwrite the first cell of the object and during allocation make sure that the first cell of an object never crosses page boundaries. Thus, only one from-space page needs to be pinned even if the object spans more than one page.

Here is a description of a recoverable copy step using the redo protocol without optimizations.

1. Pin the from-space page of the *object cell that will* be overwritten by the forwarding pointer and the to-space pages to which the object is to be copied.

2. Copy the object to to-space and insert a forwarding pointer in its from-space copy.

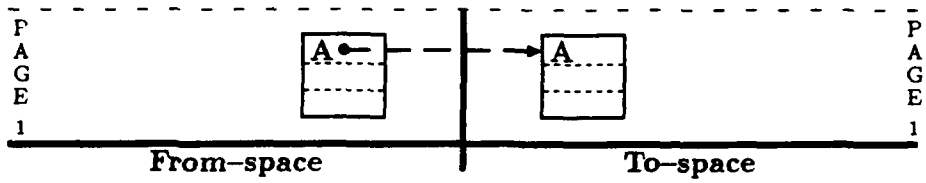
3. Spool a redo record to the log describing the change to from-space. The record contains the from-space address of the object and the to-space address of the object (the forwarding pointer). Spool a second redo record describing the change to to-space. The record contains the to-space address of the object and the value of the object.

4. The copy action is over and the collector can continue.

5. When both redo records are in the stable log, unpin the pages pinned in the first step.

Figure 3.6 illustrates the unoptimized copy step. Figure 3.6.a shows virtual memory after object A has been copied. The two redo records written to the log for the changes to virtual memory are shown in Figure 3.6.b.

The redo protocol makes the copy step recoverable, but it pins at least two pages and it writes the whole object to the log. Using this step to copy every object in the heap would mean writing the whole object graph to the log.



3.6.a: Virtual Memory

	Redo		Redo		
	from-space address of object	forwarding pointer	to-space address of object	object value	

3.6.b: Log Showing Redo Record For Copy

Figure 3.6: The Copy Step

Optimized Copy Step

To optimize the copy step we require that the recovery system be able to recover from-space after a crash to its state at the *last flip*, *except for the from-space cells* overwritten by forwarding pointers. The recovery system based on write-ahead logging and repeating history described in Chapter 4 satisfies this requirement; the recovery system based on versioning and intentions used in our implementation (described in Chapter 7) also satisfies it. In addition we ensure that the cells overwritten by forwarding pointers are recoverable from the redo information we write to the log for the copy. Since transactions do not modify from-space after a flip except for the forwarding pointers, we can redo a copy by re-copying the object from from-space to to-space and taking the cell overwritten by the forwarding pointer from the log. Below we describe this optimized copy step: it requires only one page to be pinned and writes small records to the log. Then we argue its correctness.

Here is the optimized copy step.

1. Pin the from-space page of the object cell that will be overwritten by the forwarding pointer.
2. Copy the object to to-space and insert a forwarding pointer in its from-space copy.
3. Spool a *copy record* to the log. The copy record contains the from-space address of the object, the to-space address of its copy, and the contents of the cell overwritten by the forwarding pointer. Figure 3.7 shows the copy record in the log.

	Copy			
	from-space address of object	to-space address of object	overwritten cell	

Figure 3.7: Log Showing Copy Record

4. The copy action is over and the collector can continue.

5. When the copy record is physically in the log, unpin the from-space page.

First we argue that the pinning optimization is correct, i.e., the to-space pages to which an object is copied do not need to be pinned. The redo protocol pins pages to prevent partial modifications from reaching disk. However, we can mask partial modifications to to-space by the copy step without pinning pages. The collector writes copy records to the log in the order in which it copies objects. Thus, the copy pointer (the allocation pointer of the collector) can be recovered after a crash by looking at the last copy record in the stable log. Any partial modifications to to-space by copy steps whose copy records did not reach the log are at addresses greater than the copy pointer. Thus, these modifications will never be observed.

The to-space pages do not have to be pinned because the disk storage for to-space is fresh, and no useful information in to-space is overwritten. Only the pages in from-space containing the cell overwritten by the forwarding pointer need to be pinned; no other page is modified by the copy step. It is easy to ensure that this cell resides entirely on a single page — always overwrite the first cell of the object and during allocation make sure that the first cell of an object never crosses page boundaries.

Next we show how to repeat the copy step using the information in the copy record.

1. Using the from-space address of the object and the to-space address of the object from the copy record, reinsert the forwarding pointer in from-space.

2. Re-copy the object from the from-space address in the record to the to-space address.

3. Take the contents of the cell overwritten by the forwarding pointer from the copy record and write it to its place in the to-space copy of the object.

Finally we argue that the repeat of the copy step is correct. There are two parts to the argument: (1) the forwarding pointer in from-space is recovered correctly, and (2) the

object in to-space is recovered correctly. The first part is trivial because the redo protocol guarantees correct recovery of the forwarding pointer. We argue the second part for a recovery system based on repeating history. After the flip, the only modification to the from-space copy of an object is the insertion of a forwarding pointer by the copy step of the collector. Thus, if we apply the redo log to the disk (i.e., repeat history) up until the point in the log where the flip occurred, the from-space object is recovered to the same state as at the time of the flip except for the cell containing the forwarding pointer. As we continue to repeat history from the time of the flip until the copy record, the from-space object does not change. Thus, when we redo the copy, the correct value of the object is in from-space except for the cell overwritten by the forwarding pointer. We recover the value for this cell from the copy record.

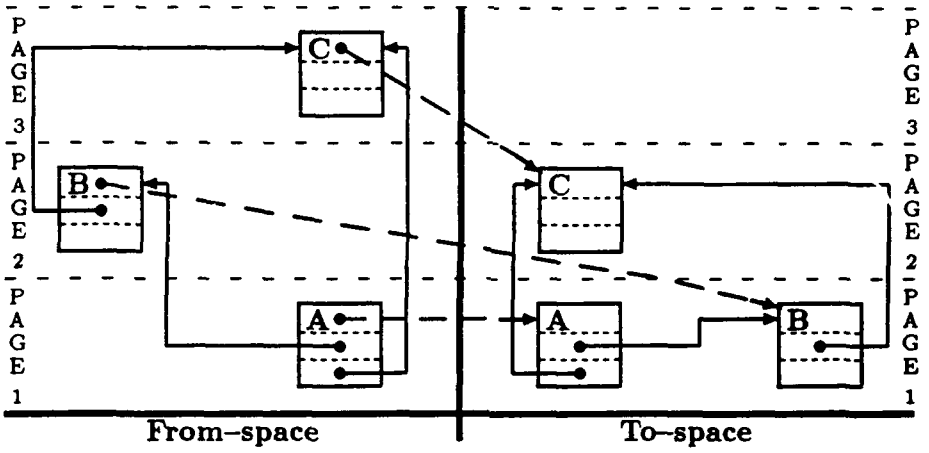
Because of the dependence on from-space for the value of an object that needs to be recopied, disk storage for the from-space object cannot be discarded until the to-space copy has reached disk. This means that the disk storage for from-space must be kept until both (1) the garbage collection is complete, i.e., all accessible objects have been copied to to-space, and (2) every to-space copy has reached disk. The discussion of the scan step will show that this condition must be made even stronger.

3.4.2 Scan Step

A scan action scans a unit of to-space, looks for pointers into from-space, and converts them into to-space pointers. The unit could be a single location, an object, or a page. The page is the best unit for three reasons: (1) it is the unit updated on disk atomically, so it is the natural unit of recovery, (2) it is the unit of synchronization for our incremental garbage collection algorithm, and (3) it is the unit with the least log overhead (the larger the unit, the less records written to the log.) Choosing a unit larger than a page would make the collector less incremental.

A scan action is made recoverable using the redo protocol. The redo protocol prevents the problem of lost forwarding pointers mentioned in Section 3.3.1, i.e., it prevents a to-space pointer to a copied object from reaching disk until the forwarding pointer for the object is recoverable.

1. Pin page to be scanned.



3.8.a: Virtual Memory

Copy A	Copy B	Copy C	Redo	
			address of page	page value

3.8.b: Log Showing Redo Record For Scan

Figure 3.8: The Scan Step

2. Scan page and update its from-space pointers to to-space pointers, copying objects as necessary.
3. Spool redo record containing new contents of page.
4. Unpin page when redo record is physically in the log.

Applying the redo protocol without optimization leads to the entire object graph being written to the log, as in the case of the unoptimized copy action.

Figure 3.8 shows the unoptimized scan step. Figure 3.8.a shows virtual memory after page 1 has been scanned. The redo record written to the log for the scan is shown in Figure 3.8.b.

Optimized Scan Action

We optimize the scan action by reducing the size of its redo record. Instead of writing the entire new value for a scanned page to the log, the optimized scan writes a very small and

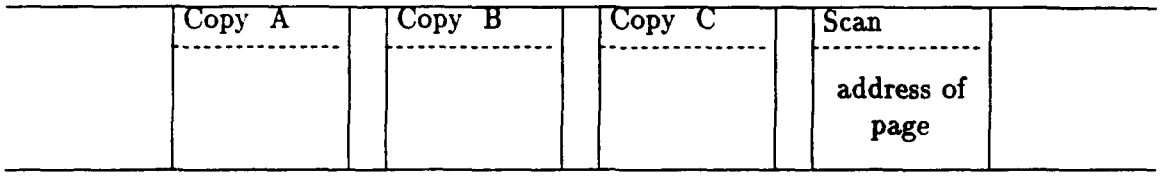


Figure 3.9: Log Showing Scan Record

specialized redo record called a scan record; the record contains the address of the scanned page. Figure 3.9 shows the scan record in the log.

The address of the scanned page is sufficient to make the scan action repeatable. By the time the scan record has been written to the log, there are copy records in the log for all objects referenced by pointers on the scanned page. That means that when the redo log is applied to the disk, all of these objects will be copied to the same place in to-space, and their forwarding pointers will be same as when they were first copied. Therefore when the page is re-scanned, all of its pointers will be assigned the same values that they were assigned during the first scan.

For a recovery system based on write-ahead logging and repeating history the scan action can be optimized even further; no scan record at all need be written to the log. We return to this optimization in Section 3.6.2.

Since the repeatability of the scan action depends on forwarding pointers in from-space, from-space cannot be discarded until both the garbage collection is complete (every page of to-space has been scanned), and every scanned page has reached disk. The writing of scanned pages to disk does not need to occur synchronously; rather, the garbage collector informs the buffer page manager as it scans each page. Then the buffer page manager can schedule the writes according to its own policies. The buffer page manager must also provide an operation that allows the collector to check if all of the scanned pages have been written.

3.4.3 Scanning An Arbitrary Page

To scan an arbitrary page of to-space, the Last Object Table must also be recoverable. Fortunately, the copy records contain sufficient information to recover the table. As each copy record is processed, it is used to update the table entry for the to-space page to which

the object was copied. The Last Object Table can be kept in recoverable or volatile memory. Less work may need to be done to recover it if it is kept in recoverable memory.

3.4.4 Movement of Objects

The information written to the log to solve the modification problem solves the naming problem caused by the movement of objects. Each copy record contains a from-space, to-space address pair for an object. There is one copy record for each accessible object. This is just the translation information required by the virtual address approach to solving the naming problem.

3.5 Other Interactions With Recovery

In the previous sections we described the modification and movement problems and our solutions to them. Here we describe three other interactions between garbage collection and recovery: (1) synchronization between the garbage collector, the transaction system and recovery, (2) roots in the recovery information, and (3) fast recovery for a system failure during garbage collection.

3.5.1 Synchronization

Because the collector moves and modifies objects it must be synchronized with the transaction system. Since it writes recovery information to the log, it also has to be synchronized with the recovery system. This synchronization has to be cheap. First we describe synchronization with transactions; then we describe synchronization with the recovery system.

Synchronization With Transactions

Each transaction is a sequence of elementary actions that read, update, and allocate individual objects. Our approach to synchronization with transactions is to require that a flip occur in an action-quiescent state. The system is action-quiescent when no elementary action is in progress. Since the elementary actions are short, a flip is not significantly delayed by waiting for an action-quiescent state. Given that a flip occurs in an action-quiescent

state, the read barrier ensures that a copying or scanning step of the atomic incremental garbage collector only observes an object in an action-consistent state.

The correctness of the above approach can be seen by viewing the system as a multi-level transaction system with two levels [50]. At the high level, there are user transactions; at the low level, there are elementary actions. An elementary action can be an update, a read, an allocate, a copy, or a scan action. A user transaction is made up of a sequence of low level read, update, and allocate actions. The garbage collector uses copy and scan actions: a copy action copies an object from from-space to to-space, and a scan action scans a single page of to-space.

At the level of transactions, a transaction obtains read and write locks that it holds for its duration. The read and write locks ensure serializability at the transaction level. At the level of actions, a synchronization mechanism ensures that only one action accesses a given object at a time, so that an action always observes a consistent object state. Since the garbage collection actions do not change the abstract values of objects, they are invisible to the transaction level: an object can be copied or scanned even while a transaction holds a write lock.

In the approach described above, synchronization between garbage collection actions and the other low-level actions is implemented cheaply by the read barrier. At a flip the collector obtains a "lock" for every stable object at once by protecting the unscanned pages of to-space. As each page of to-space is scanned, the "locks" for the objects in it are released by changing its protection. Restricting flips to occur in an action-quiescent state ensures that the collector obtains the "locks" at an appropriate time.

Synchronization With the Recovery System

Because the collector writes to the log, we must ensure that a garbage collection trap cannot occur inside a procedure of the recovery system when it is in the middle of writing a record to the log. If it did, the log would not be readable after a crash. A general solution for this problem is to have the recovery system construct an entire record in memory outside of the heap and then copy the record to the log buffer which is also outside of the heap. Clearly no trap can occur while the record is being copied. Other ad hoc solutions that require less

copying are possible, but more complex to implement.

3.5.2 Roots in Recovery Information

Because the collector runs while transactions are active, it must be careful to account for the modifications made by active transactions to ensure that no objects are lost. For example, suppose a stable object A contains a pointer to object B, and that B is not accessible from any other object. Now suppose that a transaction T modifies A to point instead to some object C. If T aborts, the pointer to B should be restored, while if T commits, the pointer to C should be installed permanently, and B becomes garbage. Suppose a collection takes place after T has modified A, but before it commits or aborts. If T modified A directly, and the collector does not look at the recovery data (the undo information), the storage for B will be reclaimed. If T then aborts, there is no way to restore the heap to its original state. Similarly, if T's modification is kept separately from A (in redo information) and the collector does not look at this recovery data, the storage for C might be reclaimed; if T commits later, we will have a problem.

To solve this problem, the collector must use the redo and undo information maintained by the recovery system for active transactions in determining which objects are accessible. An object must be considered accessible if (1) it is directly accessible from the stable root; (2) it is directly accessible from undo or redo information for an active transaction that has modified some other accessible object; or (3) it is accessible from some other accessible object. If objects are updated directly, so that the latest redo information for an object is reflected in the state of the object in the heap, then the redo information can be ignored.

If the recovery information is organized as a log, the log records for active transactions can be used as roots for collection, in addition to the usual roots. However, reading the log during collection to find the records for active transactions can be expensive. We show how to avoid this expense in our discussion of recovery in Chapter 4.

3.5.3 Fast Recovery Even if a Crash Occurs During Garbage Collection

Even if a crash occurs in the middle of a collection, the time for recovery should be short and independent of heap size. Fast recovery depends on the checkpoints and other optimizations a recovery system uses. In this section we describe fast recovery for a recovery system based

on write-ahead logging and repeating history; in Section 7.2.3 we describe fast recovery for the recovery system of our implementation.

The copy and scan actions of the atomic incremental garbage collector have been made atomic using the redo protocol. Therefore the optimizations that work for repeating history and the redo protocol, checkpoints and end-write records, continue to work for the atomic garbage collector. By increasing the frequency of checkpoints, and the frequency at which dirty pages are written back to disk, the time for recovery can be shortened.

In particular, if there is a scan record for a page in the log and there is also an end-write record for that page, that page does not have to be rescanned during recovery. If there is a copy record for an object in the log and there is also an end-write record for the page of from-space holding the object's forwarding pointer, the forwarding pointer does not have to be reinserted during recovery. If there is a copy record for an object in the log and there is also an end-write record for the page(s) of to-space to which the object has been copied, the object does not have to be recopied during recovery. The exact processing of copy and scan records during recovery from a system failure is described in detail in Section 4.8.

3.6 Discussion

In this chapter we showed how to make an incremental collector atomic by dividing its work into copy and scan actions. We made the copy and scan actions recoverable using the redo protocol. In Chapter 4 we show how to incorporate the collector into a recovery system based on write-ahead logging and repeating history. It can also be incorporated into other recovery systems. We summarize the constraints on the recovery system by stating some of the invariants that it, the mutator, and the garbage collector maintain.

1. From-space can be recovered to its state at the time of the last flip (using the disk and the log), with the exception of the cells that were overwritten by forwarding pointers.

2. The cell that gets overwritten by a forwarding pointer is on the disk for from-space or it is in a copy record in the log.

3. The disk backing store for from-space is available until the garbage collection is complete and every scanned page has reached disk.

4. The mutator does not access or modify from-space after the flip.

These invariants are subsumed by the invariants we derive in Chapter 6.

In the remainder of this section we discuss the performance of the collection algorithm, an additional optimization for the scan step, enhancements to Ellis's algorithm to shorten the pauses, and an adaptation of our algorithm to work with Baker's incremental collector.

3.6.1 Performance

A major goal in the design of any computer system is to achieve the required functionality at the minimum cost. For recovery systems, this means that we would like to pay a minimal run-time cost in order to ensure that the state can be recovered quickly after a crash. The repeating history recovery algorithm and its accompanying redo protocol have been designed with this goal in mind.

To keep the run-time cost of recovery low, designers of recovery systems

1. avoid random synchronous writes to disk by writing to a log,
2. avoid synchronous writes to the log except for transaction commit,
3. and then minimize writing to the log.

In fact, high performance transaction systems avoid forcing the log for the commit of every transaction by using group commit [18]. By basing the atomic garbage collector on the redo protocol, we avoided forcing I/O to disk and the log. Then we optimized the protocol for the copy and scan steps to minimize writing to the log.

Our atomic garbage collector never delays itself or transactions because it is waiting for the completion of a synchronous write to disk or to the log. In comparison, Detlefs [15] focused on minimizing writing to the log; as a result his concurrent atomic garbage collector requires both random synchronous writes to the disk and synchronous writes to the log. We discuss the details of his algorithm in Section 8.4.

3.6.2 Further Optimization of Scan

For a recovery system based on write-ahead logging and repeating history the scan action can be optimized further. The scan action pins the page it is scanning until the scan is complete, and all of the copy records for objects referenced by pointers on the page are in the stable log. However, the scan does not write a scan record to the log; instead the buffer

page manager includes an indication that the page is scanned in the next end-write record it writes for the page.

For repeatability of the scan we depend on the copy records for objects copied during the scan, the redo record for the first update to an object on the scanned page, and the end-write record: the read barrier ensures that the redo record for the update action will not be written to the log until the page of the updated object has been scanned; the scan action ensures that the scanned page will not reach disk before the copy records on which it depends are in the log; thus, to repeat history we repeat the scan of the page just before redoing the first update to the page if there is no subsequent end-write record for the page. If there is no update record for an object on a scanned page before the end-write record for that page, we avoid repeating the scan because the end-write record indicates that the scanned page reached disk.

3.6.3 Enhancements to Ellis's Algorithm

Though the distribution of pauses when using Ellis's algorithm cannot be changed, the length of the pauses can be shortened and made predictable. We describe two enhancements to Ellis's algorithm: one to shorten the time for a flip and the second to bound the time taken to scan a page. Both enhancements were suggested by Ellis [17]; they are largely orthogonal to atomic garbage collection.

First, a flip is shortened by delaying some of its work until later in the collection or until the work is required by a trap: instead of copying all of the root objects, the flip copies just the objects referenced by the registers, and protects stack pages and pages containing own variables against access. A trap for one of these pages is handled just like a trap for a page of to-space: the pointers on the page are scanned and the page is unprotected to allow the access.

Second, if a page contains pointers to many large objects, then the time to scan a page may be long and unpredictable. Normally the collector would copy these objects as it finds the pointers to them during the scan (assuming they have not yet been copied). To avoid the unpredictability, Ellis proposes an optimization that allows the collector to delay copying an object until it scans the object itself. The copy can be delayed because the

mutator cannot access an object until it is scanned. The optimization depends on a minor assumption about the implementation of objects: we assume that each object has as its first cell a descriptor containing the size of the object.

Instead of copying a large object when encountering the first pointer to it on a scan, the collector allocates enough room to hold the object as a place holder in to-space, copies the object's descriptor to the first cell of the place holder, writes a forwarding pointer in the from-space object pointing to the place holder, and inserts a back pointer in the place holder pointing to the object in from-space. Later when the collector scans the page with the place holder it notices the large object by reading the descriptor and it copies the rest of the object using the back pointer. This scheme is easily extended so that at most one page of a multi-page object gets copied at a time.

3.6.4 Atomic Incremental GC For Baker's Algorithm

Baker's original incremental garbage collection algorithm and Zorn's algorithm can also be made atomic using the redo protocol. We sketch how it would work for Baker's algorithm. There are two changes with respect to what has been presented so far: (1) the recoverable scan action, and (2) synchronization between the garbage collector and transactions.

Recoverable Scan Action

In Baker's algorithm, the collector scans only a single memory cell at a time in response to a read barrier trap. However, for efficiency reasons (to amortize the cost of pinning) the collector's systematic scan of to-space may need to scan several locations at a time. Therefore, we design the scan action for Baker's algorithm to scan as little as a single location or as much as an entire page. We require only that all of the cells scanned by a single action belong to the same page.

We apply the redo protocol to make the scan action recoverable and optimize it as before.

1. Pin page on which memory cells reside.
2. Scan cells and update from-space pointers in them to to-space pointers, copying the referenced objects to to-space if necessary.

3. If no objects were copied, unpin the page immediately. Else, unpin the page when the copy record for the last object copied is in the stable log.

Writing a scan record for every scanned cell or every group of scanned cells would be expensive. As noted earlier, the scan record is redundant, and we choose not to write it for every scanned cell in Baker's algorithm. However, to reduce the number of pages that need to be rescanned after a crash, the systematic scan of to-space writes a scan record to the log for each page scanned.

Synchronization

In Ellis's algorithm the collector acquires low-level locks on all objects at a flip by virtue of the read barrier, and releases its lock on an object after it scans the page of that object. In Baker's algorithm, the collector also acquires the low-level locks on all objects at a flip, but it releases the lock on an object after it copies the object. When the collector scans an object during its systematic scan of to-space, it must reacquire the low-level lock in the same way that read or update actions acquire it. If the collector is called to handle a read barrier trap and scan a single location in an object, the read or update action accessing that object temporarily releases its lock to the scan action. When an update action makes an access to memory that causes a trap, it must be in a state in which it can release the lock. Specifically, it must not be in the middle of writing to the log; otherwise, the garbage collector will not be able to write its copy records to the log.

Chapter 4

Recovery

In this chapter we assume that all of the roots of the heap are stable and we show how atomic garbage collection can be integrated into a recovery system based on update-in-place and write-ahead logging. Using the resulting algorithms, the time for recovery from a system crash is short and independent of heap size, even if the crash occurs during garbage collection. Chapter 5 shows how the atomic garbage collector and recovery system can be used to support the model of a stable heap described in Chapter 2, i.e., a heap in which the roots are partitioned into stable and volatile sets, and an object persists because it is reachable from a stable root. The approach outlined in the current chapter is appropriate for an object-oriented transaction system such as Avalon, where an object persists because it is created in a persistent area.

We provide an overview of our recovery algorithm. Then we describe the problems arising from the integration of the atomic garbage collector with the recovery system and their solutions: finding roots in the recovery information, translating addresses in the recovery information, allocating spaces recoverably, and recovering independently of heap size. Finally we present algorithms for aborting a transaction and recovering after a system failure.

4.1 Overview

Our recovery algorithms for update-in-place and write-ahead logging are based on the repeating history paradigm of Mohan [34]. However, our algorithms use physical redo and logical undo for all updates, whereas Mohan's allow logical redo for updates to objects totally contained with a single page. We chose physical redo for our algorithms because it is

simpler both to explain and to implement than logical redo. Employing mechanisms similar to those used by Mohan, our algorithms could also support logical redo.[1]

Below we review our assumptions about object allocation. Then we describe how the recovery system works during normal operation, and the invariants it maintains.

4.1.1 Allocation

As described in Section 3.2.2, we assume that the virtual address space for to-space is contiguous, that copy actions allocate upward from the low end of to-space using the copy pointer, and that allocate actions allocate downward from the high end of to-space using the allocation pointer. The algorithms presented here could easily be changed to accommodate other allocation strategies. For example, a to-space could consist of several disjoint virtual address ranges. The changes to the recovery system to support other allocation strategies are obvious; we limit our discussion to this simple strategy in order to simplify our presentation.

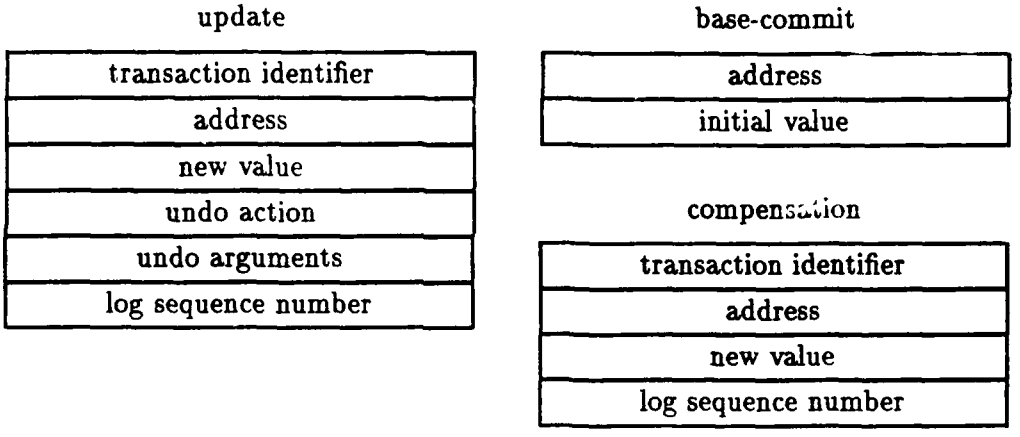
4.1.2 Normal Operation

Below we review the workings of the recovery system during normal operation, and describe the records it writes to the log. Figure 4.1 illustrates the log records.

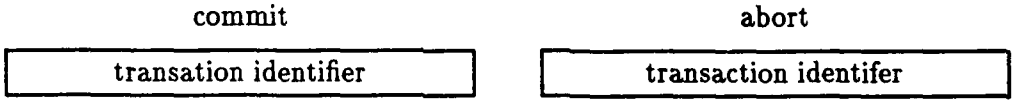
When a transaction updates an object it follows the write-ahead log protocol and writes an *update record* to the log. The update record contains a transaction identifier, the address of the updated region in the object, a new value for the region, the name of an undo action, and arguments for undo. The log records for a transaction are back chained to facilitate undo, so that each update record holds a log address pointing to the previous update record written for the same transaction. The log address is called a *log sequence number* or LSN.

When a transaction allocates a new object, it subtracts the size of the object from the allocation pointer, initializes the object, and writes a *base-commit record* to the log. The base-commit record contains the address for the object, and the initial value of the object. The transaction does not follow the write-ahead log protocol, neither for reasons of undo nor for reasons of redo: the newly allocated area for the object is fresh, so no information is overwritten; and the object can be accessed after a crash only if the base-commit record

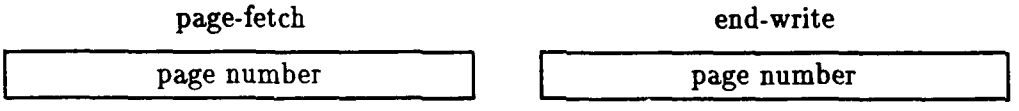
[1]To ensure the idempotence of logical redo, Mohan's algorithm keeps the log address, usually called the log sequence number or LSN, of the last log record describing an update to a page on the page itself.



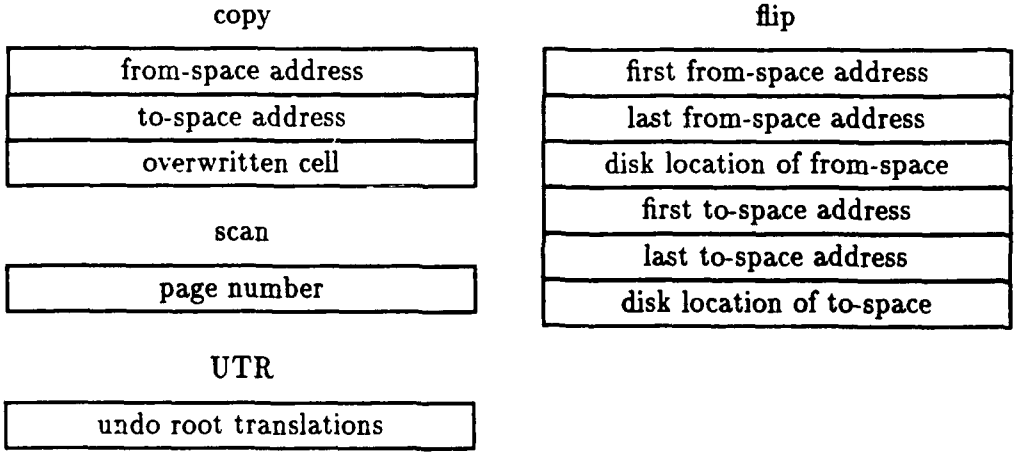
4.1.a: Update Entries



4.1.b: Outcome Entries



4.1.c: Buffer Manager Entries



4.1.d: Garbage Collection Entries

Figure 4.1: Format of Log Entries

is in the stable log.

When a transaction commits, it writes a *commit record* to the log and forces the log buffer to the log. The commit record contains the transaction's identifier.

When a transaction aborts, it follows the backward chain of its update records. As it encounters each update record, it undoes the update using the undo information in the record. Since the undo modifies the object, it follows the write-ahead log protocol and writes a *compensation log record* (CLR) describing the undo action. The CLR contains a transaction identifier, the address of the updated region, a new value for the region, and an LSN for the back chain. The LSN points to the next update that abort has to undo. When all of its updates have been undone, the transaction writes an *abort record* to the log. The abort record contains the transaction's identifier.

The buffer manager writes a *page-fetch record* to the log whenever it reads a page of the stable heap from disk; it writes an *end-write record* to the log just after a page gets written back to disk. Both page-fetch and end-write records contain the page number of the affected page.

The recovery system also takes checkpoints and writes checkpoint records to the log at regular intervals so that the time for recovery after a system failure will be short and independent of heap size. We defer discussion of checkpoints till Section 4.6.

The garbage collector writes copy, scan, flip and UTR records to the log. Copy and scan records are for the copy and scan actions of the atomic garbage collector; we described them in Section 3.4. A UTR record contains translation information for addresses in undo information; we describe it in Section 4.3. A flip record marks the beginning of an atomic garbage collection; we describe it in Section 4.5.

4.1.3 Invariants

We present the invariants that the recovery system maintains as it logs the records described in the previous section; we state them informally here and formally in Chapter 6. First we define some terminology.

The *committed value* of an object is its value just after the commit of the last transaction to update it.

If a transaction holds a write lock on an object, the *current value* of the object is the value of the object seen by that transaction. Otherwise, the object's current value is equal to its committed value.

Object A is said to be *directly reachable* from object B, if object B contains a reference to object A.

Object A is said to be *reachable* from object B if it is equal to B, it is reachable from an object directly reachable from B's current value, or it is reachable from an object directly reachable from B's committed value.

Object A is said to be *commit reachable* from object B if it is equal to B, or it is commit reachable from an object directly reachable from B's committed value.

A transaction is said to be *active in the log* if there are one or more update records for the transaction in the log, but no subsequent commit or abort record.

The log can be divided into a sequence of *garbage collection intervals*. An interval starts with a flip and ends with a flip; it consists of two segments: (1) the garbage collection segment, when all live objects are copied to to-space, and (2) the segment after garbage collection is complete until the next flip.

The repeating history invariant mentioned in Section 2.2.3 holds; it is a lower level invariant that helps to prove Invariants 4.2 and 4.3, which are stated below.

Invariant 4.1 relates the information in the log to the state of the stable heap in virtual memory.

Invariant 4.1 *The disk state that would be produced by redoing the log (both the stable part of the log on disk and the volatile part still in the log buffer) against the disk backing store is the same as the current state of the stable heap in virtual memory.*

The recovery system maintains this invariant by spooling a log record every time an object in the stable heap is modified, whether by a transaction or by the garbage collector.

Invariant 4.2 tells how the committed value of an object can be restored from the log and the disk during normal operation.

Invariant 4.2 *If an object is reachable from a root, then its committed value is in the disk state that would be produced by redoing the whole log (both its stable and volatile parts) against the disk, and then aborting the transactions that are active in the whole log.*

The repeating history invariant ensures that the redo produces a state in which transactions can be aborted. However, some update records whose actions are undone while aborting transactions may have been written to the log during a previous garbage collection interval; their undo information may contain addresses from these intervals. In order for abort to work, the objects at these addresses must be in the heap and the addresses themselves must be translatable to to-space addresses. We discuss these problems in Section 4.2 and Section 4.3 respectively.

Invariant 4.3 tells how to recover the stable state after a crash. It holds because Invariant 4.2 holds and transaction commit forces the log.

Invariant 4.3 *If an object is commit reachable from a root, then its committed value is in the disk state that would be produced by redoing the stable log against the disk, and then aborting the transactions that are active in the stable log.*

4.2 Roots in Recovery Information

In order for the recovery system to work, objects reachable from its redo and undo information must remain in the heap; yet, these objects may not be reachable from the heap's normal roots. In Section 3.5.2 we called this the problem of finding roots in recovery information. In this section, we describe how to solve the problem for a recovery system that uses update-in-place and write-ahead logging.

No special treatment is necessary for the addresses in the redo information in an update or compensation log record. All of the objects referenced by the redo information were in the to-space that was current at the time of the update. If the update needs to be redone during recovery, it is redone against the disk backing store for that space. Furthermore, a copying collector does not collect and reuse individual objects in a space, so the referenced objects are also recoverable by applying the redo log to the disk.

In contrast, special treatment is required for the addresses in the undo information in an update record. When abort applies the undo operation to an object, that object is in the current to-space even though the update record may have been written when a previous space was current. For the undo action to succeed, the objects reachable from the undo information must still be in the heap. Thus, the addresses in the undo information for

active transactions must be treated as roots for collection. These addresses are called the *undo roots* and they include the address of the updated object and all of the addresses in the arguments for the undo action.

We can divide the undo roots into two categories according to when the update records containing them were written to the log: (1) since the last flip, or (2) before the last flip. In the first category, all of the undo roots are to-space addresses so the objects they reference are still in the heap. In the second category, the undo roots may contain references to objects that may not be copied to to-space unless these references are treated as roots for collection. Since these undo roots are in the log, accessing them during garbage collection requires accessing the log.

However, by following a simple heuristic, the garbage collector can avoid most if not all accesses to the log to find undo roots. Only the transactions that were active at the last flip may have written update records to the log before the flip. Furthermore, most transactions are shorter than the time between stable flips. Thus, if the collector defers its check for undo roots until all the objects reachable from the other roots have been copied to to-space and scanned, most if not all of the transactions that were active at the time of the flip will be complete and little or no work will remain.

The exact details for accessing the log and handling the undo roots are presented in Section 4.4.

4.3 Translating Undo Roots

Since some update records for active transactions may have been written to the log before the last flip or even the penultimate flip, both garbage collector and transaction abort require a mechanism for translating undo roots: the garbage collector to from-space addresses and abort to to-space addresses. There is already enough information in the copy records written to the log by the collector to translate the undo roots, but this information is expensive to access. To speed the translation of these records during normal operation we maintain a data structure in volatile memory; to speed translation after a crash we write a special record to the log containing address translations. We begin this section by stating a translation invariant and showing that there is already enough information in the copy

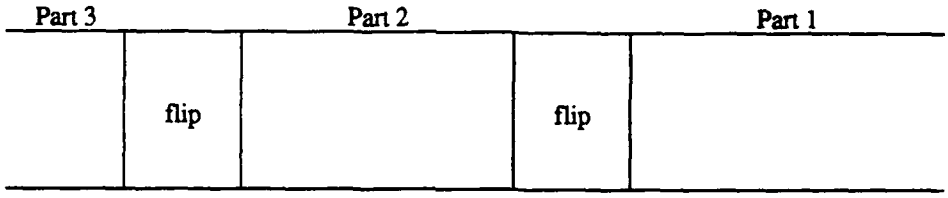


Figure 4.2: Three Log Segments

records to satisfy it. Then we describe the volatile data structure and the log record that speed address translations. In Section 4.4 we show how to maintain the information in the data structure and the log record.

The following translation invariant must hold:

Invariant 4.4 (Translation) *Using records in the log, an undo root in an update record for an active transaction is translatable to a from-space or a to-space address if there is an active garbage collection, or a to-space address otherwise.*

We show that the invariant holds. Divide the log into three segments: (1) the part written since the last flip, (2) the part written since the penultimate flip but before the last flip, and (3) the part written before the penultimate flip. Figure 4.2 shows the three segments. In the first segment the undo roots are to-space addresses; they do not require translation. In the second segment the undo roots are from-space addresses. If garbage collection is not yet complete, then the invariant holds. If garbage collection is complete and from-space has already been freed, then the object at an address given by an undo root must have been copied to to-space by the collector; thus, there is a copy record for it in the log. We can translate the undo root by reading the log records written since the last flip and searching for the appropriate copy record. In the third segment, the undo roots are for the spaces that preceded from-space. These addresses can also be translated by reading the log to find the appropriate copy records.

Transaction abort and garbage collection run during the normal operation of a stable heap. If they have to translate undo roots by reading the log, they will be slow, and they will also consume processing power and I/O bandwidth that could be used for other computations. Therefore, the collector builds a data structure for translation called the Undo Translation Table (UTT) to speed translation. The UTT maps an undo root in an

<<flip number, undo root>, to-space address>
...
<<flip number, undo root>, to-space address>

Figure 4.3: Format of UTR

update record to its corresponding current address. In particular, the UTT maps from a <flip number, undo root> to a <current address, list of transaction identifiers> pair. The flip number is the number of the flip that started the log interval in which the update record was written. Because virtual addresses may repeat every second garbage collection interval but refer to different objects, we pair the flip number together with the address in order to uniquely identify the object referenced by the undo information. The list of transaction identifiers associated with an entry allows the entry to be deleted when no active transaction needs it.

The UTT is kept in volatile memory, so after a crash it is lost. In order to allow fast translation of undo roots after a crash, the information in the UTT is written to the log in an Undo Translation Record (UTR). The UTR contains the address translation map of the UTT, except that it does not contain the transaction identifiers. The transaction identifiers are not required because the recovery system aborts all active transactions when it recovers from a crash. Thus, the recovery system can discard the information in the UTR as soon as recovery is complete, so it does not need the transaction identifiers to selectively discard information. Figure 4.3 shows the contents of a UTR.

4.4 Maintaining the UTT and Processing Undo Roots

Before a garbage collection completes and discards from-space, new entries for the undo roots with from-space addresses must be inserted in the UTT and the tracing of those roots must be complete. These two activities are inter-related and this section describes one procedure to do both.

When a stable heap is first created, the UTT is initialized to the empty state. At every garbage collection, the following procedure for updating the UTT and tracing the objects reachable from the undo roots is executed. The procedure is executed after all of the objects

reachable from the other roots have been copied to to-space and scanned. By that time little or no work should remain.

1. Delete entries for completed transactions from the UTT.
2. Use the addresses in remaining entries as roots for collection, copying each referenced object to to-space if it has not yet been copied and updating the address in the entry to the to-space address of the copy. (This step covers the undo roots in update records written before the penultimate flip.)
3. Trace backwards through the chains of update records in the log for active transactions and find the update records that were written between the penultimate and last flip. For each undo root in an update record:
   (a) Treat the address as a root for collection, i.e., if the object it references has not yet been copied to to-space, copy it.
   (b) If no entry for <flip no., undo root> exists in the UTT, create an entry mapping it to its to-space address.
   (c) Add the transaction to the list of transactions associated with the entry for this undo root in the UTT.
4. Finish scanning to-space.
5. Construct a UTR from the UTT and write it to the log.

Since transaction abort and garbage collection depend on the UTT for the translation of addresses in undo roots, from-space cannot be discarded until the UTR is in the stable log.

4.5 Recoverable Allocation of Spaces

At a flip, the garbage collector allocates a new space for to-space. The new space requires a chunk of virtual addresses and a disk backing store. When a garbage collection completes, the collector deallocates from-space. These allocations and deallocations need to be recoverable so that the recovery system can reinitialize the virtual memory maps after a crash and determine which updates to from-space should be redone.

For the allocation, the garbage collector writes a flip record, shown in Figure 4.4. A flip record contains the virtual address range and the location of the disk backing store for both from-space and to-space. The information for from-space is also available in the last flip record, but we repeat it for easier access.

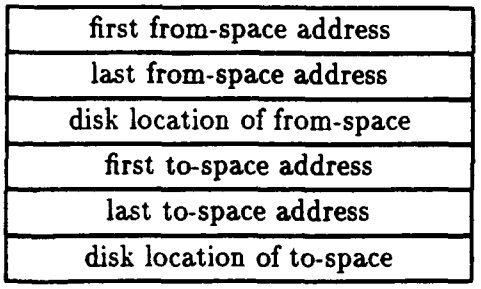


Figure 4.4: Format of Flip Record

For the deallocation, we attach additional meaning to the UTR and wait to write it to the log until from-space can be deallocated. From-space can be deallocated when (1) the garbage collection is complete, (2) every scanned page of to-space has been written to disk since it was scanned, and (3) the UTR is in the stable log. To use the UTR as a sign that from-space can be deallocated, we delay writing the UTR until the first two conditions hold. From-space is not actually discarded until the UTR is in the stable log.

4.6 Recovery Independent of Heap Size

The time for recovery after a crash must be short and independent of heap size. This will not be the case if recovery must read the whole log. Therefore, the recovery system checkpoints periodically: in a low-level action-quiescent state it constructs a checkpoint record and writes the record to the log. The record summarizes information that recovery could obtain by processing the prefix of the log written before the checkpoint. The buffer manager also writes dirty pages back to disk regularly and independently of checkpoints. Frequent checkpoints and regular flushing of dirty pages to disk by the buffer manager allow the time for recovery to be short and independent of heap size.

In this section we describe the contents of the checkpoint record and how it is constructed. In Section 4.8 we describe in detail how the checkpoint record, the other records in the log, and the disk are used after a system failure to recover the stable heap.

Figure 4.5 shows the contents of a checkpoint record. The first two items in the record, the buffer page list and the transaction list, also appear in Mohan's algorithm. The last six items, the LSN of the last UTR, the LSN of the last flip, the address of the Last Object Table (LOT), the Scanned Set (SS), the allocation pointer, and the copy pointer are related

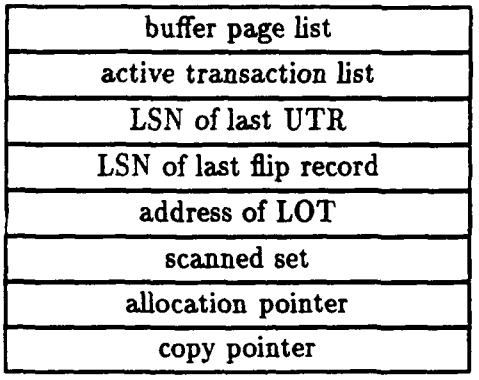


Figure 4.5: Format of Checkpoint Record

to atomic garbage collection.

The buffer page list in the checkpoint record summarizes the information in the page-fetch and end-write records in the prefix of the log before the checkpoint. It is a list of <page number, LSN> pairs, one entry for each dirty page at the time of the checkpoint. The LSN is the LSN when it first became dirty.

The transaction list in the checkpoint record summarizes information about active transactions in the prefix of the log before the checkpoint. It is a list of <transaction identifier, LSN> pairs, one for each active transaction. The LSN points to the last update or compensation log record written for the transaction.

After a crash, recovery uses the translation information in the last UTR written to the stable log in order to translate addresses in the undo information for the transactions it aborts. It also uses the mapping information in the last flip record to re-initialize the virtual memory maps. The LSN of the UTR and the LSN of the flip record in the checkpoint record allow the last UTR and flip records written to the log to be found quickly. Alternatively, the contents of these records could be included directly in the checkpoint record.

The Last Object Table (LOT) allows the garbage collector to scan an arbitrary page of to-space; it was introduced in Section 3.2. Recovery reconstructs the LOT after a crash using information in the copy records written to the log since the last flip. A checkpoint could include the whole LOT in its record to summarize the information in the copy records preceding it. However, the LOT is large. Instead, the garbage collector maintains the LOT in the stable heap, a checkpoint records the address of the LOT in its checkpoint record,

and the buffer manager writes page-fetch and end-write records to the log for the pages of the LOT just like any other page of the stable heap. Then the LOT is recovered like other objects, except that its "update" actions are the copy actions, and its "update" records are the copy records.

The Scanned Set (SS) is the set of to-space pages that have been scanned. After a crash, recovery uses the scan records written since the last flip to rebuild the SS. Then it uses the rebuilt SS to reset the protections on the pages of to-space. A checkpoint records the Scanned Set in its record to summarize the information in scan records written since the last flip and until the checkpoint. The SS is represented as a bitmap, one bit for each page of to-space, both in the stable heap and the checkpoint record. If the bitmap is too large for the checkpoint record, the address of the SS could be kept in the record and the SS could be maintained in a way similar to the LOT, except that its "update" records would be the scan records.

The allocation and copy pointers in the checkpoint record summarize information that could be recovered from the base-commit and copy records that precede the checkpoint.

4.7 Transaction Abort

Abort runs on top of garbage collection, so the read barrier prevents it from accessing an object in to-space until the object has been scanned. However, abort also accesses undo information from the log, which may contain from-space addresses or even addresses that predate from-space. To prevent these out-of-date addresses from creeping into scanned objects, abort translates them to to-space addresses before using them to undo an update.

Here is the procedure to abort a transaction.

1. Trace backwards through the chain of update records in the log for the transaction, skipping those records for which a compensation log record has already been logged. For each update record:

   (a) Translate the address of the object for which the record was written and the addresses in the undo information to to-space addresses, copying objects to to-space if necessary. The translator handles four cases according to when the record was written to the log and whether garbage collection in the current interval is complete:

      i. The record was written since the last flip: no translation is necessary.

ii. The record was written before the last flip, but since the penultimate flip, and garbage collection in the current interval is not complete: The address is a from-space address. If the referenced object has not yet been copied to to-space, copy it. Translate the address to a to-space address using the forwarding pointer in the from-space object.

iii. The record was written before the penultimate flip and garbage collection is not complete: Translate the address to a from-space address using the UTT. If the object referenced by the from-space address has not yet been copied to to-space, copy it. Translate the from-space address to a to-space address using the forwarding pointer in the from-space object.

iv. The record was written before the last flip and garbage collection in the current interval is complete: Translate the address using the UTT. The resulting address is a to-space address.

(b) Undo the update using the translated information.

(c) Write a compensation log record. The compensation log record should contain only to-space addresses. Link the CLR to the update record that precedes the one just undone in the transaction's chain.

2. Write an abort record to the log containing the transaction's identifier.

As an alternative to the above procedure, abort could run below the level of the garbage collector and see from-space addresses. Then, if abort touched a page that had already been scanned, it would have to reprotect the page and remove it from the set of pages that had been scanned. This alternative is coded in the formal description of the algorithms in Chapter 6. In practice this alternative is harder to implement, because the abort must run in a different protection domain than the rest of the run-time recovery mechanisms so that it can access the protected pages. Also, if the object whose modification is undone is small, rescanning the page on which it resides is unnecessary additional work.

4.8 Full Recovery Algorithm

The recovery algorithm for system failure makes three passes over the log. The first, or *analysis*, pass determines which pages of virtual memory might have been dirty at the time of the crash, and thus might need updates reapplied to them. It also determines which transactions were active at the time of the crash. The second, or *redo*, pass processes the log in the forward direction and repeats history; i.e., starting with the first log record that describes an update that may not have reached disk, it redoes the updates recorded in

update, compensation, base-commit, copy and scan records on the pages that were dirty. The third, or *undo*, pass goes backward through the log and aborts the transactions that were active at the time of the crash.

4.8.1 Analysis

The analysis pass is an optimization to reduce disk I/O during recovery; using the information obtained from it, recovery can avoid redoing many updates that reached disk before the crash.

The analysis pass makes one forward pass through the log. On that pass it

1. determines which pages might have been dirty at the crash and for each dirty page the LSN (log sequence number) current at the time it was fetched into memory,
2. determines which transactions were active at the crash and for each active transaction the LSN of its last log record,
3. determines min-LSN, or the LSN of the first log record that needs to be redone during the redo pass,
4. finds the LSN of the last stable flip,
5. initializes the UTT,
6. initializes the SS,
7. initializes the allocation pointer,
8. initializes the copy pointer,
9. and determines the virtual address range and disk store for the spaces that were active.

Analysis builds two tables. The first, called the Buffer Page Table (BPT), maps page numbers to LSNs. The BPT is initialized from the checkpoint record; at the end of the analysis pass it contains an entry for each page that was dirty at the time of the crash. The LSN in the entry corresponds to the last time that the page was fetched into main memory.

The second table, called the Transaction Table (TT), maps transaction identifiers (TID) to LSNs. The TT is initialized from the checkpoint record; at the end of the analysis pass it contains an entry for each transaction that was active at the time of the crash or was in the process of aborting but its abort record was not yet in the stable log. The LSN in the entry corresponds to the last update or compensation log record in the stable log for that transaction.

Min-LSN is calculated by taking the minimum over all of the LSNs in the BPT. All of the updates recorded in the log before min-LSN are on disk. Thus, the redo pass through the log can begin at min-LSN.

Here is a complete description of the analysis pass.

1. Starting with the most recent checkpoint record written to the log, make a forward pass through the log and process each log record according to its type:
   (a) Checkpoint record:
      i. Initialize the BPT, the TT, and the SS from the corresponding items in the checkpoint record.
      ii. Set the LSN of the last flip and the LSN of the last UTR according to their entries in the checkpoint record.
      iii. Set the allocation and copy pointers according to their respective entries in the checkpoint record.
   (b) Page-fetch for page p: Insert <p, LSN of this record> in BPT.
   (c) End-write for page p: Delete entry for page p from BPT.
   (d) Update record or CLR for transaction t: Insert or change existing entry in TT for t to <t, LSN of this record>.
   (e) Commit or abort for transaction t: delete entry for transaction t from TT.
   (f) UTR:
      i. Initialize UTT from UTR.
      ii. Delete entries for from-space pages from BPT.
      iii. Set the LSN of the last UTR to the LSN of this record.
   (g) Flip:
      i. Set the LSN of the last flip to the LSN of this record.
      ii. Remember the mapping information in the flip record.
2. If the UTT has not been initialized (i.e., the LSN of the last UTR is less than the LSN of the checkpoint record), read the last UTR and use it to initialize the UTT.
3. If no flip record has been read (i.e., the LSN of the last flip is less than the LSN of the checkpoint record), read the last flip record and remember the mapping information in it.

4.8.2 Redo

The redo pass repeats history by replaying the log, determines the allocation and copy pointers for the stable heap, reprotects the unscanned pages of to-space, and recovers the Last Object Table. At the end of the redo pass, the stable heap is in a state that it was in

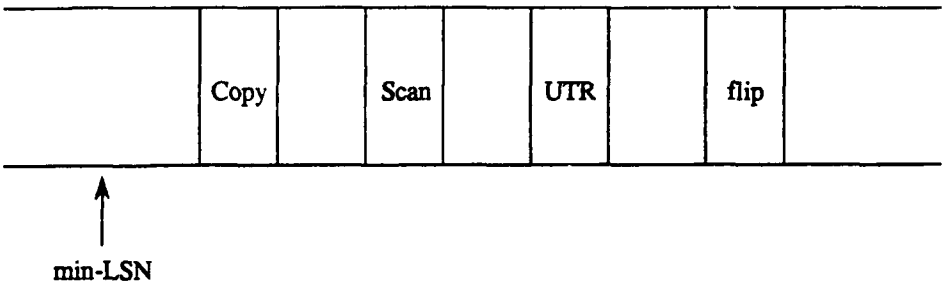


Figure 4.6: Copy and Scan Records Before the Last Flip

at some point before the crash. This is a state in which active transactions can be undone using transaction abort and a state from which garbage collection can be re-started and completed. Also, the allocation pointer points to the first location in to-space that can be allocated when transactions continue, and the copy pointer points to the first location to which an object can be copied when garbage collection continues.

To repeat history, the redo pass starts at min-LSN and redoes the modifications recorded in update, compensation, base-commit, copy, and scan records that might not have reached disk before the crash. A modification may affect more than one page. Redo uses the information in the BPT, collected by the analysis pass, to determine if there are end-write records for the modified pages in the log. If there is an end-write record for a page, a modification to that page does not have to be redone. The end-write record is in the log if there is no entry for the page in the BPT, or if there is an entry, but the fetch LSN recorded for the page in the entry is greater than the LSN of the modification's record.

The updates recorded in copy and scan records written before the last flip never have to be redone; their effects are guaranteed to be on disk. We explain why below; Figure 4.6 pictures the log records mentioned in the explanation. Before starting a new garbage collection interval, the collector deallocates the from-space from the previous collection interval. In order to deallocate a from-space, the collection in the interval must be complete, all of the scanned pages of the interval's to-space must have reached disk since they were scanned, and a UTR for the interval must be in the stable log. Thus, the effect to to-space of any copy or scan record between min-LSN and the last flip is guaranteed to be on disk.

Redo recovers the allocation pointer by processing information in base-commit records. Redo does not need to recover an allocation pointer for from-space, so it does not have to

consider base-commit records recorded before the last flip. Allocate actions allocate in order of decreasing addresses; however, the order of base-commit records in the log may not reflect the actual order of allocation. This may occur because many mutators run concurrently; an allocate action could be interrupted between the time it allocates space for an object and the time it writes the base-commit record for the object to the log. Therefore, we recover the allocation pointer by taking the minimum over the allocation pointer in the checkpoint record and the pointers that are recovered from the base-commit records in the suffix of the log following the checkpoint record.

Redo recovers the copy pointer by processing information in copy records. It does not need to recover a copy pointer for from-space, so it does not have to consider copy records recorded before the last flip. Copy actions allocate in order of increasing addresses and the collector writes copy records in the order in which objects are copied. Thus, the copy pointer can be recovered from the last copy record in the stable log; if there is not copy record in the suffix of the log following the checkpoint, it can be recovered from the checkpoint record.

Redo recovers the Last Object Table by processing information in copy records. The object allocated at the highest address on a page is the last object on the page. Since we only need to recover the LOT for to-space, we do not have to consider copy records written before the last flip.

Redo determines which pages of to-space were scanned before the crash by processing the scan records and using them to reconstruct the SS. Again, since we only care about to-space, we do not need to consider scan-records written before the last flip.

Here is a complete description of the redo pass.

1. Initialize the memory maps for to-space. If the LSN of the last UTR is less than the LSN of the last flip, then initialize the maps for from-space.
2. Starting at min-LSN, the minimum LSN for all of the entries in the BPT, make a forward pass through the log and process each log record according to its type:
   (a) CLR or update record: For each page affected by the update, if the updated page is in the BPT and the LSN of this record is greater than or equal to BPT[updated page number], then redo the part of the update for that page.
   (b) Base-commit for object at address *o*:
      i. For each page affected by the base-commit, if that page is in the BPT and the LSN of this record is greater than or equal to BPT[page number], then redo that part of the base-commit for that page.

ii. If LSN of this record is greater than the LSN of the last flip then set the allocation pointer to the minimum of its current value and *o*.

(c) Copy from address *f*, to address *t*, and the previous contents of the cell overwritten by the forwarding pointer is *o*: If the LSN of the copy record is greater than the LSN of the last flip then:

i. If the page of *f* is in the BPT and the LSN of this copy record is greater than or equal to BPT[page of *f*], then reinsert the forwarding pointer in the from-space object.

ii. If the page of *t* is in the BPT and the LSN of this copy record is greater than or equal to BPT[page of *t*], then insert *o* in the first cell of the object in to-space.

iii. For each to-space page in the BPT affected by the copy for which the LSN of this copy record is greater than or equal to BPT[to-space page number], redo the part of the copy for that page.

iv. Set the copy pointer to the next address after the object at t.

v. Set LOT[page of *t*] to the maximum of its current value and *t*.

(d) Scan for page *p*: If the LSN of the scan record is greater than the LSN of the last flip, then

i. If page *p* is in the BPT and the LSN of scan record is greater than or equal to BPT[*p*], then redo the scan.

ii. Put page *p* in the SS.

(e) Flip:

i. Set allocation pointer to the low address in the virtual address range.

ii. Set the SS to a state where all pages are unscanned.

3. Protect each page between the low end of to-space and the copy pointer, except for those pages in the SS.

4.8.3 Undo

Undo aborts each transaction that was active at the time of the crash. It follows the abort procedure for each transaction in the TT.

4.9 Discussion

In this chapter we described a recovery system based on write-ahead logging and repeating history, and we showed how to integrate our atomic garbage collector with it. We identified two key problems: (1) finding undo roots in the log and (2) translating the undo roots to to-space addresses. We argued that by delaying the garbage collector's handling of undo roots until near the end of a collection, the collector would usually avoid accessing the log.

However, for the cases where there really are undo roots in the log, we showed how to maintain information to translate these undo roots to to-space addresses efficiently. Both the cost to maintain the information and the cost to use the information is low. We also showed how to use checkpoints to keep the recovery time short and independent of heap size, even if a crash occurs during garbage collection. Finally, we showed how to do transaction abort and how to recover after a crash.

Until now we deferred discussing two additional optimizations in order to simplify the presentation of the recovery algorithm. First, the page-fetch records are superfluous. The information in page-fetch records can be inferred from update, compensation, copy, scan and base-commit records. Namely, when one of these records is processed during the analysis pass, if a page the record updates is not in the Buffer Page Table (BPT), that page should be inserted in the BPT with the LSN of the record. Second, scan records can be combined with end-write records. We described this optimization in our discussion of the atomic garbage collector in Section 3.6.2.

Chapter 5

Volatile Objects in a Stable Heap

The algorithms presented so far provide low overhead recovery and automatic storage management for stable objects. Nevertheless, there is a cost; update, copy and scan actions must follow the write-ahead log protocol. Not all objects in a stable heap are stable; some are volatile, and need not survive crashes, for example, objects associated with procedure invocations, objects local to transactions, objects used to implement the transaction system, and some global objects. For performance it is important that the cost of accessing, modifying and garbage collecting these volatile objects be low. In this chapter we show how to avoid the costs of recovery and atomic garbage collection for volatile objects. First we show how to avoid recovery costs by tracking the dynamically changing set of stable objects. Then we show how to avoid the costs of atomic garbage collection by dividing the heap into stable and volatile areas.

5.1 Concurrent Tracking of Newly Stable Objects

The state of a stable object, an object that is commit reachable from a stable root, must persist across failures. To ensure the persistence of a stable object, the recovery system follows the write-ahead log protocol and records an update record in the log whenever the object is modified. A volatile object, an object that is reachable from a volatile root but not from a stable root, does not need to persist. Thus, writing to the log when it is modified is unnecessary and only increases the cost of the modification. We would like to avoid this cost when possible.

Volatile objects may be atomic or non-atomic. Since non-atomic objects may never be

accessible from a stable root, they never require recovery. Therefore, the discussion that follows concerns atomic objects only, and in it the word object should always be read atomic object.

To avoid the logging cost for volatile objects, the recovery system uses separate recovery schemes for stable and volatile objects. For stable objects it uses update-in-place and writes an update record containing both redo and undo information to the log (following the write-ahead log protocol). For volatile objects it uses update-in-place, but it keeps the undo information in volatile memory. To remember which volatile objects require undo, the recovery system also keeps track of a Modified Volatile Object Set (MVOS) for each transaction.

All newly allocated objects are volatile. A volatile object can become stable if a transaction updates a previously stable object and makes the volatile object reachable from it. These objects are called *newly stable objects*. A newly stable object actually becomes stable when the transaction that made it reachable commits. When an object becomes newly stable, the recovery system writes an initial value to the log for it, and treats it as a stable object for all subsequent updates to it.

To differentiate between stable and volatile objects, the recovery system keeps track of the stable and newly stable objects. A stability tracker finds newly stable objects as they become reachable and makes sure that their values are recoverable. The idea to track newly stable objects is due to Oki [38], but his algorithm is incorrect; Section 5.1.3 shows why.

In the remainder of this section we describe the changes to the recovery system in greater detail. Throughout we assume an approach to recovery based on physical redo and logical undo. We begin by deriving a correctness condition. Then we introduce the data structures used by the tracking algorithm and the invariants on them, and show that the invariants imply the correctness condition. We present the tracking algorithm, show how to incorporate it into update actions, and show how it works through an example. Then we sketch the changes to transaction abort and recovery. Finally we show simplifications to the algorithm possible for a recovery system that uses physical undo.

5.1.1 Correctness

In our derivation of correctness criteria for a stability tracking algorithm we use terminology from Section 4.1.3. We also define some additional terminology. An object is said to be *recoverable using the stable log* if its last committed value is in the stable heap state produced by redoing the stable log against the disk, and then undoing the transactions that are active in the stable log. An object is said to be *recoverable using the log* if its last committed value is in the stable heap state produced by redoing the entire log (stable and volatile) against the disk, and then undoing the transactions that are active in the entire log. When we use the term recoverable without further modification we mean recoverable using the stable log.

Determining how soon an object must be recoverable is the key question for the tracking of stable objects. Clearly a newly stable object must be recoverable when the transaction that makes it stable commits. However, this is not soon enough for a recovery system that uses logical undo.

The repeating history invariant requires that repeating history (i.e., redoing the stable log against the disk) produces an actual state from the history of the stable heap. This is a state from which it makes sense to undo active transactions using logical undo. To satisfy the invariant, a recovery system that tracks newly stable objects must ensure that repeating history restores the state of all objects reachable from the stable roots. It must also ensure the recoverability of all objects reachable from an object for which there is an update action that may need to be undone, and all objects reachable from the undo information for that update. This implies correctness conditions for tracking newly stable objects:

Correctness Condition 5.1 *If an object is reachable from a stable root in the stable heap state produced by redoing the log against the disk, that object is recoverable from the log.*

Correctness Condition 5.2 *If an object is reachable from an update record for an active transaction in the log, that object is recoverable from the log.*

To achieve these correctness criteria, the algorithm we present below ensures that the objects reachable from an update record are recoverable from the log before an update action spools the record to the log.

5.1.2 Data Structures

To keep track of the stable objects and to find newly stable objects, the recovery system uses two sets. We describe the sets and the invariants on their states.

Following Oki [38], we call the first set the Accessibility Set or AS. The AS includes all objects that are reachable from a stable root. It is a superset of these objects, so it may contain objects that are no longer reachable or objects that are not yet reachable because the update action making them reachable has not yet completed. An update action uses the AS to determine if the object it is modifying may be reachable from a stable root. If so, it scans the updated part of the object for newly stable objects and writes an update record to the log.

The recovery system maintains two invariants for the AS:

Invariant 5.1 *If an object is reachable from a stable root, it is in the AS.*

Invariant 5.2 *If an object is in the AS and it is reachable from a stable root, then its value is recoverable using the log.*

The second clause in the hypothesis of Invariant 5.2 concerning reachability from a stable root is technical; it allows the recovery system to insert an object in the AS without following the redo protocol. We discuss this issue in greater detail in Section 5.1.8. Taken together these invariants imply Correctness Condition 5.1.

The second set is called the Logged Set or LS. The tracking algorithm maintains the following invariant for the LS:

Invariant 5.3 (LS) *If an object is in the LS and it is reachable from a stable root, then all of the objects reachable from it and its undo roots are in the AS, i.e., they are recoverable using the log.*

Membership in the LS allows the tracker for newly stable objects to prune work from its search tree: if an object is in the LS, the tracker does not process it or the objects reachable from it.

The recovery system maintains the AS and the LS: a stability tracker finds newly stable objects and inserts them in the sets at the appropriate time; update and undo actions

invoke the tracker when they make an object newly stable; recovery ensures that the proper objects are in the sets after a crash; and the garbage collector deletes objects from the sets when they are collected as garbage. A well-engineered implementation also requires a good representation for the sets. Below we discuss these issues, and also present the other changes to the recovery system necessary to avoid logging costs for volatile objects.

5.1.3 Concurrent Stability Tracker

Before an update action writes its update record to the log, it invokes the stability tracker to find the objects that it will make newly stable. A single update may make a large graph newly stable. Many transactions run concurrently, and we do not want to delay an update of one transaction because an update by another transaction makes a large graph newly stable. Thus, we require that invocations of the tracking algorithm be able to run concurrently with updates and with each other. This concurrency should be possible even if more than one stability tracker is working on parts of the same object sub-graph. We achieve this concurrency by making the critical sections (or atomic steps) of the tracker very short.

To track newly stable objects from an object o, the tracking algorithm does a depth-first traversal on the object graph rooted at o. On the way down the graph, it inserts newly stable objects (objects that are not yet in the AS) into the AS and it writes their initial values to the log. Inserting an object in the AS and recording its initial value in the log occur in an atomic step. When the downward traversal reaches an object in the LS, it does not search the sub-graph rooted at that object; the object's membership in the LS indicates that the sub-graph is already recoverable from the log. On the way up from the traversal, the tracking algorithm inserts newly stable objects in the LS.

Why Both AS and LS

Before describing our concurrent tracker in greater detail, we explain why it uses both the AS and the LS. Membership in the AS determines when an update needs to be logged. Membership in the LS allows the stability tracker to avoid searching parts of the object graph. Using separate sets for these functions allows greater concurrency.

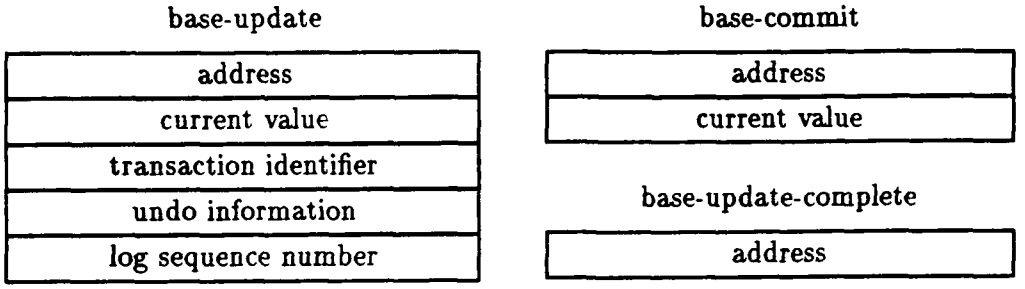


Figure 5.1: Log Records for Initial Object Values

A single set could provide both the function of the AS and the LS, at the cost of longer atomic steps. Assume that the set is called the AS. The recovery system must still maintain Invariants 5.1 and 5.2 for the AS. However, now it must also maintain Invariant 5.3 for the AS instead of the LS; i.e., if an object is in the AS and it is reachable from a stable root, then all of the objects reachable from it and its undo roots are also in the AS. To ensure the latter invariant, the tracker would have to process the whole object graph rooted at a newly stable object in a single atomic step. For large object graphs this tracking step would be long, and it would prevent progress by other transactions.

Oki's algorithm [38] uses a single set, but it is wrong because its stability tracker fails to maintain Invariant 5.3 for the AS.[1] Thus, it incorrectly prunes work and allows a transaction to commit before all of its effects are recoverable. Section 5.1.5 describes an example that Oki's algorithm handles incorrectly.

Log Records Written by the Tracker

If any active transaction holds a write-lock on a newly stable object, the tracker writes the object's initial value to the log in a *base-update record*; otherwise it writes a *base-commit record*. Figure 5.1 pictures these records. The base-commit record contains the address of the object and its current value. The base-update record contains the address of the object, its current value, and undo information; the current value is required in case the locking transaction commits, and the undo information in case it aborts. The undo information is a sequence of undo operations together with their arguments; it is a sequence rather than a

[1] Oki describes his tracking algorithm in the context of the recovery system used by the current Argus implementation. We describe how our algorithm fits into that implementation in a technical memorandum [23].

single operation since the locking transaction may have updated the object more than once. The undo information is a source of undo roots for garbage collection. The record also contains the transaction identifier of the locking transaction (which may be different from the transaction doing the tracking) and an LSN that points to the locking transaction's previous update or base-update record.

The undo information in a base-update record complicates recovery. Unlike an update record, a base-update record may be written to the log before the object graph reachable from it is recoverable. This can occur when there is a cycle in a newly stable object graph: the record for one object in the cycle must be written to the log first, at which point the other objects in the cycle are not yet recoverable. The complication arises for a transaction aborted by a crash, if the transaction locks an object for which a base-update record was written. Recovery cannot apply the undo actions in the base-update record unless the objects reachable from it are in the heap; it needs a way to discover if this is true. Checking membership in the LS will not work, because recoverability of an object and its sub-graph does not imply membership in the LS.

To solve the problem, the tracker writes an additional record to the log, called the *base-update-complete record*, when it finishes tracking the graph reachable from a base-update record. The base-update-complete record contains the address of the object for which the base-update record was written. After a crash, recovery applies the undo actions recorded in a base-update record only if there is a corresponding base-update-complete record in the log. The presence of the base-update-complete record in the stable log guarantees that the object graph reachable from the base-update record is recoverable. If the base-update-complete record is not in the stable log, then the object for which the base-update record was written cannot be reachable so the undo actions do not have to be applied.

Figure 5.2 shows an object graph and a log that illustrate the order in which records are written to the log. Suppose object A is stable. An update action of a transaction makes the cycle containing B and C reachable from A. Suppose some other transaction has a lock on B. The tracker logs a base-update record for B and a base-commit record for C. Then it writes the base-update-complete-record for B. Finally the update action logs its update record. Object B cannot be reachable after a crash unless the update record for

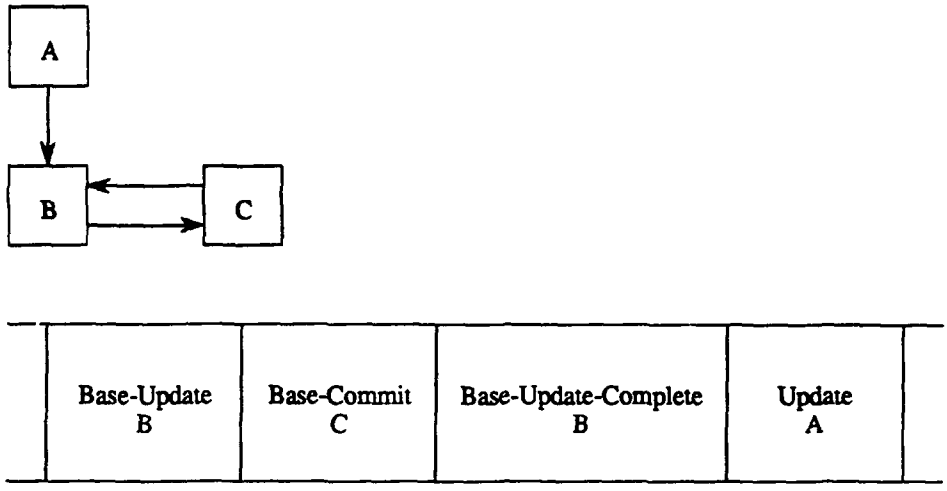


Figure 5.2: Use of Base-Update-Complete Record

A is in the log. The base-update-complete record for B precedes the update record, so if the base-update-complete record is absent after a crash (because it did not make it to the stable log), then B could not be reachable.

The Tracker

The procedure for tracking newly stable objects takes two arguments. The first argument is the object whose value should be tracked for newly stable objects. The second argument is the set of objects that has already been visited by the traversal; it permits the tracking of cycles in the graph.

The procedure has a local variable called the Newly Accessible Object Set or NAOS. It uses the NAOS to take a snapshot of the roots of the sub-graphs that need to be searched by the tracker. Without it, a concurrent update action could interfere with the tracker.

Here is the code for the stability tracker, **track_newly_stable**(*o*, *visited*).

1. If *o* is in the LS, return.
2. In an atomic step, if *o* is not in AS, then
   (a) If an active transaction holds a write lock on *o*, log a base-update record for *o* and insert the record into the active transaction's update chain; otherwise, log a base-commit record for *o*.
   (b) Put *o* in the AS.

3. In an atomic step construct the NAOS, i.e., for each object p referenced directly by o or by o's undo information where p is not in *visited* and p is not in the LS, insert p in the NAOS.

4. For each object p in the NAOS invoke **track_newly_stable**(p, *visited* $\bigcup \{o\}$).

5. In an atomic step, insert o in the LS.

6. If a base-update record was written for o in step 2a, write a base-update-complete record for o to the log.

The stability tracker is the only piece of code in the run-time recovery system that inserts objects in the AS and the LS. It is easy to check that it maintains Invariant 5.3 for the LS and Invariant 5.2 for the AS.

For simplicity of presentation, the stability tracker is described above as a recursive procedure performing a depth-first traversal. For efficiency, it can be rewritten as an iterative procedure that traverses the graph in some other order. Other transformations could also increase its efficiency:

- Only one visited set is required for a top-level invocation of the tracker and its recursions. The visited set can be kept small by deleting entries as they enter the LS.

- In step 3, the tracker could differentiate between objects that are in the AS but not the LS, and objects that are not in the AS. It could put the former in another set called the NYLS, or Not Yet in Logged Set, and the latter in the NAOS. Some other tracker may already be processing the objects in the NYLS and inserting them in the LS. Thus, in step 4 the tracker could invoke itself recursively on all of the objects in the NAOS before invoking itself on the objects in the NYLS.

- Instead of writing a separate base-update-complete record for each object for which a base-update record is written, the trackers can keep a global set that holds objects for which a base-update record has been written to the log and for which tracking is complete. The top-level invocations of the tracker write this set to the log incrementally: when a top-level tracker completes, it writes the whole set to the log in one base-update-complete record, and empties the set in one atomic step.

5.1.4 Update Action

An update action on a stable or newly stable object updates the object, calls the stability tracker to process the newly stable objects reachable from the updated part of the object, and writes an update record to the log. The tracker must complete its work before the

update record is logged in order to maintain Invariant 5.1 for the AS and Correctness Condition 5.2.[2]

Here is an outline for an update action on object o by transaction t.

1. Obtain a write lock on o.
2. In an atomic step, if o is not in the AS:
   (a) Update o.
   (b) Store undo information for o in volatile memory.
   (c) Insert o in the MVOS for transaction t (the set of volatile objects modified by t).
   (d) Return.
3. Pin the page of o.
4. Update o.
5. Construct the update record. Track newly stable objects reachable from the update record, i.e., for each object p referenced directly by redo or undo information in the update record where p is not in the LS, invoke **track_newly_stable**(p, $\{o\}$).
6. Spool the update record describing the modification to the log buffer, linking the record into transaction t's chain of update records.
7. The update action is over and the transaction can continue.
8. Unpin the page of o after the update record is in the stable part of the log.

Step 5 invokes the tracker on objects that are not in the LS. We could avoid this call to the tracker for an object that is in the AS: since some other update must have made the object newly stable, the tracker for the other update eventually will put the object in the LS, and this update could just wait until the object is in the LS. However, the other tracker may be working on some other part of the graph so we insert the call to minimize the delay.

In general the objects that an update action makes reachable and the outcome of an update action cannot be known until the update makes its modification. However, tracking for the update action must complete before the action logs its update record. Therefore, the

[2]Technically, updating an object before tracking the newly stable objects for the update does not quite work with the way we have described the invariants. (Between the update and the completion of the tracking some objects may be reachable from the stable roots but not in the AS.) However, a simple trick fixes the problem. Since an object is pinned and locked for the duration of an update action, the update to the object is not visible until the action completes — the pinning prevents the update from reaching disk, and the lock prevents other transactions from accessing the object. Thus, we can think of the update actually occurring after the tracking completes.

update action invokes the tracker after it updates the object, but before it logs the update record. Yet, tracking may cause memory allocation and trigger a garbage collection. Since a flip must occur in a low-level action quiescent state, the update action and the tracker must be implemented so that the garbage collector can finish both before the flip. We say more about coordination between update actions and the garbage collector in Section 5.2.4.

5.1.5 Example

Figure 5.3 shows how the tracking algorithm works by example. It shows a sequence of six snapshots of the tracker, the log, and an object graph in virtual memory. At the beginning, objects A and B are stable, and objects C and D are volatile. In the first snapshot, an update action has modified object A, inserting a pointer to object C. The update invokes **track_newly_stable**(C,{A}). In the second snapshot, A's tracker has added C to the AS, logged a base-commit record for C, and computed an NAOS. In the third snapshot, a second update action has modified object B and inserted a pointer to object C. The second update action has invoked **track_newly_stable**(C,{B}). The fourth snapshot shows that A's tracker has invoked **track_newly_stable**(D,{A,C}), which has added D to the AS and logged a base-commit record for D. By the fifth snapshot, A's tracker has completed, popped its stack, and inserted D and then C in the LS; that allowed the update action that modified A to write its update record. In the last snapshot, B's tracker has noticed that D is in the LS and it completed; the update action that modified B has logged its update record.

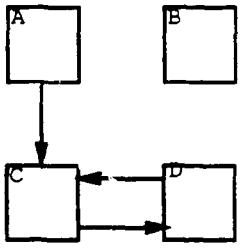
The third snapshot illustrates the bug in Oki's algorithm [38]; his algorithm would have noticed that C is in the AS, would not have invoked B's tracker, and would have written an update record to the log before the graph rooted at C is recoverable. (We described the bug in Oki's algorithm in Section 5.1.3.)

5.1.6 Representation of the AS and the LS

The AS and the LS could be represented as explicit sets using a hash table, but two other representations allow a faster membership check as well as simpler deletion and recovery. The two representations are: (1) two bits per object and (2) a single bit per object and an explicit set.

Using two bits per object, one bit would indicate membership in the AS and the second

Snapshot 1



Log

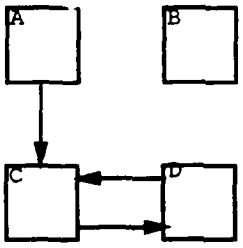
AS: A, B

LS: A, B

Tracker State for Update to A

O: C Visited: A NAOS:	

Snapshot 2



Log

Base-Commit C	

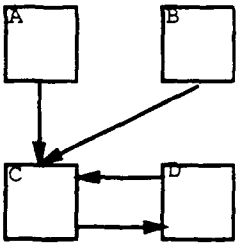
AS: A, B, C

LS: A, B

Tracker State for Update to A

O: C Visited: A NAOS: D	

Snapshot 3



Log

Base-Commit C	

AS: A, B, C

LS: A, B

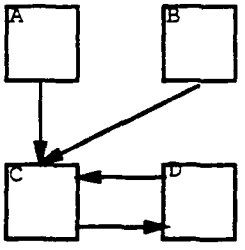
Tracker State for Update to A

O: C Visited: A NAOS: D	

Tracker State for Update to B

O: C Visited: B NAOS:	

Snapshot 4



Log

Base-Commit C	Base-Commit D	

AS: A, B, C, D

LS: A, B

Tracker State for Update to A

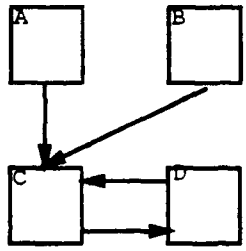
O: C Visited: A NAOS: D	O: D Visited: A, C NAOS:

Tracker State for Update to B

O: C Visited: B NAOS:	

Figure 5.3: Concurrent Tracking

Snapshot 5



Log

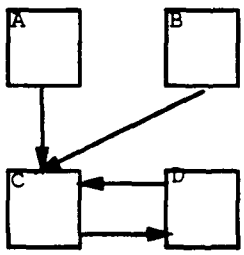
Base-Commit C	Base-Commit D	Update A	

AS: A, B, C, D
LS: A, B, D, C

Tracker State for Update to B

O: C Visited: B NAOS:	

Snapshot 6



Log

Base-Commit C	Base-Commit D	Update A	Update B	

AS: A, B, C, D
LS: A, B, D, C

Figure 5.3: Concurrent Tracking (cont)

bit would indicate membership in the LS. These bits are placed in the object descriptor, which also holds the type and the size of the object. The current Argus system uses a similar representation for its AS [38].

To understand the single bit and explicit set representation, notice that there are only three states possible with respect to an object's membership in the AS and the LS: (1) not in either, (2) in the AS only, or (3) in both. The bit in each object would show whether the object is in both sets, while the explicit set would contain entries for objects that are in the AS but not the LS. Thus, an object is in the AS if its bit is set or if it is in the explicit set.

If two bits are available in object descriptors, it is clearly the best choice. If only one bit can be spared in the object descriptor, the single bit and an explicit set is a good alternative. The explicit set would be very small, so that the costs for storage and the cost for lookup could be very cheap. Below, when knowledge of the representation is necessary for us to describe the solution to a problem, i.e., recovery of the AS and LS, and deletion from the sets, we assume the representation using two bits.

5.1.7 Transaction Abort

To abort a transaction, the recovery system must undo its updates both to stable objects and to volatile objects. It finds the updates to stable objects by following the back pointers in the transaction's chain of update records. It determines which volatile objects have been updated by checking the transaction's MVOS, which is the set of volatile objects that the transaction has updated.

There is an invariant on logs that constrains them to be well-formed. In a well-formed log the update and base-update records for a transaction precede the compensation records for the transaction. The abort algorithm depends on this constraint to make sure that it undoes each update of an aborting transaction exactly once. To ensure the constraint, abort undoes all of a transaction's updates to volatile objects before it undoes the updates to stable and newly stable objects. Some of the objects in a transaction's MVOS may no longer be volatile; these are objects for which base-update records have been written to the log. Abort uses the AS to determine whether an object in the MVOS is volatile: a volatile object is not in the AS.

Using logical undo in its most general form, an undo action for a stable object could make a new object reachable from a stable root. Thus, in general, undo actions must also check for newly stable objects. However, in most cases the implementor of an undo action can code it to avoid the check for newly stable objects.

We outline the undo action for an update to a stable or newly stable object, and how it fits into transaction abort.

Details of an Undo Action for a Stable Object

Here is an outline for an undo action on a stable or newly stable object o by transaction t.

1. If this undo action is part of the undo pass of recovery, the record to be undone is a base-update record, and o is an object for which tracking is not complete in the stable log, return without doing any work.
2. Translate addresses in the undo information to to-space addresses.
3. Pin the page of o.
4. Carry out the undo action.
5. Construct the compensation log record for the undo action. Track newly stable objects reachable from the redo information in the CLR; i.e., for each object p referenced directly by the redo information where p is not in the LS, **track_newly_stable**(p, $\{o\}$).
6. Spool the CLR describing the modification to the log buffer. Chain the CLR into the transaction's chain of update records by making it point to the next update to be undone.
7. The undo action is over and the abort can continue.
8. Unpin the page of o after the CLR is in the stable part of the log.

Details of Transaction Abort

Here are the details of abort for transaction t.

1. For each object, o, in t's MVOS:
   (a) In an atomic step, if o is not in the AS, then use the undo information in volatile memory to undo the effects of t on o.
2. Trace backward through t's chain of update and base-update records in the log. Undo the updates as described above.

3. Release t's write locks.

4. Write an abort record to the log.

5.1.8 Recovery from System Crash

With several minor exceptions, recovery from a system crash remains unchanged. The exceptions are the processing of base-update and base-update-complete records, and recovery of the AS and the LS.

Base-update and Base-update-complete Records

Base-update records are handled as base-commit records during redo; i.e., redo uses the information in the record to recover the state of the object and the allocation pointer. The analysis pass also uses the base-update records together with the base-update-complete records to construct the *Base Update Set*. This is the set of objects that have base-update records in the log for which tracking is incomplete. A checkpoint occurs in a low-level action-quiescent state, so at the checkpoint there are no invocations of the tracker active and tracking is complete for every object. Thus, the analysis pass initializes the Base Update Set to empty when it processes the checkpoint record. Then, each time it encounters a base-update record for an object, it adds that object to the set; each time it encounters a base-update-complete record for an object, it deletes the object from the set. The undo pass uses the Base Update Set to determine whether the undo information from a base-update record should be processed (in step 1 of the undo action in Section 5.1.7).

Recovery of the AS and the LS

After a crash we need to recover the AS and the LS such that the Invariants 5.1, 5.2, and 5.3 are not violated. In order to reduce the overhead of maintaining the AS and the LS, the tracker does not follow the redo protocol when it inserts the object into one of the sets. As a result, there are two problems in recovering the sets: (1) an object may have a bit set in its descriptor for the AS or the LS, even though the object does not belong in the set, and (2) an object may belong in the AS or the LS, but the appropriate bit is not set in its

descriptor. We discuss the first problem for the AS and the LS together, and the second problem individually for each set.

Because of the first problem an object may be in the AS even though its base-commit record is not in the stable log, or in the LS even though not every object reachable from it is recoverable. However, these objects cannot be reachable from a stable root after a crash; so neither Invariant 5.2 for the AS nor Invariant 5.3 for the LS is violated. Also, no transaction will be able to access these objects after recovery completes.

To recover the AS and to reestablish Invariant 5.1 for the AS after a crash, redo must ensure that every object commit reachable from a stable root is in the AS. Therefore, the redo pass reinserts an object in the AS whenever it recovers the object's initial value from a base-commit or base-update record. If redo does not have to restore an object's initial value from the log (because the object's base-commit or base-update record precedes min-LSN), that object already has its AS bit set on disk, since the tracker sets the bit and spools the base-commit or base-update record in a single atomic step. This procedure to recover the AS ensures that every object that belongs in the AS gets re-inserted in the AS. It may also insert objects that do not belong, but these objects cannot be reachable from a stable root.

To recover the LS, redo reinserts an object in the LS when it recovers an object from a base-commit or base-update record; it also reinserts an object when it processes a base-update-complete record. This procedure may add objects to the LS that do not belong in the LS, but these objects cannot be reachable from a stable root. Also, it may fail to insert an object in the LS that really could be in the LS, i.e., the object is in the AS and every object reachable from it is in the AS. However, redo's failure to put an object in the LS is not incorrect, it just reduces the efficiency of the tracker. If an object belongs in the LS, the tracker will reinsert the object the first time it tracks the sub-graph to which the object belongs. In practice we expect that the number of objects that do not get re-inserted in the LS will be low.

5.1.9 Deleting Objects from the AS and the LS

When an object is no longer accessible from any root, stable or volatile, the garbage collector frees its storage, which automatically deletes it from the AS and the LS. However, some

objects in the AS and the LS may still be reachable from a volatile root, but no longer reachable from a stable root. Deleting these objects is strictly an efficiency problem, in that future modifications to the objects will be more expensive than necessary.

A stop-the-world algorithm to detect precisely which objects should no longer be in the AS and the LS is simple: first, trace from the stable roots and mark the reachable objects; second, trace from the volatile roots and delete objects from the AS that were not marked in the first step. Designing an incremental algorithm is more complex. Since we do not expect many stable objects to remain reachable from a volatile root after they have become unreachable from a stable root, we do not present a solution here. If the problem turns out to be troublesome in a real system, we will need to find an acceptable solution.

5.1.10 Discussion

Our algorithms are complex because we require concurrency and logical undo. We discussed the impact of concurrency in Section 5.1.3; we discuss the impact of logical undo here.

In a recovery system with logical undo, an update action needs to make sure that base-commit and base-update records for all of the objects that it makes stable are in the log before the update record. Otherwise, after a crash it may be impossible to recover the object graph rooted at the updated object before carrying out the logical undo action. This means that the latency for an update action depends on the amount of state the update makes stable, which is not unreasonable.

However, a recovery system with physical undo can decrease the latency of update. In that case, base-commit and base-update records for the objects that an update action makes stable do not have to be in the log until the transaction commits. If there is a crash before the transaction commits, undo will still work, since it restores the object to the exact physical state that it was in before the update. Thus, an update action can spool work for a stability tracker thread, and return as soon as the update record is recorded in the log.

Besides decreasing update latency there are two other benefits to a recovery system that uses physical undo: (1) if a transaction updates a single object many times and waits to invoke the tracker, it can save tracking work; and (2) tracking work can be scheduled for a convenient time (e.g., during communication delays or page faults).

The tracking and update algorithms would also be simpler for a recovery system using physical undo: the tracking algorithm would not need to write base-update-complete records; no tracking would be required for the undo information in an update record; and no tracking would ever be required for an undo action.

Even in a recovery system that allows the use of logical undo, not every update action takes advantage of it. For many actions physical undo is still the cheapest and simplest approach. Such a system may be able to achieve some of the advantages cited above for tracking in a system with physical undo.

5.2 Dividing the Heap

Atomic garbage collection ensures the continued recoverability of stable objects in a stable heap despite garbage collection. However, it is significantly more expensive than normal garbage collection since it needs to follow the write-ahead log protocol whenever it copies an object or scans a page. Many of the objects in a stable heap are volatile and do not require the functionality of atomic garbage collection. This section presents a method for organizing and collecting a stable heap that avoids these extra costs when reclaiming storage for volatile objects.

We divide the heap into two areas whose address spaces are disjoint: the volatile area and the stable area. All objects are created in the volatile area. Objects that become stable are copied to the stable area at an appropriate time. The disk storage for the volatile area is discarded after a crash, and recovery is performed using the log and the disk storage for the stable area. The volatile area can be collected independently of the stable area and without requiring the collection to be atomic; its collector can be stop-the-world, incremental or generational, but for the purposes of presentation we assume that it is stop-the-world. The stable area is collected using an atomic incremental garbage collector; its collector can also be generational.

Besides cheap garbage collection for volatile objects, there are two other motivations for dividing the heap. First, dividing the heap allows garbage collection work to be concentrated in the volatile area, where there is likely to be proportionally more garbage than in the stable area. We hypothesize that volatile objects are more likely to be short lived than

stable objects. Usually a volatile object is associated with local state for a procedure or a transaction and is short-lived, whereas a stable object outlives the transaction that created it and is longer lived. This is similar to the principle behind generational garbage collection, which says that more recently created or younger objects are more likely to become garbage than older objects.

Second, dividing the heap reduces the amount of disk storage that needs to be allocated permanently to a stable heap. Only the disk storage associated with the stable area needs to survive crashes.

Deciding when to move a newly stable object to the stable area is the key issue in the design of algorithms for dividing the heap. We begin by examining this issue and choosing one alternative; based on our choice we show how to move newly stable objects to the stable area, how to garbage collect the volatile area, and how to garbage collect the stable area. Then we present a second approach based on another alternative.

5.2.1 When to Move Objects to the Stable Area

A newly stable object can be moved from the volatile area to the stable area as soon as it becomes reachable from a stable or newly stable object, or at a subsequent volatile garbage collection. Moving objects at an intermediate point incurs the disadvantages of both schemes. We compare the two possibilities below.

Recovery is more expensive, both at run-time and after a crash, when the volatile collector moves the newly stable objects. At run-time, information is written to the log twice for each newly stable object: once by the recovery system when the object becomes newly stable; the second time by the volatile collector when it moves the object to the stable area. After a crash, recovery must process both sets of records, so it is also more expensive. In contrast, if the recovery system moves an object as soon as it detects that the object is newly stable, it writes to the log once; the volatile collector does not have to write additional information.

However, accessing volatile objects is cheaper when the system moves a newly stable object at a volatile garbage collection after it becomes reachable. If moved immediately upon becoming reachable, a forwarding pointer has to be left in the object in the volatile

area, since other volatile objects may point to the object. Then every access to a volatile object has to check for a forwarding pointer. This could be expensive on stock hardware. If moved at a garbage collection of the volatile area, no special check for a forwarding pointer is required for normal run-time access to the object, since the collector takes care of fixing up the other pointers to the object.

Fortunately the cost of checking the forwarding pointer can be avoided for most objects. Objects can be atomic or non-atomic. Only atomic objects can ever be reachable from a stable root; accesses to non-atomic objects never have to be checked for forwarding pointers. Furthermore, atomic objects can be immutable or mutable. A program cannot distinguish between two copies of an immutable object, so a newly stable immutable object can be copied to the stable area without inserting a forwarding pointer in its volatile copy and immutable objects can always be accessed without checking for forwarding pointers. Copying immutable objects this way could lead to two performance problems: (1) increased storage, since an immutable object may be copied multiple times to the stable area if it becomes reachable through more than one path, and (2) slower equality tests, since two copies of an immutable object will have to be compared more frequently. The first problem can be avoided by keeping track of the immutable objects that have been moved to the stable area. Using this information, we can ensure that each object is moved at most once, and the next garbage collection discards the volatile copies of immutable objects that have been moved. This solution also minimizes the effect of the second problem.

Only atomic mutable objects require forwarding pointers when they are copied to the stable area. However, these objects are already the most expensive to access. A read operation on an atomic mutable object must acquire a read lock before accessing the object, and a write operation must acquire a write lock. The extra time to check for a forwarding pointer compared to the time to obtain a read or write lock is very small; in fact, checking for the forwarding pointer can be combined with obtaining a lock, so the extra cost is negligible. The check for the forwarding pointer will be most noticeable for an equality test; without acquiring any locks, equal checks whether two pointers name the same object.

Since the algorithms for dividing the heap are much simpler when newly stable objects are moved to the stable area as soon as they are detected and the extra costs for checking

forwarding pointers should be small, we choose this alternative. We outline another solution based on moving newly stable objects during volatile garbage collection in Section 5.2.6.

5.2.2 Moving Objects to the Stable Area

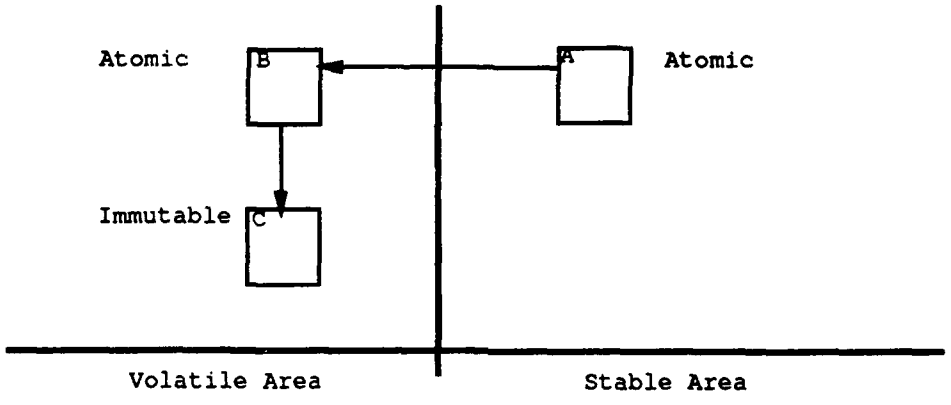
When an update action modifies an object in the stable area, it invokes the tracker to find the objects it made newly stable and make them recoverable. We modify the concurrent tracker so that it also copies the newly stable objects from the volatile to the stable area. When the tracker copies an atomic mutable object, it leaves a forwarding pointer in the volatile copy of the object. When the tracker copies an immutable object, it makes an entry for the object in the *Newly Stable Immutable Map* (NSIM). An entry in the NSIM maps the volatile address of an immutable object to the address of its copy in the stable area. The tracker uses the forwarding pointers in atomic objects and the NSIM to avoid making more than one copy of an object in the stable area.

Figure 5.4 shows the changes to the stable heap made by an update that makes an object newly stable. The update modifies object A, which is in the stable area, to point to a mutable atomic object B, which is in the volatile area. Object B points to object C, which is immutable. When the tracker copies B, it inserts a forwarding pointer in its volatile copy. When the tracker copies C, it inserts an entry in the NSIM that maps from the volatile address of C to the stable address of its copy.

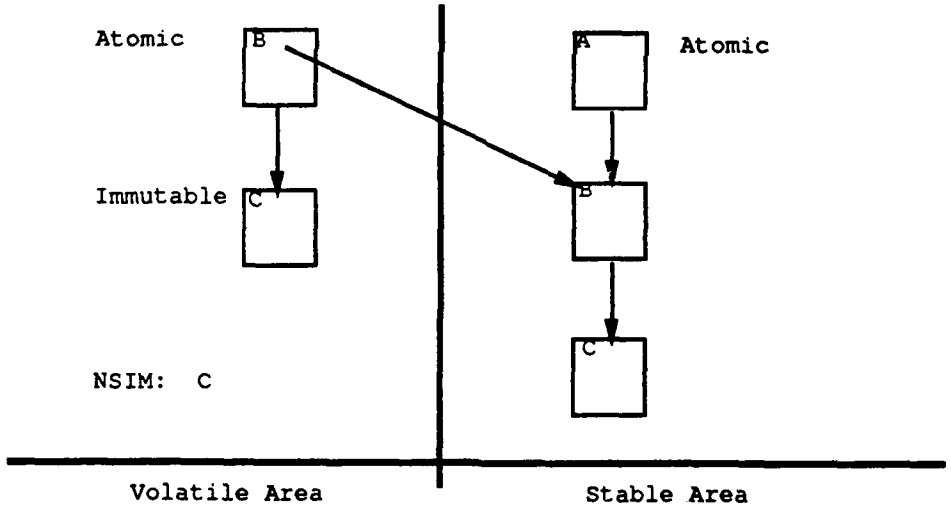
Below we present an overview of the modifications to the tracker. Then we show how to copy a newly stable object, how to scan a newly stable object for volatile pointers, and the exact modifications to the tracking routine and update actions. Finally, we briefly discuss a simpler but less concurrent tracking algorithm.

Overview of the Modified Tracker

The original tracker uses two sets, the AS and the LS, to find and keep track of the newly stable objects. It ensures three invariants on the sets: (1) if an object is reachable from the stable roots, then it is in the AS; (2) if an object is in the AS and it is reachable from a stable root, then it is recoverable from the log; and (3) if an object is in the LS and it is reachable from a stable root, then the objects reachable from it are in the AS. The modified



5.4.a: Update Modifies A



5.4.b: Tracker Moves Objects

Figure 5.4: Dividing the Heap: First Method

tracker continues to maintain these invariants, but it also maintains an additional invariant that governs the movement of newly stable objects to the stable area: if an object is in the AS, then it and the objects directly reachable from it, are in the stable area. This new invariant ensures that an object is in the stable area and all of its pointers point to other objects in the stable area before the tracker writes a base-commit or base-update record for it to the log. This simplifies recovery because it enables the tracker and the rest of the recovery system to name all objects in the log using stable addresses.

Using the new tracker, an object can be in one of four states: (1) in the volatile area, (2) in the stable area, but not in the AS, (3) in the stable area and the AS, but not the LS, and (4) in the stable area, AS, and LS. An object begins in the volatile area. When an update action causes the object to become newly stable, the tracker copies it to the stable area. Then the tracker scans the stable copy of the object, copies other objects directly reachable from the object to the stable area, and translates the volatile addresses in the object to stable addresses. Next, in an atomic step, the tracker logs a base-commit or base-update record for the object and inserts the object in the AS. Finally, the tracker invokes itself recursively on the object to insert the objects reachable from it into the AS; when all of these objects are in the AS, the tracker inserts the original object in the LS. Thus, the order in which the tracker copies objects to the stable area is alternatively breadth-first and depth-first: it is breadth-first when it scans an object and copies all of the objects directly reachable from the object; it is depth-first when it chooses the next object to track.

We also change the indicator for stability; previously, membership in the AS was the first indication that an object may be stable or newly stable. Now the tracker copies an object to the stable area before it inserts the object in the AS; thus, location in the stable area indicates that an object may be stable or newly stable. Therefore, an update action checks for location in the stable area to determine whether an object might be stable.

Copying an Object to the Stable Area

To copy objects, the tracker needs to be able to distinguish between atomic mutable and immutable objects. A bit in the object's descriptor can be used for this purpose.

When the tracker copies an atomic mutable object from the volatile area to the stable

area, it leaves a forwarding pointer in the volatile copy. The forwarding pointer overwrites a fixed location in an object; that location could be the descriptor. To distinguish the normal contents of the overwritten cell from the forwarding pointer requires a bit in the object descriptor.

To avoid copying an immutable object more than once to the stable area, the tracker maintains a *Newly Stable Immutable Map* (NSIM). The NSIM contains an entry for each immutable object that was copied from the volatile area to the stable area since the last garbage collection of the volatile area. The entry maps the object's volatile address to its stable address.

The procedure for copying an object takes an address of an object o in the volatile area as an argument and returns the address of the object in the stable area. Execution of the copy procedure occurs as an atomic step. Here is a detailed description of **copy**(o):

1. If o has a forwarding pointer in it, return the forwarding pointer.
2. If o is immutable and there is an entry for it in the NSIM, return the stable address in the entry.
3. Allocate space in the stable area for the object and copy the object to that space.
4. If o is immutable, then insert an entry mapping it to its new stable address in the NSIM. Otherwise, put a forwarding pointer in o to its stable copy.
5. Return the stable address.

Scanning an Object in the Stable Area

Here is the procedure, **scan**(o), for scanning a newly copied object o in the stable area for pointers into the volatile area.

1. For each address a in o, in an atomic step, if a is a pointer to the volatile area replace it in o by **copy**(a).

Checking if a is a pointer to the volatile area and replacing it by **copy**(a) occurs as an atomic step so that concurrent trackers will not interfere with each other. The scan occurs before the object is recoverable, i.e., before its base-commit or base-update record is written to the log; therefore the scan does not have to follow the redo protocol.

The New Concurrent Tracker

The new concurrent tracking algorithm replaces the first step of the old algorithm by two steps that copy and scan newly stable objects. Here is the code for the stability tracker, **track_newly_stable**(o, *visited*). (We repeat the whole algorithm to avoid confusion.)

1. If o is a volatile address, invoke **copy**(o) to copy the object it references to the stable area and set o equal to **copy**(o).
2. If o is not in the AS, then
   (a) Invoke **scan**(o).
   (b) For each volatile address p in the undo information for o, in an atomic step, replace p by **copy**(p).
   (c) In an atomic step, if o is not in the AS:
      i. If any active transaction holds a write lock on o, log a base-update record for o and insert the record into the transaction's update chain; otherwise, log a base-commit record for o.
      ii. Put o in the AS.
3. In an atomic step construct the NAOS, i.e., for each object p referenced directly by o or by o's undo information where p is not in *visited* and p is not in the LS, insert p in the NAOS.
4. For each object p in the NAOS invoke **track_newly_stable**(p, *visited* $\bigcup \{o\}$).
5. In an atomic step, insert o in the LS.
6. If a base-update record was written for o in step 2(c)i, write a base-update-complete record for o to the log.

Modifying Update and Undo Actions

Updates to objects in the stable area follow the write-ahead log protocol and call the tracker to check for newly stable objects. However, as we noted earlier, an object in the stable area may not yet be in the AS, so its tracker may still be active. Thus, an update to the object may interfere with the tracker because the update may reinsert volatile addresses in parts of the object that the tracker has scanned. This could lead to a violation of the invariant that governs the movement of newly stable objects to the stable area. Therefore, an update action to an object in the stable area waits until the object is in the AS before it proceeds. Alternatively, in case the tracker that copied the object is working on some other part of the graph, update could call its own invocation of the tracker to put the object in the AS.

At the completion of an update to an object in the stable area, there must be no pointers to the volatile area in the object. Otherwise, the invariant governing the movement of objects to the stable area would be violated. Therefore, the update action scans the part of the object it updated; it translates the volatile addresses to stable addresses and tracks newly stable objects.

Below we present the details of the update action. Again, to avoid confusion we present the full algorithm. The changes from the algorithm in Section 5.1.4 were discussed above: (1) checking location in the stable or the volatile area to determine whether the object is stable or volatile; (2) waiting for an object in the stable area to enter the AS; and (3) scanning the updated part of the object for pointers into the volatile area.

Here is an outline for an update action on object o by transaction t.

1. In an atomic step, if o is in the volatile area and there is no forwarding pointer in o:
   (a) Obtain a write lock on o.
   (b) Update o.
   (c) Store undo information for o in volatile memory.
   (d) Insert o in the MVOS for transaction t.
   (e) Return.
2. If o is in the volatile area, replace o by the stable object referenced by o's forwarding pointer.
3. Obtain a write lock on o.
4. Wait until o is in the AS.
5. Pin the page of o.
6. Update o.
7. Scan the updated part of o translating volatile addresses to stable addresses and tracking newly stable objects, i.e., for each address p in the updated part of o:
   (a) If p is in the volatile area, replace it by **copy**(p).
   (b) If p is not in the LS, invoke **track_newly_stable**(p,{o}).
8. Construct the update record. Translate volatile addresses in the undo information to stable addresses and track newly stable objects, i.e., for each address p in the undo information (including o):
   (a) If p is in the volatile area, replace it by copy(p).

(b) If p is not in the LS, invoke **track_newly_stable**(p,{o}).

9. Spool the update record describing the modification to the log buffer, linking the record into transaction t's chain of update records.
10. The update action is over and the transaction can continue.
11. Unpin the page of o after the update record is in the stable part of the log.

The preceding changes also apply to undo actions: an undo action scans the part of the object it modifies; it translates the volatile addresses to stable addresses and tracks newly stable objects.

Simple Tracking Algorithm

A simple but less concurrent tracking algorithm that uses one set, the Accessibility Set, and processes the whole object graph rooted at a newly stable object in a single atomic step was mentioned briefly in Section 5.1.3. This simple algorithm is easily adapted to support the copying of newly stable objects to the stable area. The AS does not have to be represented explicitly by a bit in the object descriptor; rather, an object is in the AS if it is in the stable area. The tracker does not have to use a stack or any special data structure to copy an object graph rooted at a newly stable object to the stable area; instead, the tracker copies the object graph to a contiguous piece of memory in the stable area so that its stack is implicitly threaded in the partially copied object graph (as in copying collection). Unfortunately, this simple algorithm impedes concurrency when a large object graph becomes stable.

5.2.3 Garbage Collection of the Volatile Area

For a garbage collection of the volatile area to be correct, the collector must be aware of all of the roots for collection; these include the normal roots and any pointers from the stable area into the volatile area. There can be pointers from the stable area to the volatile area only if the stability tracker is in the middle of tracking a newly stable object graph, or if an update action has not finished scanning the part of the object it updated. Because the tracking algorithm itself uses dynamic storage structures, it might be impossible to delay garbage collection until a tracking-quiescent and update-quiescent state (i.e., a state

in which there are no pointers from the stable to volatile areas). Thus, the garbage collector for the volatile area must be able to deal with these pointers.

Two solutions are possible: (1) trackers and update actions leave information for the collector that says which parts of the stable area might contain volatile pointers, and the collector searches for the pointers, or (2) the garbage collector finishes updates and tracking that were in progress as its first step. Since the collector runs alone, it can finish tracking without using any additional storage by using the simple tracking algorithm mentioned above. We assume the second solution in our description of the volatile collector below.

The volatile collector collects the volatile copies of newly stable immutable objects: for each entry in the NSIM, it inserts a forwarding pointer in the volatile copy of the object to point to the stable copy. Then the collector deletes all entries from the NSIM since they are no longer needed. When the collector runs, it uses the forwarding pointers to update pointers to the volatile copy of an immutable object to point to the stable copy.

To summarize, the collector for the volatile area:

1. suspends transactions,
2. completes updates and the tracking of newly stable objects,
3. inserts forwarding pointers in the volatile copies of newly stable immutable objects using the information in the NSIM,
4. flips, i.e., turns to-space into from-space and allocates a new to-space,
5. collects the volatile area,
6. and resumes transactions.

5.2.4 Garbage Collection of the Stable Area

An atomic incremental garbage collector collects the stable area. Because the system does not keep track of pointers to the stable area from the volatile area, every flip of the atomic collector must be accompanied by a collection of the volatile area. When the volatile collector scans objects in the volatile to-space during this collection, it may encounter pointers to objects in the stable from-space. It copies such objects to the stable to-space using the atomic copying step.

A flip of the atomic garbage collector must occur in an update-quiescent state. However, the flip may be triggered because the tracker runs out of space in the stable area for

newly stable objects, and may occur in the middle of an update action. Therefore the implementation must arrange data structures such that the collector can finish both tracking and updates immediately before the flip. To finish tracking the collector needs memory in the stable area to hold newly stable objects. It can use the new stable to-space for this purpose. This may cause two anomalies that do not normally occur: (1) objects in the stable from-space may have pointers to objects in the stable to-space that are not forwarding pointers, and (2) base-commit, base-update, and update records may have a combination of stable from-space and stable to-space addresses. As long as the collectors can tell the difference between a forwarding pointer and a regular pointer the first anomaly should not cause any problems; e.g., the forwarding pointer overwrites the descriptor. The second anomaly should also not be a problem: if the newly stable object in the stable from-space that contains pointers to objects in the stable to-space is reachable from a stable root, the stable collector will copy it to to-space and scan it. By the time it is scanned, copy records for objects it references will be in the log.

When the volatile collector runs during a stable flip, it may encounter the forwarding pointers inserted into objects in the volatile area by the stability tracker or the collector itself during the preprocessing of the NSIM. These pointers may point to the stable from-space. If the volatile collector encounters such a pointer, and the stable object has not yet been copied to the stable to-space, it copies the object using the atomic copying step and updates the forwarding pointer in the volatile object to point directly to the stable to-space copy.

5.2.5 Discussion

One of the major goals in the design of our atomic garbage collector was to keep the pauses for collection short. For that reason we designed an incremental collector based on the algorithm of Ellis. Using Ellis's algorithm there are two kinds of pauses: (1) to scan a page, and (2) to flip. The time to flip for the atomic incremental collector of the stable area depends on the choice of collection techniques for the volatile area. In this chapter we have assumed that a stop-the-world collector collects the volatile area; thus, the time during which transactions are suspended during a flip of the atomic collector depends on the time

taken to collect the volatile area. If the number of volatile objects is small, this may be reasonable; otherwise, some other approach is necessary. Below we sketch three possible approaches: (1) incremental collection in the volatile area, (2) generational collection in the volatile area, and (3) keeping track of the pointers from the volatile area to the stable area. Deciding among the approaches requires further work; measurements of programs running on top of our prototype implementation (described in Chapter 7) may provide a basis for a choice.

Incremental Volatile Collection

Incremental volatile collection requires a minor change to the discussion of the stable collector in Section 5.2.4. Instead of a full volatile garbage collection occurring during a stable flip, only the steps leading up to the volatile flip (i.e., the completion of tracking and preprocessing of the NSIM) and the volatile flip itself occur during the stable flip. Also, the volatile collection started by the volatile flip must complete before the stable collection; this ensures that no pointers from the volatile area to the stable area are overlooked.

This approach appears to be simple, but there are two problems with it. First, a volatile incremental collector based on the read barrier of Ellis may not be very incremental. The rate of access to the volatile area is high; thus, depending on locality, many traps may occur just after the volatile flip and delay transactions as much as a stop-the-world collector.

Second, there may be hash tables in the volatile area whose hashing function is based on object address. We have been quietly assuming that the programmer does not use these tables at the application level. However, these tables are quite useful in the implementation, e.g., for keeping track of read and write locks or marshaling arguments for remote procedure call. If the number of entries in such a table is small, its contents can be copied to to-space and rehashed in a single pause the first time it is accessed after the volatile flip without the delay being noticeable. If such tables are large, the time to copy and rehash them may significantly delay transactions.

Generational Volatile Collection

The volatile collector could be generational. Assuming two generations, the system keeps track of the locations of all pointers from the older generation to the younger generation and the locations of all pointers from the older generation to the stable area. (Any of the techniques used by existing generational collectors for keeping track of the pointers is applicable: remembered sets [47], card marking [44], and virtual memory [43].) This enables the collector to collect the younger generation without collecting the older generation, and to collect the stable area without collecting the older generation. Thus, a flip of the atomic collector in the stable area is delayed only by the time to collect the younger generation, and not by the time to collect the entire volatile area. This discussion generalizes to more than two generations in the obvious way.

As with an incremental collector, a generational collector must also solve the hash table problem.

Keeping Track of Pointers

Instead of collecting the volatile area to find the pointers from the volatile area to the stable area, the system could keep track of the locations of the pointers. Then a flip of the atomic collector of the stable area could occur without a corresponding flip or collection in the volatile area. The cost of this approach depends on the number of pointers from the volatile area to the stable area and the rate of modification to the volatile area.

Other Issues

The stable collector could also be generational. A possible benefit of generational garbage collection might be cheap collection of garbage that is generated when a transaction aborts after it makes objects newly stable. Assume two generations in the stable area and no generations in the volatile area. Collecting the younger generation independently of the older generation requires keeping track of the locations of pointers from the older generation to the younger generation. Since the collector must be atomic, this set of locations would also have to be recoverable. The set of locations is recoverable from the log because all updates are recorded in the log; however, additional information may have to be recorded

in checkpoint records to avoid gathering information from the prefix of the log before a checkpoint. We say more about generational collection for the stable area in our conclusions in Chapter 9.

Using static analysis a compiler for a language such as Argus may be able to deduce that an object will be made stable by the transaction that creates it. For these cases the compiler could emit code to create these objects directly in the stable area and avoid the cost of moving them to the stable area soon after creation.

5.2.6 Moving Newly Stable Objects at the Next Volatile Garbage Collection

Instead of the stability tracker moving an object to the stable area as soon as it becomes reachable from the stable root, the garbage collector could move it at a later collection of the volatile area. Though this alternative is more complex, we favored it initially because it avoids the cost of checking forwarding pointers and we used it in our implementation. Subsequently, we realized that many of the checks for forwarding pointers can be avoided when objects are moved as soon as they are detected.

This second alternative is also interesting for another reason. We have been assuming that almost all accesses to atomic objects are preceded by the acquisition of a lock. However, an optimizer for a language such as Argus may be able to eliminate code to acquire a lock when static analysis shows that the transaction already possesses the lock. In that case the cost of accessing atomic objects would decrease and the cost of checking for forwarding pointers would be more noticeable.

We present the second alternative below. First we discuss how it moves newly stable objects to the stable area. Then we discuss changes to the collectors of the volatile and stable areas.

Moving Objects to the Stable Area

During a volatile garbage collection the collector moves all objects that became stable or newly stable since the beginning of the last volatile collection to the stable area. The roots for determining which objects to copy are in the stable area and in undo information for stable and newly stable objects.

To find the roots in the stable area without scanning the entire stable area, the collector uses the *Stable to Volatile Set* (SVS). The SVS, analogous to the Remembered Set of generation scavenging [47], is a data structure maintained by the recovery system that contains an entry for each object in the stable area that has a pointer into the volatile area. When an update action modifies a stable object and inserts a pointer to an object in the volatile area in it, the action inserts the stable object into the SVS. The update action already has to do this check to track the newly stable objects, so keeping track of the SVS does not incur much additional run-time cost.

The recovery system maintains the undo information for stable and newly stable objects in the log. Specifically, the relevant undo information can be found in the update and base-update records that were written to the log since the penultimate flip but before the last flip on behalf of transactions that are still active. The log records for a transaction are back-chained in the log, so the collector can find the records for the active transactions and scan their undo information without looking at every log record written since the penultimate flip. Reading the log during volatile collection is expensive, and is one of the main disadvantages of this scheme. There are ways to avoid reading the log by duplicating undo information in the volatile area, but they increase the cost of modifying stable and newly stable objects.

Given the roots, the volatile collector determines which objects to move from the volatile area to the stable area in the usual way. Namely, it scans each object in the SVS and the undo roots for stable and newly stable objects, looking for pointers to the volatile area; each such pointer refers to an object that must be moved. As usual, when the collector moves an object it inserts a forwarding pointer in the volatile copy of the object to avoid moving it more than once. The collector recursively scans each object it moves. The collector continues scanning until all the moved objects have been scanned and no more objects need to be moved.

When the volatile collector moves newly stable objects it modifies the stable area in three ways: (1) it copies newly stable objects to the stable area; (2) it scans objects in the SVS and translates volatile addresses in them to stable addresses; and (3) it scans pages of the stable area to which newly stable objects have been copied. We show how to make each

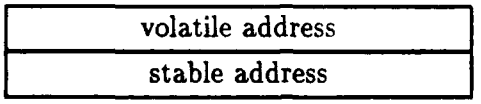


Figure 5.5: Format of V2scopy Record

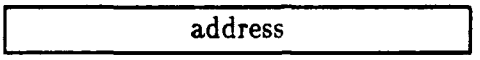


Figure 5.6: Format of S4vscan Record

of these modifications repeatable by recovery.

Copying When the volatile collector moves a newly stable object to the stable area, it writes a *v2scopy record* to the log. Figure 5.5 pictures the record. The v2scopy record contains the old volatile from-space address of the object and its new stable address. The collector need not follow the redo protocol when it moves the object since the newly allocated memory for it in the stable area is fresh, and recovery does not depend on its previous contents.

Recovery uses the addresses in v2scopy records to translate volatile addresses to stable addresses. When the base-commit or base-update record for a newly stable object is written to the log, the object is in the volatile area. It does not yet have a stable address, so the recovery system uses its volatile address to identify it in log records until it is moved to the stable area. Thereafter recovery uses its stable address in log records.

Recovery also uses the addresses in v2scopy for its redo pass. Redo reconstructs a volatile copy of the object from its base-commit or base-update record, applies any applicable update records, and then re-copies the object to the stable area when it processes the v2scopy record.

Scanning an Object in the SVS When the volatile garbage collector scans an object in the SVS, it modifies an existing object in the stable area; it must follow the redo protocol. It pins the page of the object in the stable area, scans the object and updates its volatile from-space pointers to stable to-space pointers, copying objects as necessary, and it spools a *s4vscan record* containing the address of the scanned object to the log. Figure 5.6 shows the s4vscan record. When the s4vscan record is in the stable log, the page is unpinned.

Scanning Pages of Stable To-space Containing Newly Stable Objects To scan a page of stable to-space to which it has copied newly stable objects, the volatile garbage collector uses the atomic scan step except that it also scans the page for volatile pointers.

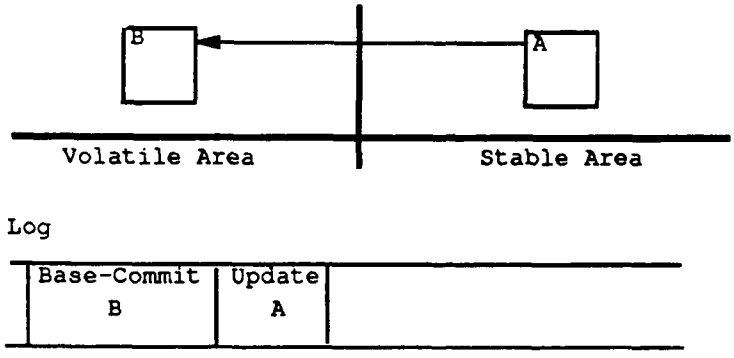
Collecting the Volatile Area

The volatile garbage collector moves the objects that have become stable since the start of the last volatile collection to the stable area, and collects the volatile area. Figure 5.7 shows the changes that the volatile collector makes to the heap and the records it writes to the log as it moves newly stable objects to the stable area. Before the volatile flip, an update to object A, which is in the stable area, inserted a pointer to object B, which is in the volatile area. The update called the tracker, which logged a base-commit record for B and inserted A into the SVS. When the garbage collector runs, it notices A in the SVS, so it scans A for pointers into the volatile area. It copies B to the stable area and logs a v2scopy record for B. When the collector has completed its scan of A, it logs a s4vscan record for A.

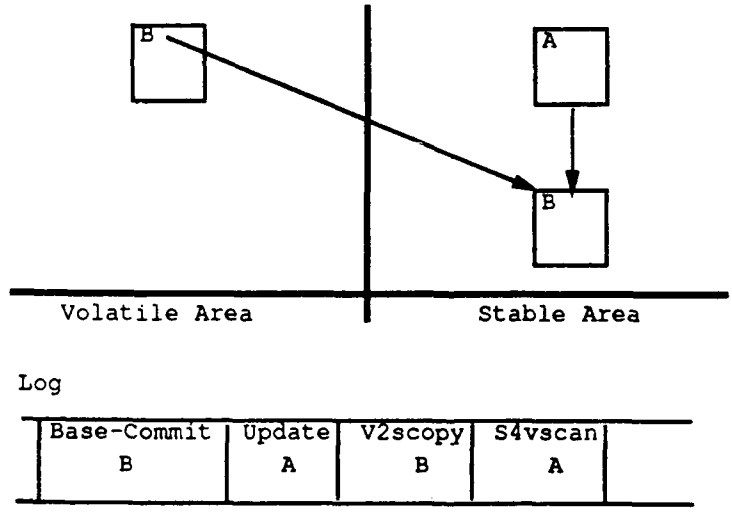
Below we show how to synchronize the volatile collector with the recovery system and how to translate volatile addresses in undo roots; then we present the full details for the stop-the-world volatile collector.

Synchronization With the Recovery System Our algorithms must ensure that newly stable objects and the object graphs rooted at newly stable objects are recoverable. If the tracker traverses a newly stable object graph and writes base-commit and base-update records for its objects within a volatile garbage collection interval, the object graph is recoverable. The redo pass can reconstruct each object in the graph and the interconnections between the objects, because the volatile addresses denoting these interconnections in the records are unique within the garbage collection interval. However, if the tracker writes some records for the object graph before a volatile flip and some after the volatile flip, the object graph is not recoverable without extra translation information.

Our algorithm must also ensure that updates recorded in update records are recoverable. If the tracking of newly stable objects and the update record to a stable object occur within a single volatile garbage collection interval, the update is recoverable. The redo pass can



5.7.a: Before Flip



5.7.b: Collector Moves Objects

Figure 5.7: Dividing the Heap: Second Method

reconstruct the newly stable object graphs and the updated object's connections to them. If tracking for a newly stable object completes on one side of a flip and the update record is written after a flip, then the volatile collector may copy the newly stable object to the volatile to-space. In that case the update record's connections would not be recoverable without information to translate the volatile from-space address of the newly stable object (which is the address recorded in the object's base-commit record) to the object's volatile to-space address (which is the address recorded in the update record).

Since we already need to ensure that a stable flip occurs in an update-quiescent and tracking-quiescent state, we also require that a volatile flip occur in the same kind of state; this obviates the need to write extra translation information to the log. Thus, when a volatile garbage collection is triggered, it finishes any tracking and updates that are in progress before it flips.

Normally the tracker needs to allocate temporary storage to run, but when a volatile garbage collection is triggered there is no more storage available in from-space. Therefore the garbage collector allocates temporary storage for the tracker in to-space. Also, since there are no concurrent trackers running during a collection, the collector can use the simple tracking algorithm.

We describe an alternative synchronization scheme in Chapter 7 when we describe our implementation. It writes translation information to the log instead of bringing the system to a tracking quiescent state. Because of constraints related to our prototype, this was a simpler solution.

Translating Volatile Addresses in Recovery Information Transaction abort may need to undo an action recorded in an update or base-update record that was written during a previous volatile garbage collection interval. The undo information in these records may contain volatile addresses. In order to use the undo information, transaction abort must first translate the volatile addresses to addresses in the current stable to-space. There is enough information in the v2scopy and copy records in the log to do the translation, but finding the right records may be slow.

This problem is similar to the undo address translation problem for atomic garbage

collection, and we propose a similar solution. The recovery system maintains information in the heap to speed translation of the volatile addresses; it also writes the information to the log so that fast translation is possible after a crash. Specifically, for each root in the undo information that the volatile collector uses for moving newly stable objects to the stable area, it makes an entry in the UTT for normal translation and an entry in the *Volatile Undo Translation Record* (VUTR) for translation after a crash. The VUTR maps from a <volatile flip number, volatile undo root> to a current address. Before the volatile garbage collector reclaims from-space, it writes the VUTR to the log. The VUTR is analogous to UTR; whereas the UTR maps stable addresses from past stable garbage collection intervals to current stable addresses and is written at the end of a stable collection, the VUTR maps volatile addresses to current stable addresses and is written at the end of a volatile collection.

Stop-the-World Volatile Collection A garbage collection of the volatile area begins by copying all of the newly stable objects in the volatile area to the stable area. Then the collector traces from the normal volatile roots and copies the objects accessible from them to the volatile to-space.

Here are the full details.

1. Complete update actions and tracking.
2. Scan each object in the SVS for volatile pointers: following the redo protocol, copy newly stable objects to the stable area as necessary, and spool a s4vscan record to the log.
3. For each undo root for an active transaction in an update or base-update record written to the log since the last volatile flip:
   (a) Make an entry in the UTT and the VUTR.
   (b) If the referenced object does not have a forwarding pointer, copy it to the stable area, inserting a forwarding pointer, and writing a v2scopy record for it.
4. Scan the newly stable objects moved to the stable area for volatile pointers using the atomic scan step, copying newly stable objects to the stable area as necessary. This step could also be carried out incrementally during earlier steps, whenever an object is moved to the stable area.
5. Collect the volatile area normally.
6. Write the VUTR to the log.

7. Free volatile from-space.

Garbage Collection of the Stable Area

An atomic incremental collector collects the stable area. Because we do not keep track of pointers from the volatile area to the stable area, a volatile garbage collection must occur as part of the stable flip. When the volatile collector encounters a pointer to an object in the stable from-space, it copies the object to the stable to-space using the atomic copying step.

Chapter 6

Specification and Correctness Argument

To increase our understanding of the interaction between garbage collection and recovery and to help derive invariants for our algorithms, we wrote a formal specification for a stable heap and modeled our implementation formally. The formal specification and implementation are written in the language Spec, which was developed for a course at M.I.T. in computer systems [49].

We begin with a brief description of Spec. Then we present the specification and discuss how it models the desired behavior of a stable heap. Finally, we show an implementation for the specification that models our algorithms for recovery and garbage collection, and demonstrate its correctness by exhibiting the abstraction function and invariants on its internal state. Neither the specification nor the implementation models every aspect of a stable heap precisely; both simplify details so as not to obscure the main points. Most importantly, the implementation does not divide the heap or track newly stable objects. Appendix A outlines the proof that the implementation conforms to the specification.

6.1 Spec

We present a brief overview of Spec here; for a full description see the language definition [49].

Spec has *expressions*, *atomic statements*, and *non-atomic statements*. An expression is a function from states to results. Both kinds of statements change the state of memory;

they are non-deterministic and their execution may be guarded by a boolean expression. An *atomic statement* executes atomically with respect to other statements. Operationally, the thread of execution non-deterministically chooses a path through an atomic statement; if it encounters a false guard on that path, it backtracks and tries another path. An atomic statement is said to *fail* if the thread encounters a false guard on all possible execution paths through it, in which case, it has no effect. Semantically, we can view an atomic statement as a relation on states that relates an input state to all possible output states corresponding to successful paths through the statement.

For a *non-atomic statement* there can be any number of execution histories (sequences of state transitions) from a given state. A non-atomic statement *fails* if it cannot make its first transition. A non-atomic statement may succeed (make its first transition), but later get stuck with all guards false. There is no backtracking for a non-atomic statement, so a non-atomic statement may get stuck in one history even though there are other histories in which a different choice would allow it to run to completion. Non-atomic statements are interesting in the presence of concurrency: if one thread gets stuck, it may become unstuck later because of state changes made by another thread.

A program in Spec consists of a collection of modules; each module declares some types, variables, and routines. There are three kinds of routines: *functions*, *atomic procedures*, and *general procedures*. A function is free of side-effects and can be invoked by an expression. An atomic procedure describes an atomic state transition and can be invoked by an atomic statement. A general procedure may contain non-atomic statements and can be invoked only by non-atomic statements. A module also has a body. In its initial state, a module starts with a single thread executing its body; later, it may fork other threads.

A module may have a crash operation that models the effect of a crash on its state and then performs a recovery procedure. A program has a special crash thread. At any time the crash thread may cause a crash, which stops and destroys the other program threads. Then it starts a new thread for each module with a crash operation, and invokes the crash operation in that thread. A crash obeys atomicity brackets for the threads it destroys, so that intermediate states of an atomic statement are never visible.

6.1.1 Constructs

We present the key constructs of Spec here; other constructs in the code should be obvious.

The construct *predicate* => *statement* denotes a guarded command. If *predicate* evaluates to true, *statement* can be executed. *Predicate* is evaluated atomically.

The construct <<*statement*>> denotes an atomic statement.

The construct *statement1* [] *statement2* denotes a non-deterministic choice. Either *statement1* or *statement2* is executed.

The outcome of *statement1* [*] *statement2* is the same as *statement1* unless *statement1* fails in which case its outcome is the same as *statement2*.

The construct DO *statement* OD denotes a loop that repeats *statement* until it fails.

The construct VAR *variable_declaration_list* | *statement* introduces new variables. A declaration may initialize a new variable explicitly; otherwise, a new variable is initialized to an arbitrary value of its type.

The HAVOC statement produces an arbitrary outcome from any state; it is used to specify arbitrary behavior when a precondition is not satisfied.

The construct *statement1* ; *statement2* denotes sequential composition: first *statement1* is executed, then *statement2* is executed.

The construct *function*{*expression1* -> *expression2*} is a function constructor. It denotes a new function having the same value as *function* except that it maps *expression1* to *expression2*. If *expression2* is left out, the new function is undefined at *expression1*. If *expression1* is *, then the new function is defined to be *expression2* everywhere.

The construct *function_type*{} denotes the function of type *function_type* that is undefined everywhere.

The construct (EXISTS *declaration_list* | *predicate*) evaluates to true if *predicate* is true for at least one binding of the declared identifiers.

The construct {*declaration_list* | *predicate* | *expression*} is a set constructor. It denotes a set whose type is SET[T], where T is the type of *expression*. The set contains x if and only if (EXISTS *declaration_list* | *predicate* /\ x = *expression*).

Logical operators include \/, /\, and ==>, which stand for *or*, *and*, and *implication* respectively. Logical expressions are evaluated left to right; however, only the parts necessary

```
VAR
   active: SET[TID] := {}              % active transactions
   committed: SET[TID] := {}           % committed transactions
   aborted : SET[TID] := {}            % aborted transactions

APROC start_t() -> TID =
  << VAR t:TID | ~(t IN (active + committed + aborted)) =>
        active := active ++ t;
        RET t >>
```

Figure 6.1: Start_t Procedure of the Stable Heap Specification

to compute the value of the expression are actually evaluated. For example, if the result of *expression1* is true then the result of *expression1* \/ expression2 is true and *expression2* is not evaluated.

Many operators in Spec are overloaded. The operator ++ denotes the concatenation of a single element to the end of a sequence or the addition of a single element to a set. The operator -- denotes the removal of a single element from a set. The operator + denotes the addition of two integers, the concatenation of two sequences, or the union of two sets. The operator * denotes the multiplication of two integers or the intersection of two sets. The operator <= denotes less than or equal for integers or the subset relation for sets.

Other non-intuitive operators are # for not equal and ! for defined. The boolean expression *function*!*expression* evaluates to true if *function* is defined at *expression*.

6.1.2 Examples

Several examples, taken from the specification of the stable heap, illustrate the features of Spec.

`Start_t` is an atomic procedure (its code is in Figure 6.1); its body consists of an atomic statement. `Start_t` has no arguments, but it returns a result of type `TID`. The `VAR` statement introduces the variable `t` of type `TID`. According to the operational interpretation of atomic statements, `start_t` non-deterministically chooses a `TID` and assigns it to `t`. Then the guarded command checks if `t` is in the set determined by the union of **active**, **committed**, and **aborted**. If it is not, the command succeeds: it inserts `t` in **active** and returns `t` to

```
TYPE
   value = UNION[scalar, OID]
   state = SEQ[value]

FUNC oids(s: state) -> SET[OID] =
    RET {i:INT | 0 <= i /\ i <= size(s) /\ s(i) IS OID | s(i)}
```

Figure 6.2: Oids Function of the Stable Heap Specification

the invoker of **start_t**. If **t** is in the union of the sets, backtracking occurs and the **VAR** statement non-deterministically chooses a new value for **t**. Backtracking continues until the atomic statement succeeds or fails. According to the semantic interpretation of atomic statements, **start_t** chooses a **TID** that is not in the union of **active**, **committed**, and **aborted**; inserts the **TID** in **active**, and returns the **TID** to the invoker.

Oids is a function that illustrates the set constructor; its code is in Figure 6.2. Its argument is a **state**, **s**, and it returns a set of **OID**s (object identifiers). A **state** is a sequence of **value**. (Sequence is a built-in type of Spec; its operations include the usual operations on sequences, e.g., head, tail, is_empty, last, and remove_last.) A **value** can be a **scalar** or an **OID**. The function returns the set of **OID** contained in the **state**, **s**.

6.2 Specification

The specification models the key features of a stable heap: part of the state of a heap is stable, part volatile; stability is based on reachability from stable roots; and computations on atomic objects run as atomic transactions. The specification does not model non-atomic state and computations on that state.

In the specification, computations on the stable heap run as atomic transactions. A transaction begins with a **start_t** operation. Then it can execute any number of **update**, **read**, and **allocate** operations; it completes with a **commit** or **abort** operation. Many transactions may run concurrently, but a given transaction executes one operation at a time. The specification models transactions using two-phase read/write locking and intentions lists.

Each object in the stable heap has a unique identifier or OID. An object map, the **os**,

maps an object's OID to its last committed state, i.e., its state just after the last transaction to modify it committed. The state of an object may contain the OIDs of other objects.

Another map, the **tos**, holds the uncommitted state, represented as intentions lists for active transactions. When a transaction commits, its intentions are installed in the **os**. When a transaction aborts, its intentions list is discarded.

The stable heap has two sets of distinguished objects: the stable roots (**SR**) and the volatile roots (**VR**). The stable roots survive crashes; the volatile ones do not. The stable root set is initialized once during the lifetime of a heap by the **init_heap** operation. Before the **init_heap** operation is called, any number of transactions may run and build up the state to be made stable by **init_heap**. Since one of the root objects could be a mutable set of objects, the above model is as general as one in which transactions dynamically insert and delete objects from the stable root set.

The volatile root set represents both global volatile roots and non-transactional state such as registers and stacks; its contents change dynamically as transactions invoke operations on the stable heap. To allow an invoking transaction to use its results, an **allocate** operation inserts its newly allocated object into the **VR**; similarly, a **read** operation inserts objects directly accessible from the object it reads into the **VR**. The **delete_VR** operation deletes objects from the **VR**; it models steps such as popping a stack or clearing a register.

To model the notion of accessibility in a heap, a transaction may access an object through a **read** or **update** operation only if the object is a volatile or stable root. To access an object that is reachable from a root, but not currently in one of the root sets, a transaction invokes **read** operations starting with the root from which the object is accessible. It traverses the graph invoking **read** operations until the desired object is in the volatile root set and can be accessed directly.

Since the additions and deletions from **VR** are non-transactional, a client of a stable heap must restrict its transactions to access the **VR** in a way that ensures their serializability. For example, each transaction may have its own stack to which it has exclusive access. Also, several transactions may run in the same local scope and share a subset of the **VR** that is visible in the scope.

A crash empties the **VR**. This models loss of state in the real world. Real systems run

code to re-initialize their volatile state after a crash. For example, an Argus guardian runs the code in its recovery section before beginning to process transactions. We do not specify recovery code; the first transactions to run after the crash re-initialize the VR.

The allocate operation initializes the object it creates. To ensure the serializability of transactions, it must also obtain a lock on the object on behalf of the invoking transaction. It obtains a read lock rather than a write lock to permit greater concurrency. This corresponds to a view that allocation does not really create a new object: it "finds" an object that no transaction has seen before in the universe of objects, and returns its identifier to the caller. The "found" object has the desired state. Allocate can obtain a read lock because the transaction has seen the state of the new object, but not modified it. Allocation operations work the same way in Argus.

6.2.1 Stable Heap Specification

Here is the specification of the stable heap.

```
MODULE StableHeap [                          % specifies stable heap
   scalar,                                   % scalars
   OID,                                      % object identifiers
   TID                                       % transaction identifiers
   ] =

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                                  Types                                      %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
TYPE
   value = UNION[scalar, OID]
   state = SEQ[value]
   OS = OID -> state
   TOS = TID -> OS

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                        Internal State (variables)                           %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
VAR
   os: OS := OS{}                            % object states
   tos: TOS := TOS{}                         % transaction object states
   RL: OID -> SET[TID] := RL{* -> {}}        % read locks
   WL: OID -> SET[TID] := WL{* -> {}}        % write locks
   SR: SET[OID] := {}                        % stable roots
```

```
   VR: SET[OID] := {}                    % volatile roots
   o_allocated: SET[OID] := {}           % allocated OIDs
   active: SET[TID] := {}                % active transactions
   committed: SET[TID] := {}             % committed transactions
   aborted : SET[TID] := {}              % aborted transactions
   init_flag: BOOL := false              % true if stable roots initialized
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                          Interface Operations                              %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
APROC init_heap(roots:SET[OID]) =
  % May be called one time over lifetime of heap
  << init_flag \/ ~(roots <= VR) => HAVOC [*]
     SR := roots;
     init_flag := true >>

APROC start_t() -> TID =
  << VAR t:TID | ~(t IN (active + committed + aborted)) =>
         active := active ++ t;
         RET t >>

APROC read(t:TID, o:OID) -> state =
  << ~((o IN (SR + VR)) /\ t IN active) => HAVOC [*]
     read_lock(t, o);
     VAR s:state |
         BEGIN WL(o) = {t} => s := tos(t)(o) [*] s := os(o) END;
         VR := VR + oids(s);
         RET s >>

APROC write(t:TID, o:OID, s:state) =
  << ~((o IN (SR + VR)) /\ state_from_roots(s) /\ t IN active) => HAVOC [*]
     write_lock(t,o);
     tos[t] := tos(t){o -> s} >>

APROC allocate(t:TID, s:state) -> OID =
  << ~(state_from_roots(s) /\ t IN active) => HAVOC [*]
     VAR o:OID | ~(o IN o_allocated) =>
         o_allocated := o_allocated ++ o;
         read_lock(t,o);
         os[o] := s;
         VR := VR ++ o;
         RET o >>

APROC commit(t:TID) =
  << ~(t IN active) => HAVOC [*]
     DO
```

```
      VAR o:OID | os(o) # tos(t)(o)  => os[o] := tos(t)(o)
    OD;
    committed := committed ++ t;
    active := active -- t;
    tos[t] := OS{};
    release_locks(t) >>

APROC abort(t:TID) =
  << ~(t IN active) => HAVOC [*]
    aborted := aborted ++ t;
    active := active -- t;
    tos[t] := OS{};
    release_locks(t) >>

APROC crash() =
  << aborted := aborted + active;
    VR := {};
    active := {};
    tos := TOS{};
    RL := RL{* -> {}};
    WL := WL{* -> {}} >>

APROC delete_from_VR(o:OID) =
  << VR := VR -- o >>

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                          Internal Operations                              %%
%%%%%%%%%%%%%%%%%%%% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
APROC write_lock(t:TID, o:OID) =
  % blocks till t obtains a write lock
  << WL(o) <= {t} /\ RL(o) <= {t} => WL[o] := {t} >>

APROC read_lock(t:TID, o:OID) =
  % blocks till t obtains a read lock
  << WL(o) <= {t} => RL[o] := RL(o) ++ t >>

APROC release_locks(t:TID) =
  % releases t's locks
  << DO
      VAR o:OID | t IN WL(o) => WL[o] := WL(o) -- t
      [] VAR o:OID | t IN RL(o) => RL[o] := RL(o) -- t
    OD >>

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                                Functions                                  %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
FUNC oids(s: state) -> SET[OID] =
    RET {i:INT | 0 <= i /\ i <= size(s) /\ s(i) IS OID | s(i)}

FUNC state_from_roots(s:state) -> BOOL =
    RET (oids(s) <= VR + SR)

END StableHeap
```

6.3 Implementation

The implementation models recovery based on repeating history and atomic incremental garbage collection. For each operation of the specification there is a corresponding operation of the implementation. The implementation also has background threads that perform benevolent side-effects on its state; the atomic incremental garbage collector runs in one of these threads.

The size of atomic steps in the implementation has been chosen to be as large as possible and still show the main issues; this avoids cluttering the invariants unnecessarily. Each specification operation is implemented as a single atomic step, except for abort, which is a sequence of atomic steps so that there may be more than one partially undone transaction at a crash.

We discuss the representations for main memory, disk, and the log, how we model garbage collection and crashes, and the abstraction function and invariants for our implementation. Then we present the implementation itself.

6.3.1 Main Memory and Disk

The implementation represents disk storage by the stable base (**sb**) and the main memory buffer pool by the volatile cache (**vc**). Both the stable base and the volatile cache are mappings from addresses to object values. Unlike real systems, the volatile cache only contains dirty objects; i.e., an object is in the volatile cache only if its value is different than the object at the same address in the stable base. Permitting other objects in the volatile cache is just an optimization.

An address is an ordered pair where the first part of the pair is a space number; this models the spaces of a copying garbage collector. An object value contains the object's

state and a forwarding pointer for copying gc. To save space our algorithm of Chapter 3 overwrites a cell of an object with its forwarding pointer. Since this is an optimization we do not model it directly in our implementation.

An object value takes up one address regardless of its size. We do not try to model logical objects that are larger than a physical page or pages that contain multiple objects; these issues are tangential to the exploration of the interactions between garbage collection and recovery.

A background process flushes objects in the volatile cache to the stable base subject to the write-ahead log protocol. This process corresponds to the buffer manager of a database system. Thus, the stable base may contain uncommitted values for objects or may be missing some committed values for objects. For this reason the implementation maintains information in a log, and the stable base together with the log allow the stable state of the heap to be recovered after a crash.

6.3.2 Log

The log consists of two parts: the stable log (`sl`), which models the part of the log on disk, and the volatile log (`vl`), which models the part of the log in main memory buffers. Another background process periodically chooses a prefix of the volatile log and concatenates it to the stable log. This models log buffers being written to disk as they fill. Commit also concatenates the volatile log to the stable log to model a log force.

The log records in the implementation are a subset of those actually used by our algorithms; we leave out checkpoint, page-fetch and end-write records. These records are used for optimizations that reduce recovery work by allowing recovery to efficiently deduce a prefix of the log that it does not have to process. Instead, our implementation has another background process that truncates the stable log by (inefficiently) calculating a prefix not required by recovery.

Our implementation uses physical undo because it is simpler to express, but its recovery system based on repeating history is designed to support logical undo with little change. Supporting logical undo would require passing the `update` operation a function instead of passing it a new value for the object it updates. The function would compute a new value

for the object and also return an inverse function for use with logical undo.

6.3.3 Garbage Collection

The garbage collector runs as a background thread. Its steps are interleaved with the steps of operations run on behalf of transactions. At each of its steps the collector non-deterministically chooses to scan an unscanned object in to-space or to copy an undo root to to-space. As an optimization, our algorithms described in Section 4.2 choose to handle the undo roots last.

To-space is the space number of the current space; the **allocate** and **copy** operations insert it in the new addresses they assign. At every flip **to-space** is increased by one, instead of reusing a previous space number. However, we can show that the implementation does not need to depend on information in virtual memory at addresses prior to from-space (the space immediately before the current to-space), subject to certain conditions.

Scanned contains the set of to-space objects scanned by the collector. To maintain correct synchronization with the collector, a **read** or **update** operation may not access an object unless it is in **scanned**. This models the read barrier of Ellis [18].

6.3.4 Crashes

A crash discards the volatile log, the volatile cache, the volatile root set, and the scanned set; this models the information loss of a real crash. Recovery repeats history by redoing the stable log against the stable base, and then undoing active transactions.

6.3.5 Abstraction Function and Invariants

The implementation has two representations for the stable heap: the real representation described above and a mirror representation of the specification's state in history variables. The history variables are required to show that the implementation conforms to the specification because the implementation throws away information from the real representation. Because of the history variables the abstraction function is almost trivial (the only interesting part is for the tos); the first invariant relates the real representation to the history variables.

In the statement of the abstraction function, the invariants, and their supporting functions, we use the operator + to indicate redo, the operator - to indicate undo, and the operator ++ to indicate the caching relationship between the stable base and the volatile cache. The overloading of these operators is defined precisely in the comments on page 126.

The abstraction function is on page 126. It shows how the implementation represents the **tos** using the stable base and the log: the current value for an object locked by an active transaction is obtained by redoing the concatenation of the stable and volatile logs against the stable base.

The invariants start on page 127. The repeating history invariant mentioned in Section 2.2.3 does not appear directly in our list of invariants. It holds, but it is subsumed by the first and second invariants.

The first invariant shows how the implementation represents the **os** using the stable base and the log: if an object is reachable from a stable or a volatile root, then its committed value is obtained by repeating history and then undoing the transactions that are active in the concatenation of the stable and volatile logs. The invariant holds only for objects reachable from a root; other objects in the **os** cannot be accessed by transactions.

The next three invariants are useful to prove that the abstraction function and the first invariant hold. The second invariant states that the undo information in the volatile log cancels the redo information in the log subject to commit reachability from a root and certain other technical constraints relating to garbage collection. This invariant is important in order to show that the implementation does not lose the states of objects reachable from the stable root when a crash discards the volatile log.

The third invariant shows how the implementation uses virtual memory (the stable base and the volatile cache) to cache information in the log. It says that the virtual memory and the log agree on the current state of the stable heap: an object's value in virtual memory is the same as the value obtained for it by redoing first the stable log and then the volatile log against the stable base.

The fourth invariant says that if no transaction holds a write-lock on an object, then its current value in virtual memory is the same as its committed value; i.e., no undo is required for it.

The garbage collector maintains four invariants. First, the key invariant says that if an object is reachable from a stable or volatile root, it is in from-space or to-space. Second, if an object is in the volatile roots or the stable roots it is in to-space. The flip ensures this invariant by copying the root objects. Third, the address of an object in the undo roots for an active transaction can be translated using information in the log to a to-space or a from-space address. The garbage collector maintains this invariant by searching the log for undo roots and copying the objects reachable from them from from-space to to-space. Since the collector does not flip until all objects in to-space are scanned and all undo roots are in to-space, these three invariants ensure that the collector does not collect objects that are reachable from a stable, volatile or undo root. Fourth, if a transaction holds a read or write lock on an object, the object is in to-space. Some objects may be locked by an active transaction, but may no longer be reachable from a stable or volatile root. This invariant ensures that these objects stay in the heap so that the lock information remains consistent.

The last invariant ensures correct address allocation in the face of crashes: if a to-space address is allocated there is a base-commit or copy record in the log for the object at that address. Thus, by reading the log after a crash, recovery can determine which addresses were allocated before the crash so it will not reallocate them afterwards.

6.3.6 Stable Heap Implementation

Here is the implementation of the stable heap. The interface operations begin on page 129 and the background processes on page 131.

```
MODULE StableHeapImpl [                          % implements stable heap
   scalar,                                       % scalars
   TID,                                          % transaction identifiers
   address,                                      % addresses
   OID                                           % object identifiers
   ] =

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                                  Types                                      %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
TYPE
   va = RECORD[space:INT, address:address]       % space number differentiates
                                                 % from-space addresses from
                                                 % to-space addresses
```

```
    value = UNION[scalar, va]
    forw = UNION[va, NULL]
    rstate = SEQ[value]                          % real state of an object
    state = RECORD[forw: forw, rstate: rstate] % first cell of object may
                                                 % hold forwarding pointer
    store = va -> state

    update = RECORD[tid: TID, address: va,
                    new_state: rstate, old_state: rstate]
    clr = RECORD[tid: TID, address: va, new_state: rstate]
                                                 % compensation log record
    base_commit = RECORD[address: va, state: rstate]
                                                 % for newly allocated objects
    outcome = ENUM[commit, abort]
    tran_outcome = RECORD[outcome:outcome, tid:TID]
                                                 % transaction outcome
    copy = RECORD[from: va, to: va]
    scan = RECORD[address: va]
    log_rec = UNION[update, clr, base_commit, tran_outcome, copy, scan]
    log = SEQ[log_rec]

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                        Internal state (variables)                        %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
VAR
    vc: store := store{}                  % (volatile cache) main memory
    sb: store := store{}                  % (stable base) disk backing store
    to_space: INT := 1                    % number of current to-space

    vl: log := log{}                      % volatile part of log
    sl: log := log{}                      % part of log on stable storage

    SR: SET[va] := {}                     % stable roots
    VR: SET[va] := {}                     % volatile roots
    init_flag: BOOL := false              % true if stable roots initialized

    a_allocated: SET[va] := {}            % allocated addresses
    scanned: SET[va] := {}                % objects unlocked by gc
    RL: va -> SET[TID] := RL{* -> {}}     % read locks
    WL: va -> SET[TID] := WL{* -> {}}     % write locks
    active: SET[TID] := {}                % active transactions

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                            History Variables                             %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
TYPE
```

```
    svalue = StableHeapSpec[scalar, OID, TID].value  % UNION[scalar, OID]
    sstate = StableHeapSpec[scalar, OID, TID].state  % SEQ[svalue]
    OS = OID -> sstate
    OIDMAP = va -> OID

VAR
    os: OS := OS{}                          % object states
    o_allocated: SET[OID] := {}             % allocated OIDs
    oidmap: OIDMAP := OIDMAP{}              % maps virtual addresses to OIDs
    committed: SET[TID] := {}               % committed transactions
    aborted : SET[TID] := {}                % aborted transactions

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                          Abstraction Function                           %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% In the following:
% store1 ++ store2 == docache(store1, store2)
% store + log == dolog(store, log)
% store - log == undolog(store, log)

% Abstraction function is identity for os, o_allocated, active,
% committed, aborted and init_flag.  That leaves tos, WL, RL, SR and VR.
FUNC toTOS() -> (TID -> OID -> sstate) =
  RET (LAMBDA (t:TID) -> (OID -> sstate) =
         t IN active =>
             RET (LAMBDA (o:OID) -> sstate =
                    VAR a:va | o = oidmap(a) /\ {t} = WL(a) =>
                        RET state2sstate((sb + sl + vl)(a)) ))

FUNC toWL() -> (OID -> SET[TID]) =
  RET (LAMBDA (o:OID) -> SET[TID] =
         VAR a:va | o = oidmap(a) /\ a.space = to_space =>
             RET WL(a) * active
         [*] RET {} )

FUNC toRL() -> (OID -> SET[TID]) =
  RET (LAMBDA (o:OID) -> SET[TID] =
         VAR a:va | o = oidmap(a) /\ a.space = to_space =>
             RET RL(a) * active
         [*] RET {} )

FUNC toSR() -> SET[OID] =
  RET {o:OID | (EXISTS a:va | a IN SR /\ oidmap(a) = o) | o}

FUNC toVR() -> SET[OID] =
```

```
  RET {o:OID | (EXISTS a:va | a IN VR /\ oidmap(a) = o) | o}


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                                Invariants                                %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Invariant on os history variable.
% The precondition is required because a crash could occur after
% allocation, but before base_commit record is in the stable log.
% (1) accessible(a) ==>
%       os[oidmap(a)] = state2sstate((sb + sl + vl - (sl + vl))(a))

% Volatile undos cancel volatile redos for commit-accessible objects.
% Translation required because there could be a copy or scan record for
% ''a'' in vl, and vl appears only on one side of the equality.  (If
% translation were not required, the right side of the implication
% would read: (sb + sl - sl)(a) = (sb + sl + vl - (sl + vl))(a). )
% (2) commit_accessible(a) ==>
%       translate_state((sb + sl - sl)(a), sl + vl) =
%       translate_state((sb + sl + vl - (sl + vl))(translate(a, sl + vl)),
%                       sl + vl)

% Cache and log agree.
% (3) sb ++ vc = sb + sl + vl

% No write lock on an object means undo not necessary to recover its value
% (4) WL(a) = {} ==> (sb + sl + vl)(a) = (sb + sl + vl - (sl + vl))(a)

% Gc invariants.
% (5a) accessible(a) ==> (a.space = to_space \/ a.space = from_space)
% (5b) a IN SR \/ a IN VR ==> a.space = to_space
% (5c) a IN translate_set(undo_roots(sl + vl), sl + vl) ==>
%        (a.space = to_space \/ a.space = from_space)
% (5d) (EXISTS t | t IN WL(a) \/ t IN RL(a)) ==> a.space = to_space

% Correct address allocation.
% For this implementation, this invariant prevents trimming of log since
% last flip; a real implementation could trim by allocating in order of
% increasing addresses or recording which ranges of addresses have been
% allocated in a checkpoint record.
% (6) a IN a_allocated /\ a.space = to_space <==>
%       (EXISTS i |
%         ((log_rec$is_copy((sl + vl)(i)) /\ (sl + vl)(i).copy.to = a) \/
%          (log_rec$is_base_commit((sl + vl)(i)) /\
%           (sl + vl)(i).base_commit.address = a)))
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                Functions Required to State Invariants                  %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
FUNC docache(base:store ,cache: store) -> store =
  VAR a:va | cache ! a => RET docache(base{a -> cache(a)}, cache{a -> })
  [*] RET base

FUNC dolog(s:store, l:log) -> store =
% returns redo log, l applied to store, s
  is_empty(l) => RET s
  [*] RET dolog(dorec(s, l.head), l.tail)

FUNC undolog(s:store, l:log) -> store =
% returns store, s with incomplete transactions in log, l undone
  RET undoactive(s, remove_complete(l), l)

FUNC dorec(s:store, lr:log_rec) -> store =
  lr IS update => VAR ur:update := lr |
      RET s{ur.address -> state{forw := nil, rstate := ur.new_state}}
  [*] lr IS clr => VAR cr:clr := lr |
        RET s{cr.address -> state{forw := nil, rstate := cr.new_state}}
  [*] lr IS copy => VAR cr:copy := lr |
        RET s{(cr AS copy).to -> state{forw := nil,
                                       rstate := s(cr.from).rstate}}
             {cr.from -> state{forw := cr.to,
                               rstate := s(cr.from).rstate}}
  [*] lr IS scan => VAR sr:scan := lr |
        RET s{sr.address ->
               state{forw := nil,
                     rstate := redo_scan_state(s(sr.address).rstate)}}
  [*] lr IS base_commit => VAR br:base_commit := lr |
        RET s{br.address -> state{forw := nil, rstate := br.state}}
  [*] RET s

FUNC undoactive(s:store, ul:log ,l:log) -> store =
  % ul contains update records only
  is_empty(ul) => RET s
  [*] VAR ur:update := ul.last |
      RET undoactive(s{translate(ur.address, l) ->
                        state{forw := nil,
                              rstate := translate_state(ur.old_state, l)}},
                     ul.remove_last,
                     l)

FUNC remove_complete(l:log) -> log =
```

```
% Returns the log, constructed by removing from l all records except
% update records for active transactions in l for which there is no
% corresponding CLR.
  RET remove_undone(extract_active(l, l))

FUNC accessible(a: va) -> BOOL =
  RET (EXISTS b:va | (b IN (SR + VR)) /\ reachable(b,a))

FUNC reachable(a:va, b: va) -> BOOL =
  RET (a = b \/
      (EXISTS c:va | directly_reachable(c, (sb + sl - sl)(a), sl + vl) /\
                     reachable(c,b)) \/
      (EXISTS c:va | directly_reachable(c, (sb + sl + vl)(a), sl + vl) /\
                     reachable(c,b)))

FUNC commit_accessible(a: va) -> BOOL =
  RET (EXISTS b:va | (b IN (SR + VR)) /\ commit_reachable(b,a))

FUNC commit_reachable(a:va, b: va) -> BOOL =
  RET (a = b \/
      (EXISTS c:va | directly_reachable(c, (sb + sl - sl)(a), sl) /\
                     commit_reachable(c,b)))

FUNC directly_reachable(a:va, s:state, l:log) -> BOOL =
  RET (EXISTS sp:rstate, v:value, ss:rstate |
          s.rstate = sp ++ v + ss /\ v IS va /\ a = translate(v AS va, l))

% Code for translate, translate_state, extract_active,remove_undone,
% state2sstate, and undo_roots can be found with the functions
% starting on page 135.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                          Interface Operations                          %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
APROC init_heap(roots:SET[va]) =
  % May be called one time over lifetime of heap
  << init_flag \/ ~(roots <= VR) => HAVOC [*]
     SR := roots;
     init_flag := true;
     force_log() >>                    % to make sure SR is recoverable

APROC start_t() -> TID =
  << VAR t:TID | ~(t IN (active + committed + aborted)) =>
        active := active ++ t;
        RET t >>
```

```
APROC read(t:TID, a:va) -> rstate =
  << ~((a IN SR + VR) /\ t IN active) => HAVOC [*]
     a IN scanned =>
         read_lock(t, a);
         VR := VR + addresses(current_state(a).rstate);
         RET current_state(a).rstate >>

APROC write(t:TID, a:va, s:rstate) =
  << ~((a IN SR + VR) /\ state_from_roots(s) /\ t IN active) => HAVOC [*]
     a IN scanned =>
         write_lock(t,a);
         vc[a] := current_state(a);
         vl := vl ++
               update{tid := t,
                      address := a,
                      new_state := s,
                      old_state := vc(a).rstate};
         vc[a].rstate :=  s >>

APROC allocate(t:TID, s:rstate) -> va =
  << ~(state_from_roots(s) /\ t IN active) => HAVOC [*]
     VAR a:va, o:OID | ~(a IN a_allocated) /\ a.space = to_space /\
            ~(o IN o_allocated) =>
         a_allocated := a_allocated ++ a;
         o_allocated := o_allocated ++ o;
         oidmap[a] := o;
         os[o] := rstate2sstate(s);
         vc[a] := state{forw := nil, rstate := s};
         vl := vl ++ base_commit{address := a, state := s};
         read_lock(t,a);
         scanned := scanned ++ a;
         VR := VR ++ a >>

APROC commit(t:TID) =
  << ~(t IN active) => HAVOC [*]
     DO
       VAR a:va |
           WL(a) = {t} /\ os(oidmap(a)) # state2sstate(current_state(a)) =>
               os[oidmap(a)] := state2sstate(current_state(a))
     OD;
     vl := vl ++ tran_outcome{outcome := commit, tid := t};
     force_log();
     active := active -- t;
     committed := committed ++ t;
     release_locks(t) >>
```

```
PROC abort(t:TID) =
  ~(t IN active) => HAVOC [*]
  << active := active -- t;
     aborted := aborted ++ t >>;
  undo_tran(t);
  << vl := vl ++ tran_outcome{outcome := abort, tid := t};
     release_locks(t) >>

APROC crash() =
  << % crash
     aborted := aborted + active;
     VR := {};
     vc := store{};
     vl := log{};
     scanned := {};
     a_allocated := {};
     to_space := 1;
     active := {};
     RL := RL{* -> {}};
     WL := WL{* -> {}};
     % recovery
     redo();                          % repeat history
     undo();                          % abort active transactions
     FORK gc();                       % restart background processes
     FORK flush_log();
     FORK flush_cache();
     FORK truncate_log() >>

APROC delete_from_VR(a:va) =
  << VR := VR -- a >>

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                          Background Processes                              %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
PROC gc() =
  DO
    << VAR a:va | a IN a_allocated /\ a.space = to_space /\
          ~(a IN scanned) => scan(a) >>
    [] << VAR a:va | a IN translate_set(undo_roots(sl + vl), sl + vl) /\
             a.space = from_space => copy(a) >>
    [*] << flip() >>
  OD

PROC flush_log() =
```

```
  DO
    << VAR vlp:log, vls:log | vl = vlp + vls =>
          sl := sl + vlp;
          vl := vls >>
  OD

PROC flush_cache() =
  % implements write-ahead log rule
  DO
    << VAR a:va | ~modification_for_address(a, vl) =>
          sb[a] := vc(a); vc := vc{a -> }
       [] SKIP >>
  OD

PROC truncate_log() =
  DO
    << VAR slp:log, sls:log | sl = slp + sls /\
          ~(EXISTS a | vc!a /\ record_for_address(a, slp)) /\
          ~(EXISTS t | active_in_log(t, sl + vl) /\
                       update_for_t(t, slp)) /\
          ~copy_to_space(to_space, slp) =>
             sl := sls
       [] SKIP >>
  OD
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                          Internal Operations                              %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
APROC force_log() =
  << sl := sl + vl;
     vl := log{} >>

APROC flip() =
  << to_space := to_space + 1;
     scan_locks();
     scan_roots();
     scanned := {} >>

APROC scan_roots() =
  << DO
       VAR a:va | a IN SR /\ a.space = from_space =>
          copy(a); SR := SR -- a ++ forward(a)
     OD;
     force_log();    % to force copy records for SR to log, easy to avoid
     DO
       VAR a:va | a IN VR /\ a.space = from_space =>
```

```
          copy(a); VR := VR -- a ++ forward(a)
     OD  >>

APROC scan_locks() =
  << DO
       VAR a:va | RL(a) # {} /\ a.space = from_space =>
           copy(a); RL[forward(a)] := RL(a); RL[a] := {}
        [] VAR a:va | WL(a) # {} /\ a.space = from_space =>
           copy(a); WL[forward(a)] := WL(a); WL[a] := {}
     OD >>

APROC copy(a:va) =
  << current_state(a).forw IS NULL =>
         VAR b:va | ~(b IN a_allocated) /\ b.space = to_space =>
             a_allocated := a_allocated ++ b;
             oidmap[b] := oidmap(a);
             vc[a] := current_state(a);
             vc[b] := vc(a);
             vc[a].forw := b;
             vl := vl ++ copy{from := a, to := b}
     [*] SKIP >>

APROC scan(a:va) =
  << vc[a] := current_state(a);
     DO
       VAR p:rstate, v:value, s:rstate |
           vc(a).rstate = p ++ v + s /\ v IS va /\ v.space = from_space =>
               copy(v AS va); vc(a).rstate := p ++ forward(v) + s
     OD;
     vl := vl ++ scan{address := a};
     scanned := scanned ++ a  >>

PROC redo() =
  % recovers vc, a_allocated, scanned, and active
  VAR l:log := sl |
  DO
    ~is_empty(l) =>
       VAR lr: log_rec := l.head |
       l := l.tail;
       lr IS update => VAR ur:update := lr |
           vc[ur.address] := current_state(ur.address);
           vc[ur.address].rstate := ur.new_state;
           active := active ++ ur.tid
       [*] lr IS clr => VAR cr:clr := lr |
               vc[cr.address] := current_state(cr.address);
```

```
            vc[cr.address].rstate := cr.new_state;
            scanned := scanned -- cr.address
      [*] lr IS tran_outcome => VAR tr:tran_outcome := lr |
            active := active -- tr.tid
      [*] lr IS copy => VAR cr:copy := lr |
            vc[cr.from] := current_state(cr.from);
            vc[cr.to] := state{forw := nil,
                               rstate := vc(cr.from).rstate};
            vc[cr.from].forw := cr.to;
            a_allocated := a_allocated ++ cr.to;
            cr.to.space # to_space =>
                to_space := cr.to.space; scanned := {}
            [*] SKIP
      [*] lr IS scan => VAR sr:scan := lr |
            vc[sr.address] := current_state(sr.address);
            vc[sr.address].rstate :=
                redo_scan_state(vc(sr.address).rstate);
            scanned := scanned ++ sr.address
      [*] lr IS base_commit => VAR br:base_commit := lr |
            vc[br.address] := state{forw := nil, rstate := br.state};
            a_allocated := a_allocated ++ br.address
  OD

PROC undo() =
  DO
     VAR t:TID | t IN active => abort(t)
  OD

PROC undo_tran(t: TID) =
  % undo effects of transaction t
  VAR l:log |
      << l := remove_undone(log_for_t(t, sl + vl)) >>;
      DO
        ~is_empty(l) =>
            << VAR a:va, ur: update := l.last |
               l := l.removelast;
               a := translate(ur.address, sl + vl);
               vc[a] := current_state(a);
               vc[a].rstate :=  translate_state(ur.old_state, sl + vl);
               a.space = to_space => scanned := scanned -- a;
               vl := vl ++ clr{tid := t,
                               address := a,
                               new_state := vc(a).rstate} >>
      OD
```

```
APROC read_lock(t:TID, a:va) =
   % blocks till t obtains a read lock
   << WL(a) <= {t} => RL[a] := RL(a) ++ t >>

APROC write_lock(t:TID, a:va) =
  % blocks till t obtains a write lock
  << WL(a) <= {t} /\ RL(a) <= {t} => WL[a] := {t} >>

APROC release_locks(t:TID) =
  % releases t's locks
  << DO
       VAR a:va | t IN WL(a) => WL[a] := WL(a) -- t
       [] VAR a:va | t IN RL(a) => RL[a] := RL(a) -- t
     OD >>

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                                  Functions                                  %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
FUNC from_space() -> INT =
  RET to_space - 1

FUNC current_state(a:va) -> state =
  % accessor of virtual memory
  vc ! a  => RET vc(a) [*] RET sb(a)

FUNC state2sstate(s:state) -> sstate =
  RET rstate2sstate(s.rstate)

FUNC rstate2sstate(s:rstate) -> sstate =
  % converter from addresses to oids
  s.head IS va =>
      RET sstate{} ++ oidmap(s.head AS va) + rstate2sstate(s.tail)
  [*] RET sstate{} ++ (s.head AS scalar) + rstate2sstate(s.tail)

FUNC state_from_roots(s:rstate) -> BOOL =
  % predicate used in requires clauses
  RET addresses(s) <= VR + SR

FUNC forward(a: va) -> va =
  RET((current_state(a).forw AS va))

FUNC redo_scan_state(s:rstate) -> rstate =
  DO
    VAR sp:rstate, v:value, ss:rstate |
        s = sp ++ v + ss /\ v IS va /\ v.space = from_space =>
```

```
            s := sp ++ forward(v AS va) + ss
  OD;
  RET s

FUNC log_for_t(t: TID, l: log) -> log =
  VAR lp:log, lr:log_rec, ls:log |
      l = lp ++ lr + ls /\
      (~(lr IS clr \/ lr IS update) \/
       (lr IS clr /\ (lr AS clr).tid # t) \/
       (lr IS update /\ (lr AS update).tid # t)) =>
          RET log_for_t(t, lp + ls)
      [*] RET l

FUNC record_for_address(a:va, l:log) -> BOOL =
  RET (EXISTS lp:log, lr:log_rec, ls:log |
        l = lp ++ lr + ls /\
        ((lr IS update /\ (lr AS update).address = a) \/
         (lr IS clr /\ (lr AS clr).address = a) \/
         (lr IS scan /\ (lr AS scan).address = a) \/
         (lr IS base_commit /\ (lr AS base_commit).address = a) \/
         (lr IS copy /\ ((lr AS copy).from = a \/ (lr AS copy).to = a))))

FUNC modification_for_address(a:va, l:log) -> BOOL =
  RET (EXISTS lp:log, lr:log_rec, ls:log |
        l = lp ++ lr + ls /\
        ((lr IS update /\ (lr AS update).address = a) \/
         (lr IS clr /\ (lr AS clr).address = a) \/
         (lr IS scan /\ (lr AS scan).address = a) \/
         (lr IS copy /\ lr.from = a)))

FUNC copy_to_space(s:INT, l:log) -> BOOL =
  RET (EXISTS lp:log, lr:log_rec, ls:log |
        l = lp ++ lr + ls /\ lr IS copy /\ lr.to.space = s)

FUNC translate(a: va, l: log) -> va =
  is_empty(l) => RET a
  [*] l.head IS copy /\ l.head.from = a => RET translate(l.head.to, l.tail)
  [*] RET translate(a, l.tail)

FUNC translate_set(sa: SET[va], l:log) -> SET[va] =
  RET {a:va | a IN sa | translate(a, l)}

FUNC translate_state(s: rstate, l:log) -> rstate =
  DO
    VAR sp:rstate, ss:rstate, v:value |
```

```
        s = sp ++ v + ss /\ v IS va /\
            ((v AS va) # translate((v AS va), l)) =>
                s := sp ++ translate(v AS va, l) + ss
  OD;
  RET s

FUNC undo_roots(l: log) -> SET[va] =
  RET undo_addresses(remove_undone(extract_active(l, l)))

FUNC undo_addresses(l: log) -> SET[va] =
  is_empty(l) => RET {}
  [*] VAR ur:update := l.head |
      RET addresses(ur.old_state) ++ ur.address + undo_addresses(l.tail)

FUNC addresses(s: rstate) -> SET[va] =
  RET {i:INT | 0 <= i /\ i <= size(s) /\  s(i) IS va | s(i)}

FUNC extract_active(l: log, reflog: log) -> log =
  % extract update and clr records from l for transactions that are
  % active in reflog (order of extracted records is preserved).
  VAR p:log, lr:log_rec, s:log |
      l = p ++ lr + s /\
      (~(lr IS clr \/ lr IS update) \/
       (lr IS clr /\ ~active_in_log((lr AS clr).tid, reflog))
       (lr IS update /\ ~active_in_log((lr AS update).tid, reflog))) =>
          RET extract_active(p + s, reflog)
  [*] RET l

FUNC remove_undone(l:log) -> log =
  % expects l to contain only update and clr records
  % removes from l update, clr pairs (updates that have been undone)
  % returns a log that only contains update records
  VAR ss:log, ur:update , sp:log, clr:clr, p:log |
      l = ss ++ ur + sp ++ clr + p /\
          ~clr_for_t(clr.tid, ss ++ ur + sp) /\
          ur.tid = clr.tid /\ ~update_for_t(clr.tid, sp) =>
              RET remove_undone(ss + sp + p)
  [*] RET l

FUNC active_in_log(t:TID, l:log) ->BOOL =
  RET (EXISTS t:TID | update_for_t(t, l) /\ ~outcome_for_t(t, l))

FUNC clr_for_t(t:TID, l:log) -> BOOL =
  RET (EXISTS p:log, clr:clr, s:log | l = p ++ lr + s /\ clr.tid = t)
```

```
FUNC update_for_t(t:TID, l:log) -> BOOL =
  RET (EXISTS p:log, ur:update, s:log | l = p ++ lr + s /\ ur.tid = t)

FUNC outcome_for_t(t:TID, l:log) -> BOOL =
  RET (EXISTS p:log, tor:tran_outcome, s:log |
        l = p ++ lr + s /\ tor.tid = t)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                                Body of Module                                %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
BEGIN
  FORK gc();
  FORK flush_log();
  FORK flush_cache();
  FORK truncate_log()

END StableHeapImpl
```

Chapter 7

Implementation

To check that we have not overlooked important issues in the design and feasibility of our algorithms, we implemented a stable heap prototype. The current implementation of Argus [32] serves as the basis of our prototype; we replaced its existing storage management and recovery algorithms.

Argus is a programming language for reliable distributed computing [31]. An Argus program consists of a set of *guardians* that cooperate to perform a task. A guardian encapsulates or guards a resource; it permits access to its resource through remote procedures called *handlers*. The model of computation within a guardian is that of the stable heap: the roots of a guardian's heap are partitioned into stable and volatile sets, the state reachable from the stable roots survives failure, and computations run as atomic transactions. Computations involving more than one guardian run as distributed transactions.

We chose to use Argus because its implementation was available to us, and we had limited time to work on the prototype. Because a guardian contains a stable heap, Argus's run-time system already provides most of the functionality we require.

Transactions in Argus may be nested and distributed. Nesting is entirely orthogonal to our discussion. Distribution requires a two-phase commit protocol; but the changes to our algorithms that are necessary to support two-phase commit are obvious. Some of the actions that we describe as occurring at commit would actually occur during the first phase, which is called prepare. Our implementation supports nesting and distribution, but we avoid further discussion of these issues in order to simplify the presentation.

7.1 Overview

The approach to recovery for transaction abort in the current Argus implementation is based on versioning. To update an object, a transaction obtains a write lock on the object, makes a copy of the object called the *current version*, and modifies the current version. It also retains the previous version, which is called the *base version*. To abort a transaction, the recovery system discards the current versions of the objects that the transaction has modified. To commit a transaction, recovery installs the current versions of the modified objects as base versions.

Argus's approach to recovery from system and media failures is based on writing redo information to a log on stable storage; at transaction commit the recovery system writes new values for the objects that the transaction modified to the log. The heap of a guardian is in virtual memory, but Argus uses a mark, sweep and compact garbage collector that is not atomic. Thus, to recover from a crash, Argus must throw away virtual memory, and recover a guardian's stable state solely from the log.

This approach to recovery is inadequate for large heaps for four reasons. First, the time to do crash recovery is not independent of heap size: Argus recovers every stable object in the heap from the log. In earlier work we showed how to shorten the recovery time by using an atomic stop-and-copy collector and making use of the disk backing store together with the log [26, 25]. However, this recovery algorithm is not independent of heap size either; it requires a traversal over the stable object graph to clear volatile information stored in objects (lock and version information).

Second, Argus's representation for mutable atomic objects incurs a high storage overhead. It represents an atomic object using an object header. The header contains a unique object identifier and pointers to the object's base version, current version, and lock information. The header requires twenty bytes of storage, and takes up space whether or not the object is active (locked by an active transaction). In a large heap, only a small fraction of the objects will be active at any one time. Since many objects are small (e.g., a variant takes up eight bytes), a header of this size is extravagant.

Third, Argus's approach to recovery creates stable garbage. When a transaction acquires a write lock on an object, it creates a current version by copying the object's base version.

When the transaction commits, the base version pointer in the object's header is updated to point to the current version and the old base version becomes garbage. If the object is stable, this garbage is stable. Stable garbage must be collected with an atomic collector so it is more expensive to collect than volatile garbage.

Fourth, Argus's approach reduces reference locality. Access to an object is indirect through its object header, and the base version, which contains the actual value of the object, changes location for every committed update. Thus, objects that are placed together because they are referenced together move apart as they are updated. A copying garbage collector can repair the damage to locality caused by updates, but locality still decays during the interval between collections.

7.1.1 Solution

An approach based on write-ahead logging and repeating history could avoid all of the problems discussed above. In fact the last three problems are similar to issues raised by Gray [21] in his comparison of write-ahead logging to shadowing. However, the interfaces between the modules of Argus's run-time system assume an approach based on versioning and shadowing; to use write-ahead logging would require changing these interfaces. Since we had limited time for the implementation, we chose to solve these problems without changing interfaces.

Our approach divides the heap into stable and volatile areas; it allocates new objects in the volatile area, and moves newly stable objects to the stable area at the volatile collection after they become stable. The stable area is collected using our atomic incremental garbage collector; the volatile area using a stop and copy collector.

Instead of the shadowing approach that uses the atomic object headers, we use an intentions lists approach. We keep a *Lock and Version Table* (LVT) in the volatile area to keep track of the volatile information associated with atomic objects. It is a hash table that maps from an object's address to its lock information and current version. When a transaction obtains a write lock on an object, it creates a current version for the object in the volatile area and installs the version in the LVT. The transaction updates the current version. When a transaction completes (commits or aborts), it removes its entries from the

LVT.

To abort a transaction, recovery discards the current versions of the objects that the transaction has modified. To commit, recovery writes the current versions of the stable and newly stable objects a transaction has modified to the log in *data records*. A data record contains the object's address and its updated value. Using the stability tracking algorithm, recovery also tracks the newly stable objects accessible from the modified objects and writes *base-commit records* for the newly stable objects to the log. A base-commit record contains the address of an object and the base version of the object at the time the record is written. We call the process of writing data records and tracking newly stable objects *committing*. To complete the commit, recovery constructs a *commit record* containing the list of modified stable objects and forces it to the log. This list is called an *intentions list*. After forcing the commit record, recovery uses the current versions of the modified objects, now called intentions, to update the original objects in place.

Updating the original object in place with the intention is more expensive than inserting a pointer in an atomic object header. Instead of updating objects at transaction commit, recovery inserts an *intention forwarding pointer* in the object's descriptor. The intention forwarding pointer points to the intention. At the next garbage collection of the volatile area, when the collector has to move the intention anyway, it overwrites the original object with the intention. By waiting until the next garbage collection, we save work for objects that are updated frequently. Thus, although an intentions scheme would appear to require more copying than Argus's shadowing scheme, in practice the amount of copying should be about the same.

Waiting until the next garbage collection of the volatile area might appear to hinder the trimming of the log, since recovery requires that an update entry be available in the log until it is applied to the stable area and reaches disk. However, since we wait until the garbage collection of the volatile area to copy newly stable objects, recovery already has a similar restriction on log trimming.

Waiting until the next garbage collection of the volatile area could also reduce locality, since the new value of an object remains in the volatile area instead of being written back to the stable area. However, we expect that the intervals between volatile collections will

be short so that the extra time that a value spends in the volatile area will also be short; thus, the decrease in locality will be small.

We do recovery for volatile atomic objects in the same way we do recovery for the stable ones, except that no data record is written to the log at commit, and no entry is made in the commit record: locks and versions for a volatile object are kept in the hash table, an intention forwarding pointer is inserted in an object's descriptor at commit, and the volatile collector copies the intention instead of the original object.

7.1.2 Discussion

Our approach avoids the four problems of the pre-existing Argus implementation: (1) it keeps volatile lock and version information in the the Lock and Version Table (LVT), so there is no volatile information kept with stable objects; (2) it keeps entries in the LVT only for those objects that are locked by active transactions, so there is no extra storage overhead for inactive objects; (3) it creates a current version for an object modified by a transaction in the volatile area and writes the object's new value back to its place in the stable area after the transaction commits, so there is no garbage in the stable area due to versioning; and (4) it updates an object modified by a transaction in-place in the stable area after the transaction commits, so objects that are placed together to increase reference locality remain together.

However, our approach increases the cost of accessing an object and obtaining a lock on it. In the original Argus design, finding the value or lock information of an atomic object uses indirection through an atomic object header; our design requires that the access go through a hash table. Two alternatives are possible. Both are based on keeping lock and version information for an object in the volatile area, and keeping a pointer to that information in the object itself.

Under the first alternative, each object has one extra cell (four bytes). The recovery system inserts a pointer to the volatile information in this cell when a transaction obtains a lock. This cell must be cleared after a crash. To avoid a traversal of the whole object graph during recovery, the clearing can be done incrementally as objects are accessed by using page protections. After a crash, recovery protects all pages of the stable area. At the

first access to a page after the crash, the recovery system fields the trap, clears the volatile information from all the objects on the page, and then unprotects the page to allow the access.

Under the second alternative, the recovery system overwrites a cell of the object when a transaction obtains a lock, and notes the modification in the log. (The overwritten cell could be the object's descriptor.) After a crash, only the object descriptors that have been overwritten need to be recovered from the log. This alternative is more expensive than using the hash table for the first access by a transaction to an object, but it speeds subsequent accesses.

We chose the hash table approach because it is the simplest to implement within the context of the current Argus system. It is also an approach used widely in database systems [6, 20].

7.2 Details of Approach

Above, we outlined our approach to recovery for atomic objects. We still need to describe how to divide the heap, how to coordinate the atomic garbage collector with the recovery system, and how to ensure a recovery time independent of heap size.

7.2.1 Dividing the Heap

To divide the heap, the recovery system finds newly stable objects and makes them recoverable, and the volatile garbage collector moves a newly stable object to the stable area at the next collection after the commit of the transaction that makes it stable. By waiting until the collection after a transaction commits to move its newly stable objects to the stable area, we avoid stable garbage from aborted transactions. We described an algorithm for a system that uses write-ahead logging in Section 5.2.6. Instead of repeating the whole algorithm here, we describe how we approach the main issues. The earlier presentation brought up three issues: (1) moving objects to the stable area, (2) synchronizing the moving of objects with the recovery system, and (3) translating volatile addresses in the recovery information. We begin by discussing tracking. Then we discuss the first two issues and the collector for the volatile area; we defer discussion of the last issue until Section 7.2.2.

Tracking Newly Stable Objects

To track newly stable objects we use a simple algorithm similar to that of Section 5.1.3. The algorithm differs because it traverses only the object graph reachable through base versions. The simple algorithm is easier to implement than the more concurrent one. If we determine that more concurrency is needed, we can implement the more complicated one. We describe how to maintain the AS for the tracker in Section 7.3.3.

The recovery system invokes the tracker as it writes data records for the objects modified by a transaction to the log at commit. Specifically, when a transaction commits, recovery creates a *Newly Accessible Object Set* (NAOS) for the transaction. As recovery constructs a data record for a modified object it scans the object's current version for contained volatile objects and inserts the volatile objects in the NAOS. Then it invokes the stability tracker for each object in the NAOS.

Moving Objects

To move newly stable objects we must keep track of the objects in the stable area that contain pointers to the volatile area. To allow intentions to be installed by the volatile collector, we already keep track of the objects in the stable area that were updated by committed transactions. We keep track of these objects in the *Stable Committed Object Set* (SCOS). Recovery inserts stable objects in the SCOS when it installs intention forwarding pointers at commit. The set of objects that contain pointers to the volatile area is a subset of the SCOS. Therefore, when the collector uses the SCOS to install the new committed versions of objects, it scans these objects for pointers to the volatile area and moves the object graphs rooted at them to the stable area.

The collector must move the newly stable objects recoverably. Therefore, as each newly stable object is moved to the stable area, the collector writes translation information for it to the log. Since the collector copies the newly stable objects while all other processing is halted, it batches the translation information into one large record to avoid overhead. That record is called the *volatile flip record* and it replaces the v2scopy records mentioned in Section 5.2.6.

When the collector moves the newly stable objects, it does not write s4vscan records

for the objects in the stable area that were updated; recovery deduces which objects were updated at a volatile flip by reading the log segment written since the previous volatile flip record. Recovery must read this segment anyway in order to recover the states of newly stable objects.

Synchronization With Recovery System

The movement of newly stable objects must be synchronized properly with the recovery system so that the newly stable object graphs and the updated objects that point to them are recoverable. In Section 5.2.6 we described how the collector brings the system to an update-quiescent state by finishing tracking and updates before doing anything else. The equivalent for the recovery system we implemented is to have the collector bring the system to a commit-quiescent state: a state in which the recovery system is not in the middle of logging a data record, and the tracking of newly stable objects has completed for all of the data records in the log. It turns out that this solution is difficult to implement within the context of the existing Argus implementation, since it requires us to write a recovery procedure for logging data records and tracking newly stable objects that can be run both by the garbage collector and the recovery system.

We implement a different solution. We allow the volatile collector to run even if the recovery system is not in a commit-quiescent state. However, we record extra address translations in the log. We describe the solution in our description of a volatile collection below.

Collection of the Volatile Area

As its first step the volatile collector updates stable objects with their intentions and copies the objects accessible from them to the stable area. As it copies each object to the stable area, it records a volatile to stable address translation in the volatile flip record. Next, the collector processes the NAOS of each transaction in the process of committing. For each object in an NAOS, it copies the object to the volatile to-space and records a translation pair in the flip record. The translation pair maps the object's old volatile address to its new volatile address in to-space. Finally, the collector traces from the regular volatile roots.

When it copies an object in the AS to to-space during the trace, it also records a translation pair in the flip record. When the volatile collection is complete, the collector writes the flip record containing the translations to the log.

The above procedure ensures that both the newly stable object graphs for which tracking is incomplete and the updated objects recorded in data records for which tracking is incomplete are recoverable: for any volatile address recorded in a data record or base-commit record written since the last flip, there is a translation pair in the flip record if the object is still accessible from a stable root. Thus, information written before the flip can be connected with information written after the flip. There are three cases with respect to the commit and tracking process: (1) data record in the log, tracking complete, and commit record in the log; (2) data record in the log, tracking complete, but commit record not in the log; and (3) data record in the log, but tracking incomplete. Figure 7.1 shows the three cases. In the first case, the object whose value is recorded in the data record is in the SCOS. Thus, the objects referenced by volatile addresses in the sub-graph rooted at the object are copied to the stable area by the volatile collector, and there is a translation for each of these addresses in the flip record. In the second case, the current version of the object whose value is recorded in the data record is in the Lock and Version Table and the objects referenced by volatile addresses in the sub-graph rooted at the current version are in the AS. Thus, there is a translation for each of the volatile addresses in the flip record. In the third case, the volatile addresses recorded in log records for which tracking is incomplete are in the NAOS of a committing transaction; thus, translations for these addresses are also in the flip record.

7.2.2 Atomic Incremental Garbage Collection

An atomic incremental garbage collector collects the stable area. In Chapter 4 we discussed the interactions between an atomic collector and a recovery system based on write-ahead logging and repeating history. Here we discuss the interactions between the collector and the recovery system of our prototype. Chapter 4 raises four issues: (1) roots in the recovery information, (2) translating addresses in recovery information, (3) recovery independent of heap size, and (4) recoverable allocation of spaces. We discuss the first two below. We

Case 1

Data	Base-Commit	Commit	Volatile-Flip
Adr: SA1 Value: VA1	Adr: VA1 Value:	SA1	VA1 → SA99

AS: ... VA1 ...

Case 2

Data	Base-Commit	Volatile-Flip
Adr: SA1 Value: VA1	Adr: VA1 Value:	VA1 → VA93

LVT: SA1 → [VA1]
AS: ... VA1 ...

Case 3

Data	Volatile-Flip
Adr: SA1 Value: VA1	VA1 → VA93

NAOS: VA1

Figure 7.1: Synchronization with Recovery System: Three Cases

defer discussion of recovery independent of heap size to the discussion of checkpoints in Section 7.2.3. The solution for recoverable allocation of spaces remains the same. We also discuss some implementation details.

Roots in Recovery Information

Roots in the information maintained by the recovery system are not a problem. Our recovery system maintains both redo and undo information in virtual memory. An object's base version corresponds to its undo information, and its current version to its redo information. The base version is the value of the object itself, so all objects reachable through base versions are retained by the tracing from the stable roots. Current versions are reachable from the Lock and Versions Table; the collector of the volatile area retains them also. Because our recovery system writes only redo information (no undo information) to the log, any object required for recovery after a crash is either accessible from a root in virtual memory, or recoverable from the part of the log that recovery processes.

Translating Roots in Redo Information

A transaction for which there is a commit record in the log in the current stable collection interval may have written one of its data records in the penultimate collection interval. This is very unlikely, and could be avoided for single site transactions, but it could occur for a distributed transaction whose commit is delayed by a communication failure during two-phase commit. The problem is one of efficiency: we must translate the addresses in the data record efficiently during crash recovery. It is not a problem of correctness: recovery can use the address translations in volatile flip records and copy records to translate the addresses in the data record to to-space addresses.

For faster recovery we could implement a solution that uses tables similar to the UTT, UTR, and VUTR used for translating addresses in undo information in Sections 4.4 and 5.2.6. This solution is important for recovery based on write-ahead logging, because the efficiency of address translation affects normal operation; both the garbage collector and transaction abort use the UTT to avoid reading copy records from the log. However, for our prototype we need to translate addresses only during crash recovery, so we chose not to

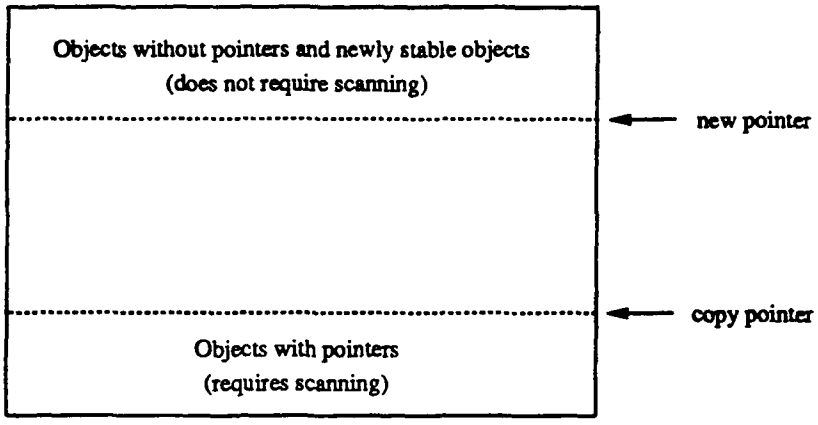


Figure 7.2: Layout of To-space

optimize translation in our implementation.

Implementation Details

Below we present some implementation details for our atomic incremental garbage collector. We describe how we scan arbitrary pages of to-space, how we schedule garbage collection work, and how we deallocate from-space at the end of a collection.

In our implementation, the collector can scan an arbitrary page of to-space without the need for the Last Object Table mentioned in Section 3.2. Each object has a descriptor holding its low-level type and its size; the type says whether or not the object may contain pointers. The collector copies objects that may contain pointers to the low part of to-space and the objects that do not contain pointers to the high part. Only the low part of to-space needs scanning. Pointers are recognizable because they are tagged; also, object descriptors do not hold bit patterns that could be confused with a pointer. Thus, the collector can scan arbitrary pages in the low part of to-space and recognize all pointers correctly. Figure 7.2 shows the layout of to-space.

To ensure that the collector finishes a collection before it runs out of room in to-space, we coordinate collection with object allocation as Baker does [4]. Effectively, object allocation occurs in the stable area when a transaction commits and makes a new object stable. Therefore, recovery checks whether or not to invoke the collector to scan a page just after committing a transaction. It bases its decision on the number of bytes that were allocated in

the stable area since the last invocation of the collector. The ratio of scanning to allocation depends on the proportion of to-space occupied by live objects, as in Baker's algorithm.

The Argus run-time has lightweight threads that it schedules within a Unix process using a round robin scheduler. We chose Baker's method instead of running the collector in one of these threads because there is no easy way to schedule the thread to ensure that it completes its work on time.

When a collection is complete and all of to-space has been scanned, recovery initiates a checkpoint to ensure that all scanned pages reach disk. When the checkpoint completes, from-space can be deallocated. We describe checkpoints in Section 7.2.3.

7.2.3 Recovery Independent of Heap Size

Our recovery system uses simple checkpoints that avoid delaying the progress of transactions. To start a checkpoint, it spools a *checkpoint record* to the log and determines which pages of the stable area are dirty. Then, while transactions are running, it writes the dirty pages back to disk in the background. When all of the dirty pages are on disk, it spools an *end-checkpoint* record to the log. A checkpoint is not complete unless its end-checkpoint record is in the log.

Except for modifications by the atomic garbage collector, objects in the stable area are physically updated only during a volatile collection. Thus, a checkpoint ensures that the updates of transactions that committed before the last volatile garbage collection are on disk. Some transactions may have been in the process of committing (writing data and base-commit records to the log when the collection started). To easily find the data and base-commit records for these transactions after a crash, the volatile flip record contains the list of committing transactions; for each transaction it contains the log sequence number of its first data record. Figure 7.3 shows all of the information in the volatile flip record: volatile to stable translations, volatile to volatile translations, and the list of committing transactions.

The checkpoint record contains the Scanned Set, the allocation pointer for the stable area at the time of the checkpoint, and the log sequence number of the last volatile flip record. The Scanned Set and the allocation pointer serve the same purpose as in the

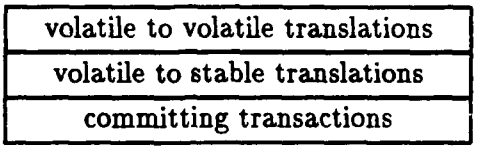


Figure 7.3: Format of Volatile Flip Record

checkpoint record for write-ahead logging discussed in Section 4.6. The pointer to the flip record allows recovery to determine quickly where to start its redo pass after a crash.

7.2.4 Recovery

Here is a sketch of crash recovery; optimizations are possible. Recovery does one redo pass through the log to recover objects in the stable area that did not reach disk before the crash. Redo starts at the minimum LSN recorded for the first data record of a committing transaction in the flip record immediately preceding the last checkpoint record.

Recovery builds an *Object Table* (OT) in the volatile area. The OT maps from an object's address to a pair of versions: one is a current version and the second a base version. We describe how recovery updates and uses the information in the OT as it processes log records. When redo processes an object's data record, it searches the OT for the object's entry. If there is not an entry it creates an empty one. In either case, redo updates the current version pointer in the entry to point to the object version recovered from the object's data record. When redo processes an object's base commit record, it creates an entry in the OT mapping from the object's address to a pair in which the current version is empty and the base version pointer points to the object version recovered from the base-commit record. When redo processes a commit record, it iterates over the list of objects that the transaction updated: it looks up the entry for the object in the OT, changes the base version pointer to point to the current version, and sets the current version pointer to nil.

When redo processes a volatile flip record, it iterates over the entries in the OT:

1. if the entry is associated with a volatile address, it rebinds the entry according to the address translation in the flip record; i.e., it deletes the entry for the old address and re-inserts it for the translated address;

2. it scans the versions in the entry and translates the volatile addresses according to the address translations in the flip record;

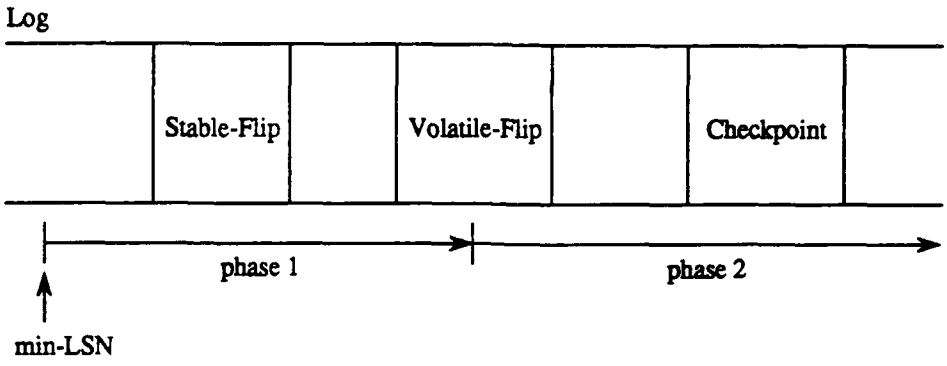


Figure 7.4: Two Phases of Recovery

3. if the entry contains a base version and it is associated with a stable address, it copies the base version to the object's place in the stable area; if the entry does not also contain a current version, it deletes the entry from the OT.

The redo pass consists of two phases: the first processes the part of the log from the starting point up to the last volatile flip record before the checkpoint record; the second processes from the volatile flip record to the end of the log. Figure 7.4 shows the phases. The checkpoint ensures that the effects of all transactions whose commit records precede the flip record are on disk. Thus, the first phase only has to process data records for transactions that were in the process of committing at the time of the volatile flip, and the base-commit records for newly stable objects reachable from these records. Our data records do not contain a transaction identifier, so in practice the first phase has to process all of the data records. If this turns out to be a performance problem, we can add transaction identifiers to the data records. However, the first phase can use information in the volatile flip record to avoid processing some base-commit records. If there is a volatile to volatile translation in the flip record for the object for which a base-commit record was logged, the object is recovered from its base-commit record. A base-commit record for which there is a volatile to stable translation in the flip record does not have to be recovered; its object was written to disk by the checkpoint. A base-commit record for which there is no translation in the volatile flip record also does not have to be recovered; it is for an object that is no longer reachable. The second phase processes all data records and base-commit records that it encounters.

Redo handling of a copy or scan record written by the atomic garbage collector depends

on whether the record was written before or during the stable garbage collection interval interrupted by the crash. If written before, redo ignores the record; the checkpoint taken at the end of the previous collection ensures that its results are on disk (we initiate a checkpoint at the end of a collection so that we can reclaim its from-space). Processing of a copy or scan record also depends on whether it was written before or after the last checkpoint record. If written before, redo ignores the record because the checkpoint ensures that the result of the scan or copy has reached disk. The copy and the scan are redone only if the record was written both after the last checkpoint and during the stable collection interval that was interrupted by the crash.

7.3 Other Implementation Details

To get a working implementation we also ported Argus to Mach, modified the Argus linker so that Argus programs can call C subroutines, and determined a representation for the AS.

7.3.1 Porting Argus to Mach

The pre-existing Argus implementation runs under Unix; we ported Argus to run under Mach [40]. Mach allows the collector to set protections on pages of virtual memory, and delivers protection traps to be handled by the garbage collector. It also allows the allocation and deallocation of areas in virtual memory so that spaces for the volatile and stable areas can be dynamically allocated, enlarged, and deallocated as required. Furthermore, Mach provides an external pager interface [55]. By writing an external pager, we can ensure that the disk backing store for the stable area survives a crash, and we can also control paging to the backing store and coordinate it with writing to the log.

Argus only runs on Vaxes because parts of its run-time are written in Vax assembler. The combination of Mach and the Vax might not be the most efficient implementation of the features discussed above, but it allows us to build a prototype. Appel and Li have looked at the proper combination of virtual memory hardware and operating system support to provide page protection features efficiently [2].

7.3.2 Calling C Subroutines from Argus

Argus has its own assembler, compiler, and linker. The format of its object files is not compatible with the Unix object file format (a.out). Thus, our implementation was constrained to be written in assembler, Argus or a combination of the two. Since the current recovery system is already written in Argus, we decided to write the new recovery system in Argus. The current garbage collector is written in assembly language; it cannot be easily written in Argus because it runs when the heap is exhausted. We did not want to write the collector in assembly language for obvious reasons. Therefore, we changed the linker to allow subroutines written in C to be called from an Argus program, and we wrote the collector in C. We also required the capability to call C subroutines in order to make Mach system calls; routines in the system call libraries distributed with Mach cannot be called directly from Argus because Argus uses a different calling convention and a different stack layout.

We divided the problem into two parts: (1) resolving externals and (2) resolving the different calling and stack conventions. To solve the first problem we modified the Argus linker so that it can load a Unix object file into the Argus executable, resolve Argus calls on an external C routine, and resolve C calls on an Argus routine. We require that all C routines called by an Argus program be pre-linked by the Unix linker into a single a.out file so that the Argus linker does not have to worry about the internal linking between C sub-programs.

To solve the second problem, the programmer writes stubs to account for the differences in calling conventions and stack layouts. A stub generator could have been written, but we only required a small number of special purpose stubs. The Argus run-time system also maintains a separate stack for calls to C subroutines; the regular stack cannot be used because a C subroutine may leave behind values that the collector could misinterpret as Argus pointers.

We restrict the use of the C library: specifically, no calls to malloc or subroutines that call malloc are allowed. Malloc would interfere with the Argus run-time system, which manages its own memory.

Finally, calls from C back into Argus are allowed, but severely restricted. An Argus

routine called from C may not allocate objects on the heap, and it may not signal exceptions. The garbage collector, written in C and invoked from Argus, calls an Argus routine to write its records to the log.

7.3.3 Finding Bits in Object Descriptors for the AS and the LS

Our implementation uses the simple tracking algorithm, so it requires an AS but not an LS. However, we designed and implemented a representation that can accommodate both sets so that we can implement the concurrent tracker in the future.

There are bits available in object descriptors for the AS and the LS, but they are not usable without extensive modifications to the compiler and run-time system. To illustrate the problem, consider the descriptor for a vector, the low-level object used to implement records and sequences. The run-time system uses three bits to indicate that the object is a vector and eighteen bits to indicate the vector's length. Since a descriptor is thirty-two bits, this should leave eleven bits that can be used for other purposes. However, all code that manipulates vectors assumes that the type code for a vector is 000, and that all other bits in the descriptor except for the vector size are 0; the code never masks out those bits. Thus, even though the eleven bits remain unused, we cannot take advantage of them.

We searched for a solution that would allow checks for membership in the AS and the LS to be fast, yet not require too much extra space. A hash table would be slow compared to bits in the objects themselves; an extra cell in every object just to hold the bit would waste a great deal of storage. We noticed two things: (1) all objects in the stable area are in the AS and the LS, so they do not require a bit; and (2) immutable objects do not require bits because they are never modified. Only atomic mutable objects in the volatile area remain.

We represent atomic mutable objects using a one cell (four bytes) object. The cell is both a descriptor and a value; it points to the actual object. This one cell object can be differentiated from other objects because no other object has a descriptor that looks like a pointer (an earlier change that we made so that forwarding pointers can be stored in object descriptors). Since all objects start on four byte boundaries and pointers never point to the interior of an object, the two least significant bits of the cell are available for other purposes.

They are used to differentiate the cell from a garbage collection forwarding pointer and to indicate membership in the AS, AS and the LS, or neither set. The cost of the indirection through the cell for access to an atomic mutable object is marginal; all access goes through the Lock and Version Table anyway.

7.4 Status of Implementation

Due to time constraints, we were not able to implement the whole design. We implemented the division of the heap into stable and volatile areas, the atomic incremental garbage collector, and changes to the run-time recovery system. We did not implement an external pager (so that the disk backing store of the stable area would actually persist after a crash), the crash recovery algorithm, or checkpoints. This partial implementation allows the measurement of the run-time costs for recovery and atomic garbage collection during normal operation. It also allows us to estimate the cost for crash recovery by measuring the volume of data written to the log.

7.4.1 Restrictions on Argus Programs

Not all programs that run under the current Argus system run under our implementation. We do not support recoverability for mutex objects, and we do not allow objects of non-atomic mutable types to be accessible from a stable root; recovery for these objects is too expensive. The Argus manual specifically discourages the use of non-atomic mutable types in stable variables, but the language allows them. We discussed both problems in our earlier work [26].

An Argus programmer uses the mutex type to implement user-defined atomic types. Instead, we propose that the language be modified so that user-defined atomic types can be implemented using built-in atomic types. Our modification would allow a nested top-level transaction to obtain locks on behalf of the transaction enclosing it; this would allow multi-level transactions [51], which is a generalization of the capability that mutex provides. It is also the way user-defined atomic types are implemented in Avalon [15, 23]. The actual language constructs are left for future work.

7.5 Measurements

Using our implementation we plan to take some micro-measurements, measurements of the costs of specific features that do not depend on the application using the heap, and macro-measurements, overall measurements of sub-systems that are application dependent. We describe our plans below.

7.5.1 Micro-Measurements

The most interesting micro-measurements are those that will help us evaluate the cost of our atomic garbage collector and the cost of supporting persistence based on reachability by tracking stable objects and dividing the heap. These measurements include the following costs: the cost of an atomic copying step, the cost of an atomic scan step, the cost to handle a trap and scan a page, and the cost of tracking a newly stable object. Using these measurements we can evaluate the cost of atomic garbage collection. We can also evaluate our decision to create atomic objects in the volatile area and move them to the stable area after they become stable. A simple alternative would be to create all objects that might ever become stable (all mutable atomic objects and all immutable objects) in the stable area, so that they would be recoverable even before they become reachable from a stable root. This simple alternative would eliminate the cost of tracking newly stable objects; however, it would increase the cost of garbage collection, because objects that would never become stable would have to be treated as if they were stable.

Other micro-measurements will allow us to compare our implementation to the implementation of non-object-oriented transaction systems, such as many databases and to the previous implementation of Argus. These measurements include the following costs: the cost of obtaining a read lock, the cost of obtaining a write lock, the cost of accessing an object that is already locked, the cost of committing a read-only transaction, the cost of committing an update transaction that makes no new objects stable, and the cost of aborting a transaction. These measurements have been published for the current Argus implementation [32].

7.5.2 Macro-Measurements

There are many interesting parts of the system to measure that depend on specific workloads. By workload we refer to a specific application: its operations, the frequency with which the operations are invoked, and the size and structure of the heap. For our implementation the most important measurements are those that measure the performance of atomic incremental garbage collection.

We would like to measure how incremental our atomic garbage collector really is; i.e., how long is a transaction delayed by the garbage collector? We need to measure two quantities: the length of the longest pause for garbage collection and the distribution of the pauses. The longest pause is likely for the flip; the time for the flip is probably dominated by the collection of the volatile area. The other kind of pause is the time to scan a single page. The distribution of pauses is such that there are many pauses just after a flip (traps to access an unscanned page), and few or no pauses just before the next flip. If the distribution of traps is too heavily skewed to occur very close to the flip, the garbage collector is not *very incremental*. The problem of trap distribution is not particular to our atomic garbage collector; it is a problem for any collector based on the Ellis read barrier. However, it would be interesting to compare the distribution of pauses for a program that accesses a normal volatile program heap and a program that accesses the stable area of a stable heap. We expect that the rate of access to the stable area of a stable heap is lower that the rate of access of a normal program to its heap; thus, the distribution of pauses should be more favorable for a stable heap.

Other macro-measurements will allow us to compare our implementation to the implementation of non-object-oriented transaction systems such as databases. These measurements could include the throughput for certain classes of transactions, for example, the debit-credit transaction. It would also be interesting to measure the cost of checkpoints and the cost of recovery from system crash as a function of the amount of log that needs to be redone. However, these parts of the system have not been implemented yet.

Chapter 8

Related Work

Our work draws on previous work on garbage collection and on recovery. Our goal was to design algorithms to efficiently manage a large stable heap; the individual garbage collection and recovery methods that we have chosen are not fundamentally new, but are based on existing ones. For that reason, the discussion of related work concentrates on other attempts or proposals to integrate garbage collection and recovery: PS-algol [3], previous work on the Argus recovery system [25, 26, 38, 39], and Detlefs's concurrent atomic garbage collection for Avalon [16]. We also discuss several proposals from the persistent programming community to build transactions on top of a persistent heap [10, 47].

8.1 PS-algol

PS-algol [3, 13, 37] is a language for persistent programming. There are many similarities between the system model of PS-algol and a stable heap: both define persistence according to reachability from a persistent root, and both use transactions to control concurrency and provide fault-tolerance. However, the transaction model of PS-algol is primitive: it restricts concurrency, and does not provide for recovery from media failure. More importantly, the run-time overhead for recovery in the implementation is high, and the persistent store is garbage collected offline while no transactions are active.

A programmer using PS-algol partitions his data into databases; databases may contain arbitrary object graphs and may be interconnected. Read/write locking on whole databases is used for concurrency control. When a program opens a database in read or write mode, the system also opens all of the databases recursively accessible from the opened database

in read mode. The first open of a database for write implicitly starts a transaction. When the program calls the commit procedure, the transaction commits and a new one starts. At commit, changes are written atomically to the databases open for writing.

The limited concurrency of the transaction model allows a simple recovery system, which uses shadow pages. Shadow pages permit recovery after a crash to be fast, but commit is very slow. Briefly, commit occurs as follows: first, recovery writes the changed and newly persistent objects to fresh disk pages; then, it updates a separate index for each modified database; finally, it updates a master directory containing pointers to the indices. This process ensures that the new values for objects replace the old values in a single atomic step. However, a simple transaction that updates a single object requires at least three synchronous disk writes, each to different places. Recovery systems for databases commit transactions with lower overhead by writing to a log.

There are two kinds of garbage collection in PS-algol: garbage collection of a program heap and garbage collection of the persistent store [11]. Garbage collection of the program heap collects volatile objects; it may occur while a transaction is active. If necessary, it may write changed and newly persistent objects to database shadow pages to make room in main memory. Garbage collection of the persistent store uses stop-and-copy collection to free disk space from databases. This garbage collection occurs offline, and all databases must be closed for its duration.

8.2 Current Argus Recovery System

The approach to recovery and storage management taken by the current Argus implementation [32, 38, 39] is suitable only for small heaps or applications that can tolerate long garbage collection pauses and long recovery times. Run-time overhead for recovery, e.g., time to commit a transaction, is low because Argus writes to a log instead of updating disk directly. However, the recovery system does not differentiate between system and media failures; it treats every crash as a total media failure. Thus, it always discards the disk backing store of virtual memory and recovers the whole stable state from the log. Therefore, the time for recovery depends on the sizes of the log and the stable state. For storage management the Argus implementation employs a stop-the-world garbage collector. The collector

can run while transactions are active; it suspends active transactions when it starts, and allows them to resume after it completes.

The idea to track newly stable objects was introduced by the Argus recovery system. However, as we mentioned earlier in Section 5.1.3, the algorithm designed for Argus and used in its implementation suffers from a race condition; we have presented a correct algorithm.

8.3 Atomic Copying Collection

In earlier work [25, 26], we introduced the concept of atomic garbage collection and showed how to provide fast recovery for small heaps. Recovery for a system crash is faster than the current Argus recovery system because it takes advantage of information left on disk in virtual memory and brings that information to a consistent state by reading a small part of the log. However, the approach is still limited to small heaps: it uses stop-the-world collection and during crash recovery it traverses the whole object graph to clear volatile information out of the representations of stable objects.

8.4 Concurrent Atomic Garbage Collection

Avalon [15] is a language for distributed computing built as an extension to C++ on top of the Camelot transaction facility [46]. Detlefs has designed an atomic concurrent garbage collector for Avalon to provide better support for large heaps [16]. His work is the closest to ours; he also bases his collector on the read barrier of Ellis and his recovery system on write-ahead logging. However, there are several important differences. Though the pauses for garbage collection and the time for recovery in his algorithms are independent of heap size, the pauses for collection are not short; a collection pause requires multiple writes to disk that are both synchronous and random. Also, his algorithm has higher overhead for handling addresses in undo information.

Below we discuss how the Avalon system model is different from our model of a stable heap. Then we discuss Detlefs's method for concurrent atomic garbage collection.

8.4.1 Different System Model

The system model differs on two key points. First, Avalon's notion of stability (Camelot and Avalon use the term recoverability in place of stability) differs from ours. In Avalon, an object is stable because the program creates it in a stable area; in our model, an object is stable because it is reachable from a stable root. Avalon's notion simplifies storage management, and its recovery system does not need to track newly stable objects.

Second, Avalon is an extension to C++, which is itself an extension to C. Because types in C can be freely converted between pointer types and scalar types, the run-time system has no way to distinguish between a real pointer and a scalar that looks like a pointer. Thus any garbage collector for C must be conservative [5, 8]; it must treat any value that could be a pointer as both a pointer and scalar: the "object" to which such a value "points" cannot be collected; neither can it be moved. Bartlett calls these *ambiguous pointers* [5].

8.4.2 Concurrent Atomic Garbage Collection

Detlefs designed his algorithm to work with Camelot's recovery system, which is based on write-ahead logging with physical redo and physical undo [42]. He bases his garbage collection algorithm on the read barrier of Ellis [18] for concurrency, and forbids the use of dangerous C constructs and uses the mostly copying collector of Bartlett [5] to deal with ambiguous pointers. By forbidding certain C constructs, the problem of conservative pointer finding is simplified and turned into a problem of conservative root finding – the collector cannot tell which registers and stack locations contain pointers. Bartlett's collector deals with these ambiguous roots by promoting the whole page of an object to to-space if the object is referenced by an ambiguous pointer. The collector promotes pages before it copies any objects, so it does not inadvertently copy an object that should be promoted. To make this algorithm concurrent, Detlefs requires that all page promotions occur during the flip.

Detlefs makes his concurrent collector atomic by making the scan of each page recoverable. As the collector scans a page, it pins each page it modifies: the to-space page it is scanning, the to-space pages to which it copies objects, and the from-space pages on which it writes forwarding pointers for the objects it copies. Next, the collector forces a *gc-scan-page* record to the log that contains the address of the scanned page and the addresses of the

to-space pages to which objects were copied. This information allows the scan to be redone after a crash. The record is forced so that the scan is redoable before any of the modified pages reach disk. Once the record has reached the stable log, the scan completes by forcing its pinned pages back to disk in a prescribed order: first, the to-space pages containing copied objects; second, the from-space pages; and finally, the scanned page. This order avoids the lose of object descriptors and forwarding pointers, which are problems that we described in Section 3.3.1.

Using the procedure described above, the scan of a single page requires at least three random synchronous writes even in the best scenario: one log buffer, one from-space page (assuming all of the copied objects are on a single page), and one to-space page (assuming the scanned page and the page to which objects are copied is the same); in most cases it probably requires many more. A transaction trying to access an unscanned object must wait for these writes to complete. In contrast, our algorithm does not delay a transaction with any synchronous writes; instead, it spools a small amount of recovery information to the log. Detlefs suggests amortizing the I/O overhead of his algorithm by scanning more than one page at a time. However, this does not solve the problem of garbage collection traps in the critical path of a transaction.

Detlefs's solution to the translation problem for addresses in undo information also incurs higher overhead than our solution. To deal with the problem his algorithm avoids moving objects that are write-locked by active transactions as well as objects directly reachable from undo information for these objects. To do this he uses the same promotion mechanism used for the targets of ambiguous roots. Since all promotion occurs during a flip, the collector must read the log and find undo information for active transactions at the flip. Our algorithm, in constrast, waits to deal with these undo roots until the end of the collection. By this time most transactions that were active at the time of the flip should be finished, and the chance that the collector will access the log is small. Promoting pages interferes with the work of the collector because it increases fragmentation, so Detlefs also suggests an alternative that avoids promoting the objects reachable from undo information. However, this alternative still reads the undo information for active transactions from the log during the flip.

8.5 Persistent Heaps

There have been several proposals from the persistent programming community to build transaction systems on top of a persistent heap. These proposals make sense only if persistence is useful in the absence of transactions; otherwise, the extra layer introduces unnecessary inefficiency. The first proposal for building transactions on top of a persistent heap is due to Thatte [47]. A more recent implementation of a persistent heap is described by Brown [10].

Both Thatte and Brown build their systems in layers. The low level provides a persistent memory, a virtual memory abstraction that survives crashes. To provide persistence, a recovery system takes checkpoints at regular intervals, or when called from a higher layer. After a crash, the recovery system restores the memory to its state at the last checkpoint.

A heap layer sits on top of the persistent memory. Certain objects in the memory are designated roots of the heap; an object persists if it is reachable from one of these roots. A normal garbage collector reclaims memory by tracing from the roots.

A major difference between Thatte and Brown is the way they implement persistent memory. Below we describe their approaches, as well as a third approach based on repeating history. Then we discuss possible problems with their approaches. Finally we describe how to layer a transaction system on a persistent heap.

8.5.1 Persistent Memory

Thatte calls his implementation Persistent Memory. His recovery system uses shadowing: it maintains two backing store pages on disk for each active page of virtual memory, using one for swapping and the second for checkpoints. At regular intervals the system checkpoints: first, it writes all dirty pages of main memory to their respective swapping pages; then, it switches the roles of the disk pages in each pair to create a new checkpoint. The checkpoint also saves processor state. After a crash the states of virtual memory and the registers revert to their states at the last checkpoint, after which normal system operation resumes.

8.5.2 Persistent Object Store

Brown calls his persistent memory layer a stable store. He uses a technique similar to undo logging (though he does not call it undo logging) to implement persistence. The first time a page is modified it is pinned in main memory until a copy of the page's old value reaches the stable log. To checkpoint, all of the modified pages are written back to their places on disk and the log is discarded. To recover after a crash, the state at the last checkpoint is restored by overwriting modified disk pages by their old copies in the undo log. A Persistent Abstract Machine runs on top of the persistent store and keeps all of its state in the persistent store; the checkpoint restores a state from which the machine can be restarted.

8.5.3 Repeating History

Persistent memory can also be implemented by using an approach based on repeating history. For every modification to persistent memory, the recovery system follows the write-ahead log protocol and writes a record describing the modification to a redo log. To checkpoint the recovery system forces the redo log. To recover after a crash the system repeats history.

8.5.4 Problems

Thatte does not address an important problem adequately – the restart of the processor after a crash. Often system crashes are caused by software and not by faulty hardware. If the processor starts with the same state as existed at the last checkpoint, what prevents it from crashing again? Also, what does it mean to restart a program in the middle of input or output with the external world? Brown's system suffers from similar problems; however, the extra layer of the abstract machine can mitigate them to some degree.

8.5.5 Transactions

Building transactions on top of a persistent heap requires an extra layer. We describe two methods. Then we discuss atomic garbage collection.

Method 1

Thatte proposes that transactions update objects in place, keep undo information in the persistent heap for the duration of the transaction, and write redo information to a log separate from the persistent store. After a system failure, the persistent heap is restored to its state at the last checkpoint; the recovery system applies the undo information saved by the checkpoint and the redo information written since the checkpoint to bring the objects up to date.

For all three approaches to building persistent memory this method for transactions has a high copy overhead. There could be four copies of an active object: (1) the copy updated by transactions, (2) the copy in the persistent store's checkpoint, (3) the copy in the redo log, and (4) the copy in the undo log. In a transaction recovery system based on repeating history built without the persistent memory layer there are never more than three copies. The time and storage required for the extra copy could be significant.

Method 2

An alternate way to implement transactions would be to use the checkpoint for commit. A transaction updates an object in place and keeps undo information in the heap. To commit, a transaction initiates a checkpoint. After a system crash the persistent heap is restored to its state at its last checkpoint, and the undo information is used to abort active transactions. This method requires less copying of objects, but commit is expensive in both Thatte's and Brown's approaches; their checkpointing mechanisms require random synchronous writes to disk. However, this second method for transactions integrates nicely with persistence based on repeating history; it is the reason that this approach to persistence was suggested.

Atomic Garbage Collection

Layering transactions on top of a persistent store does not necessarily solve the problems of atomic garbage collection. For the first approach to transactions, a collector for objects accessed by transactions still has to be coordinated with the recovery system. If the collector moves objects, it must record appropriate information in the redo log.

For the second approach no special coordination of the garbage collector with the recovery system for transactions is necessary. Undo information is kept in the persistent heap and the pointers in it will be updated by the collector. However, the collector runs on top of the persistence layer, so the recovery system for persistence treats the copying and scanning of objects just like other modifications. Thus, depending on the implementation, it saves physical undo or redo information, which is an exact copy of the bytes modified by a copy or a scan. In contrast, we integrated our atomic garbage collector with the recovery system for transactions; our collector uses logical redo, which writes small copy and scan records to a log.

Chapter 9

Conclusion

Below we summarize the contributions of this dissertation. Then we suggest ideas for future work.

9.1 Summary

This dissertation has presented an integrated garbage collector and recovery system for an object-oriented transaction system. The garbage collector is atomic and incremental: the individual steps of the collector are recoverable, and the pauses for garbage collection are short and independent of heap size. The time for recovery is also short and independent of heap size, even if a crash occurs in the middle of garbage collection.

To make our collector atomic, we identified the basic steps of the collector, the copy and scan steps, and made them redoable using the write-ahead log protocol, a technique used widely in database recovery systems. By writing to a log the collector avoids writing synchronously to disk; by using the write-ahead log protocol, it also avoids forcing the log. Then we optimized the protocol for copy and scan by reducing the size of the log records that they write. Thus, a transaction waiting for the garbage collector is never delayed by synchronous writes, and the collector writes little information to the log.

We incorporated our collector into a recovery system based on repeating history and write-ahead logging, which uses the same techniques for recovery as our collector. Thus, the optimizations of the recovery system to shorten recovery time, end-write records and checkpoints, also shorten the time for recovery after a crash during garbage collection. Two of the problems that were solved in order to integrate the collector with the recovery

system are: (1) finding roots for collection in the recovery information, and (2) translating addresses in undo information. By delaying the processing of roots in recovery information until the end of the collection, we showed that the collector could avoid most accesses to the log to find these roots. We also showed how the collector and transaction abort could avoid accessing the log to translate addresses in undo information; the collector keeps a translation table in volatile memory, and writes the table to the log at the completion of each garbage collection so that the table is available after a crash.

In the second part of the dissertation, we showed how to do recovery and garbage collection for a stable heap whose roots are partitioned into volatile and stable sets. By dividing the heap into a volatile area and a stable area, our system avoids the cost of atomic garbage collection for volatile objects. By tracking objects as they become reachable from a stable root, our system avoids crash recovery costs for volatile objects. The tracker is concurrent, so that no transaction is delayed by another transaction that has made a large object graph stable.

We also formalized the notion of a stable heap: we presented a specification for it and for the atomic incremental garbage collector and recovery system used to implement it. Formalizing the specification and the implementation helped us to derive the invariants that our algorithms maintain, and increased our confidence in their correctness.

Finally, we implemented a stable heap prototype to show that our algorithms are feasible; we will use the prototype to measure the overheads associated with our algorithms.

9.2 Future Work

As a more advanced feasibility study, we would like to incorporate our atomic incremental garbage collector into an object-oriented database system. The basic algorithm would remain the same, but some changes might be necessary, depending on how the database partitions its data into segments. Instead of collecting the entire database at once, each segment might be independently collectible.

A related topic is the automatic division of the stable area of a stable heap into independently collectible subareas. Subdividing the stable area would reduce the disk storage required for a stable heap. Currently, we require two copies of the heap on disk, one for

from-space and the second for to-space, because we collect the whole stable area every time. By allowing the stable area to be partitioned, we only require extra disk storage for the sub-area that is collected.

A heap can be partitioned automatically by using generations. However, the appropriateness of generational collection for a persistent or stable heap is an open question. Its appropriateness depends on the application: an application such as a recoverable queue that uses a stable heap for fault-tolerance may be very different from a database application that uses a heap for long term storage. Thus, we need to collect information on object lifetimes for many different applications. We can collect some of this information by measuring existing database systems.

In a stable heap, a great deal of information is encoded in inter-object references. If one of these references is lost or corrupted, a whole sub-graph of the stable heap could be lost or corrupted. By using transactions we avoid losing or corrupting these references haphazardly. However, someone might run incorrect transactions anyway. Thus, we need tools, (e.g., an object editor and an object scavenger), to fix a stable heap and to salvage old states of particular objects. Since the log contains the history of all computations, it has all of the information that these tools might require. Determining the functionality of these tools and how they might retrieve information from the log efficiently is an area for future work.

Argus uses a stable heap at each logical node in a distributed system to provide reliable distributed computing, but neither our model of a stable heap nor Argus's model allows one node to directly reference an object on another node using the object's name or address. To do so requires a distributed garbage collector such as Ladin's [33] or Shapiro's [43]. Both Ladin's and Shapiro's collectors separate the collection process into two parts: a distributed garbage collector that determines which objects are globally accessible; and a local garbage collector at each node that collects objects that are no longer accessible locally or globally. A garbage collector for a distributed stable heap could be based on Ladin's or Shapiro's collector and use our atomic garbage collector as the local collector at each node.

References

[1] A. Albano, L. Cardelli, and R. Orsini. A Strongly Typed Interactive Conceptual Language. *ACM Transactions on Database Systems*, 10(2):230–260, June 1985.

[2] Andrew W. Appel and Kai Li. Virtual Memory Primitives for User Programs. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 96–107, April 1991.

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

[4] Henry Baker. List Processing in Real Time on a Serial Computer. *Communications of the ACM*, 21(4):280–294, April 1978.

[5] Joel F. Bartlett. Compacting Garbage Collection with Ambiguous Roots. WRL Research Report 88/2, Western Research Laboratory, Palo Alto, CA, January 1988.

[6] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Publishing Company, Reading, Ma., 1987.

[7] Peter B. Bishop. Computer Systems with a Very Large Address Space and Garbage Collection. Technical Report MIT/LCS/TR-178, Laboratory for Computer Science, MIT, Cambridge, Ma., May 1977.

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

[9] Rodney A. Brooks. Trading Data Space for Reduced Time and Code Space in Real-Time Garbage Collection on Stock Hardware. In *Proceedings 1984 ACM Symposium on Lisp and Functional Programming*, pages 256–262, 1977.

[10] Alfred Brown and John Rosenberg. Persistent Object Stores: An Implementation Technique. In Alan Dearle, Gail M. Shaw, and Stanley B. Zdonik, editors, *Implementing Persistent Object Bases: Principles and Practice/ The Fourth International Workshop on Persistent Object Systems*, pages 199–212. Morgan-Kaufmann Publishers, San Mateo, California, 1990.

[11] Jack Campin and Malcolm Atkinson. A Persistent Store Garbage Collector with Statistical Facilities. Persistent Programming Research Report 29, Department of Computing Science, University of Glasgow, Glasgow, Scotland, 1986.

[12] M. Carey, D. DeWitt, J. Richardson, and E. Sheikta. Object and File Management in the EXODUS Extensible Database System. In *Proceedings of the 12th International Conference on Very Large Databases*, August 1986.

[13] W. P. Cockshott, M. P. Atkinson, K. J. Chisholm, P. J. Bailey, and R. Morrison. Persistent Object Management System. *Software-Practice and Experience*, 14:49–71, 1984.

[14] Robert Courts. Improving Locality of Reference in a Garbage-collecting Memory Management System. *Communications of the ACM*, 31(9):1128–1138, September 1988.

[15] David Detlefs, Maurice Herlihy, and Jeannette Wing. Inheritance of Synchronization and Recovery Properties in Avalon/C++. *IEEE Computer*, 21(12), December 1988.

[16] David L. Detlefs. Concurrent, atomic garbage collection. Technical Report CMU-CS-90-177, Department of Computer Science, Carnegie Mellon University, Pittsburgh, Pa., October 1990.

[17] 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.

[18] John R. Ellis, Kai Li, and Andrew W. Appel. Real-time concurrent collection on stock multiprocessors. Technical Report 25, Systems Research Center, Digital Equipment Corporation, Palo Alto, Ca., February 1988.

[19] D. Gawlick and D. Kinkade. Varieties of Concurrency Control in IMS/VS Fast Path. *Database Engineering*, 8(2):63–70, June 1985.

[20] James N. Gray. Notes on Database Operating Systems. In R. Bayer, R. M. Graham, and G. Seegmuller, editors, *Operating Systems-An Advanced Course*, volume 60 of *Lecture Notes in Computer Science*, pages 393–481. Springer-Verlag, New York, 1978.

[21] James N. Gray, Paul McJones, Mike Blasgen, Bruce Lindsay, Raymond Lorie, Tom Price, Franco Putzolu, and Irving Traiger. The Recovery Manager of the System R Database Manager. *ACM Computing Surveys*, 13(2):223–242, June 1981.

[22] Theo Haerder and Andreas Reuter. Principles of Transaction-Oriented Database Recovery. *ACM Computing Surveys*, 15(4):287–317, December 1983.

[23] Maurice P. Herlihy and Jeannette M. Wing. Avalon: Language Support for Reliable Distributed Systems. In *Proceedings of the Seventeenth International Symposium on Fault-Tolerant Computing*, July 1987.

[24] Elliot Kolodner. Design bug in argus recovery system. DSG Note 158, Laboratory for Computer Science, MIT, Cambridge, Ma., November 1989.

[25] Elliot Ko'odner, Barbara Liskov, and William 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, June 1989.

[26] Elliot K. Kolodner. Recovery Using Virtual Memory. Technical Report MIT/LCS/TR-404, Laboratory for Computer Science, MIT, Cambridge, Ma., July 1987.

[27] Butler. W. Lampson. *Atomic Transactions*, volume 105 of *Lecture Notes in Computer Science*, pages 246–265. Springer-Verlag, New York, 1981. This is a revised version of Lampson and Sturgis's unpublished *Crash Recovery in a Distributed Data Storage System.*

[28] 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.

[29] B. G. Lindsay, P. G. Selinger, C. Galtieri, J. N. Gray, R. A. Lorie, T. G. Price, F. Putzolu, I. L. Traiger, and B. W. Wade. Notes on Distributed Databases. Technical Report RJ2571, IBM Research Laboratory, San Jose, Ca., July 1979.

[30] Barbara Liskov. Overview of the Argus Language and System. Programming Methodology Group Memo 40, Laboratory for Computer Science, MIT, Cambridge, Ma., February 1984.

[31] Barbara Liskov, Mark Day, Maurice Herlihy, Paul Johnson, Gary Leavens, Robert Scheifler, and William Weihl. Argus Reference Manual. Technical Report MIT/LCS/TR-400, Laboratory for Computer Science, MIT, Cambridge, Ma., November 1987.

[32] Barbara Liskov, Paul Johnson, and Robert Scheifler. Implementation of Argus. In *Proceedings of the Eleventh Symposium on Operating Systems Principles*, November 1987.

[33] Barbara Liskov and Rivka Ladin. Highly Available Distributed Services and Fault-tolerant Garbage Collection. In *Proceedings of the 5th ACM Symposium on the Principles of Distributed Computing*, pages 29–39, August 1986.

[34] David Maier, Jacob Stein, Allen Otis, and Alan Purdy. Development of an Object-Oriented DBMS. In *Proceedings of the Object-Oriented Programming Systems, Languages and Applications*, pages 472–482, November 1986.

[35] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. Technical Report RJ6649, IBM Almaden Research Center, San Jose, Ca., January 1989.

[36] David Moon. Garbage Collection in a Large Lisp System. In *Proc. of the 1984 Symposium on Lisp and Functional Programming*, pages 235–246, 1984.

[37] Ron Morrison. PS-algol Reference Manual Fourth Edition. Persistent Programming Research Report 12, Department of Computing Science, University of Glasgow, Glasgow, Scotland, February 1988.

[38] Brian Oki. Reliable Object Storage to Support Atomic Actions. Technical Report MIT/LCS/TR-308, Laboratory for Computer Science, MIT, Cambridge, Ma., May 1983.

[39] Brian Oki, Barbara Liskov, and Robert Scheifler. Reliable Object Storage to Support Atomic Actions. In *Proceedings of the Tenth Symposium on Operating Systems Principles*, pages 147–159, December 1985.

[40] Richard F. Rashid. Threads of a New System. *Unix Review*, 4(8):37–49, August 1986.

[41] Mark Reinhold. Personal communication.

[42] Peter M. Schwarz. *Transactions on Typed Objects.* PhD thesis, Carnegie-Mellon University, December 1985. Available as Technical Report CMU-CS-84-166.

[43] Marc Shapiro, David Plainfosse, and Oliver Gruber. A Garbage Detection Protocol for a Realistic Distributed Object-support System. Rapport de Recherche INRIA 1320, INRIA, November 1990.

[44] Robert A. Shaw. Improving garbage collector performance in virtual memory. Technical Report CSL-TR-87-323, Computer Systems Laboratory, Stanford University, Stanford, Ca., March 1987.

[45] Patrick G. Sobalvarro. A lifetime-based garbage collector for LISP systems on general-purpose computers. Bachelor's Thesis. Massachusetts Institute of Technology. Cambridge, Ma., September 1988.

[46] Alfred Z. Spector, J. J. Bloch, Dean S. Daniels, R. P. Draves, Daniel Duchamp, Jeffrey L. Eppinger, S. G. Menees, and D. S. Thompson. The Camelot Project. *Database Engineering*, 9(4), December 1986.

[47] Satish M. Thatte. Persistent Memory: A Storage Architecture for Object-Oriented Database Systems. In U. Dayal and K. Dittrich, editors, *Proceedings of the International Workshop on Object-Oriented Databases*, Pacific Grove, CA, September 1986.

[48] David Ungar. Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm. In *ACM SIGSOFT/SIGPLAN Practical Programming Environments Conference*, pages 157–167, April 1984.

[49] William Weihl, Butler Lampson, and Jonathan Amsterdam. 6.826 principles of computer systems: Lecture notes and handouts fall 1990. Technical Report MIT/LCS/RSS-14, Laboratory for Computer Science, MIT, Cambridge, Ma., August 1991. Handout 11.

[50] William Weihl and Barbara Liskov. Implementation of Resilient, Atomic Data Types. *ACM Transactions on Programming Languages and Systems*, 7(2):244–269, April 1985.

[51] Gerhard Weikum. A theoretical foundation of multi-level concurrency control. In *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on the Principles of Database Systems*, pages 31–42, Cambridge, Ma., March 1986.

[52] Gerhard Weikum. Principles and Realization Strategies of Multilevel Transaction Management. *ACM Transactions on Database Systems*, 16(1), 1991.

[53] Daniel Weinreb, Neal Feinberg, Dan Gerson, and Charles Lamb. An Object-Oriented Database System to Support an Integrated Programming Environment. Submitted for publication, 1988.

[54] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Effective "static-graph" reorganization to improve locality in garbage-collected systems. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 177–191, June 1991.

[55] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. In *Proceedings of the Eleventh Symposium on Operating System Principles*, pages 63–76, November 1987.

[56] Stanley Zdonik and Peter Wegner. Language and methodology for object-oriented database environments. In *Proceedings of the 19th Annual Hawaiian Conference on Systems Science*, January 1986.

[57] Benjamin G. Zorn. Comparative performance evaluation of garbage collection algorithms. Technical Report UCB/CSD 89/544, Computer Science Division (EECS), University of California, Berkeley, California, December 1989.

Appendix A

Proof Sketch

To prove that the implementation implements the specification, we use induction on the transitions of the implementation. First, we show that the initial state of the implementation corresponds to the initial state of the specification. Then, for each transition of the implementation that corresponds to a transition in the specification, we assume that the implementation begins in a state that corresponds to a state of the specification and that the inputs correspond, and we use the abstraction function and the invariants to show that the state after the transition in the implementation corresponds to the state after the transition in the specification, and the output of implementation corresponds to the output of the specification. Using the same type of argument, we show that the remaining transitions of the implementation have no effect on the state returned by the abstraction function.

To complete the proof, we need to show that the invariants hold: we show for each transition that if the invariant holds before the transition it also holds after the transition.

The implementation throws away state that the specification keeps forever; we use history variables in the implementation so as not to lose this information. The history variables make the abstraction function trivial in many cases. The parts of the abstraction function that are not trivial deal with the `tos`, `WL`, `RL`, `SR` and `VR`. Invariant (1) relates the state of the implementation to the state of the `os` history variable.

Below, we present the inductive argument and then an argument for Invariant (1). Proofs of the remaining invariants are left for the reader.

In the arguments we use the notation `v.pre` where `v` is the name of the variable to denote state of a variable before a transition. Correspondingly `v.post` denotes the state of

v after the transition. When the variable has the same state before and after the transition, no suffix is used.

A.1 Induction

Applying the abstraction function to the implementation's initial state clearly produces the specification's initial state. Below we present the inductive argument for transitions that occur in the specification; then we present the argument that the other transitions produce benevolent side-effects.

A.1.1 Transitions Occurring in Specification

Below we argue the induction for `read`, `write`, `commit`, the first atomic transition of `abort` and `crash`. The remaining operations of the implementation that correspond to operations of the specification, `allocate`, `init_heap`, `start_t`, and `delete_from_VR`, match the specification closely and the arguments for them are trivial.

Read

The `read` operation in the implementation is called on TID `t` and va `a`, and returns state `s`; the corresponding `read` operation in the specification is called on TID `t` and OID `oidmap(a)`. The implementation's `read` operation modifies `VR` and `RL`; the specification also changes its `VR` and `RL`. Thus, we must show that the part of the abstraction function dealing with `VR` and `RL` still holds after the transition, and that the output of the specification is `state2sstate(s)`. There are two cases to argue: (1) transaction `t` holds a write lock, and (2) it does not hold a write lock.

If `t` holds a write lock, the implementation's `read` operation obtains a read lock, it adds the addresses in `(sb ++ vc)(a)` to the `VR`, and it returns the value of `(sb ++ vc)(a)`. The specification's `read` operation obtains a read lock, it adds the oids in `tos(t)(o)` to the `VR`, and it returns the value `tos(t)(o)`. It is clear that the abstraction function for `RL` still holds after this transition. We must show that the right addresses have been added to `VR`, and that the correct state is returned. Applying the abstraction function for `tos` to the implementation's initial state, we get `toTOS(t)(oidmap(a)) = state2sstate((sb`

+ sl + vl)(a)). By Invariant (3), we get (sb + sl + vl)(a) = (sb ++ vc)(a), so the implementation updates VR correctly and returns the right result.

If t does not hold a write lock, the implementation's read operation obtains a read lock, it adds the addresses in (sb ++ vc)(a) to the VR, and it returns the value of (sb ++ vc)(a). The specification's read operation obtains a read lock, it adds the oids in os(o) to the VR, and it returns the value os(o). Again, the abstraction mapping for RL clearly holds. We must show that the right addresses have been added to VR and the correct state is returned. Applying Invariant (1) to the implementation's initial state, we get os(oidmap(a)) = state2sstate((sb + sl + vl - (sl + vl))(a)). The precondition for Invariant (1) holds because a must be in the VR or the SR. By assumption t does not hold a write lock and since t obtained a read lock, no other transaction holds a write lock, so by Invariant (4) (sb + sl + vl - (sl + vl))(a) = (sb + sl + vl)(a). By Invariant (3), we get (sb + sl + vl)(a) = (sb ++ vc)(a). So the implementation updates VR correctly and returns the right result.

Write

The write operation in the implementation is called on TID t, va a, and state s; it modifies WL, vl and vc. The corresponding write operation in the specification is called on TID t, OID oidmap(a), and state state2sstate(s); it modifies WL and tos. The implementation's write operation obtains a write lock, sets vc(a) to s, and puts an update record in the vl. The specification obtains a write lock and sets tos(t)(oidmap(a)) to state2sstate(s). The mapping for WL clearly holds; so we only need to consider tos. Since the implementation appends an update record to vl, we have (sb + sl + vl.post)(a) = s after the transition in the implementation; and (sb + sl + vl.post) unchanged everywhere else. Applying the abstraction function we get toTOS(t)(oidmap(a)) = state2sstate(sb + sl + vl.post)(a), and the abstraction function for tos produces the correct result after the transition.

Commit

The commit operation in the implementation is called on TID t; it modifies WL, RL, active, committed, os, vl, and sl. The commit operation in the specification is also called on TID t; it modifies WL, RL, active, committed, os, and tos. The modifications to WL, RL, active, and committed in the specification correspond exactly to the modifications in the implementation, so they require no further consideration. That leaves os and tos.

We deal first with os. In the specification, for each object o for which tos(t)(o) is defined, os(o) is set to tos(t)(o). In the implementation, for each object at address a, for which transaction t holds a write lock, os(oidmap(a)) is set to (sb ++ vc)(a). Applying the abstraction function for tos(t) to the initial state of the implementation, we get toTOS(t)(oidmap(a)) = state2sstate(sb + sl + vl)(a) for each a for which *transaction t holds a write lock*. Applying Invariant (3), we get (sb + sl +vl)(a) = (sb ++ vc)(a), so the implementation updates os correctly.

Now we deal with tos. Applying the abstraction function for tos after the transition in the implementation for transaction, t, we find that toTOS(t) is the OS function that is undefined everywhere, and toTOS is unchanged for all other transactions. This is exactly the post-transition state for tos in the specification.

Abort

The abort operation in the implementation is called on TID t; its first atomic step corresponds to the abort operation of the specification. The first atomic step modifies active and aborted. The abort operation in the specification is also called on TID t; it modifies active, aborted, tos, WL, and RL. The modifications to active and aborted in the specification correspond exactly to the modifications in the implementation, so they require no further consideration. That leaves tos, RL and WL.

We deal first with tos. The specification sets tos(t) to an OS function that is undefined everywhere, elsewhere leaving unchanged. Applying the abstraction function to the final state of the implementation, we get the same result since the implementation removes t from active.

The arguments for RL and WL are similar; we argue for RL. The specification deletes t

from the read lock set for each object that was read locked by t. Applying the abstraction function to the final state of the implementation, we also get that t is deleted from all the read lock sets because the implementation removes t from active.

Crash

The crash operation of the specification modifies aborted, active, VR, RL, WL, and tos. The crash operation of the implementation modifies aborted, active, VR, RL and WL in the corresponding manner; the only interesting modification is to tos. The specification sets tos to the function that is undefined everywhere. Applying the abstraction function to the final state of crash operation in the implementation we also get that toTOS is a function that is undefined everywhere since crash sets active to the empty set.

A.1.2 Remaining Transitions of the Implementation

We still need to show that the remaining transitions of the implementation that do not correspond to transitions of the specification produce benevolent side effects. For each such transition, we need to show that the result of the abstraction function applied to the starting state equals the abstraction function applied to the final state. The interesting transitions are the last atomic step of abort, the atomic steps of gc, truncate_log, and the atomic steps of undo_tran. The other two transitions, flush_log and flush_cache, do not affect any component of the abstraction function.

Last Atomic Step of Abort

The last atomic step of abort appends an abort record to the vl and releases locks for t from RL and WL. Since t was already deleted from active before this transition, this has no effect on toRL, toWL, or toTOS.

GC

We need to show that the three atomic transitions of gc are each benevolent side effects. They are flip, copy and scan.

Flip Two procedures are called by `flip`: first `scan_locks` and then `scan_roots`. We discuss them one at a time.

`scan_locks` calls copy and modifies RL and WL. copy modifies `a_allocated`, `oidmap`, `vc`, and `vl`. These modifications could only affect the parts of the abstraction function that deal with SR, VR, RL, WL, and tos. Since `scan_roots` is called in the same atomic step, we deal with SR and VR in the discussion of `scan_roots`. The argument for RL is similar to WL, so we discuss WL and tos below and leave RL for the reader.

Consider an object at address a for which `WL(a) = {t}` before the call to `scan_locks`. Assume also that t is in `active` and that `s = (sb + sl + vl.pre)(a)`. (If t is not in `active` or `WL(a)` is empty, the result returned by toWL and toTOS is not affected). Suppose that when `scan_lock` calls copy, the object at a is copied to b. Then after the `scan_lock` transition, we have `oidmap(a) = oidmap(b)`, `WL(b)={t}`, `WL(a)` is empty, and `(sb + sl + vl.post)(b)=s`. Thus, neither the result returned by toTOS or toWL is affected by the transition.

Now we consider `scan_roots`. `scan_roots` modifies SR and VR, and calls copy. The arguments for VR and SR are the same; we argue for SR. Calls to copy could affect the parts of the abstraction function that deal with RL, WL, and tos. However, all of the objects in SR that were read-locked or write-locked before `flip` were already copied as part of the `scan_locks` transition; we showed above that toRL, toWL, and toTOS return the same results before and after `scan_locks`. When copy copies objects that are not locked, it cannot affect RL, WL, or tos.

For each object a in `SR.pre` there is an object b in `SR.post` such that `oidmap(a) = oidmap(b)`. Also, `SR.pre` and `SR.post` are the same size. Thus, the result returned by toSR after `scan_roots` is the same as the result returned before `scan_roots`.

Copy Next we consider copy operations called after the `flip`. Suppose that copy is called for the object at address a, `a.space = to_space - 1`, and the object is copied to b. (If the object had been copied previously, copy does not change any implementation state.) copy modifies `a_allocated`, `oidmap`, `vc`, and `vl`. These modifications could only affect the parts of the abstraction function that deal with RL, WL, tos, SR, and VR. However, the guards on

the read, write and allocate operations ensure that after the flip no additional object can be locked or added to VR unless it is already in to-space. Thus, any copy that occurs after the flip is a benevolent side-effect.

Scan Last we consider scan. Suppose that scan is called for the object at address a, and a.space = to_space. scan modifies vc, vl, scanned and calls copy. The only part of the abstraction function that could be affected is tos. Suppose s = (sb + sl + vl.pre)(a) before scan and t = (sb + sl + vl.post)(a) after the scan. scan calls copy on each of the objects referenced by s. We just showed above that any copy after the flip is a benevolent side-effect. Since copy updates the oidmap we have state2sstate(s) = state2sstate(t), and the result returned by toTOS is not affected.

Truncate_log

truncate_log modifies sl. The only part of abstraction function on which it could have an effect is tos. By Invariant (3) sb + sl + vl = sb ++ vc, and the invariant holds both before and after transition. Since neither sb or vc are modified by the transition, the result returned by toTOS is unaffected by truncate_log.

Undo_tran

There are two atomic transitions in undo_tran. The first one does not change any of the implementation's global state, so it does not affect any component of the abstraction function.

The second one modifies vc, vl and scanned. Because the transaction for which undo_tran is called is never in active, the second transition does not affect any component of the abstraction function either.

A.2 Invariant (1)

Invariant (1) relates the os history variable to the sb, sl and vl. Transitions of the implementation that do not affect at least one of these state variables trivially maintain the invariant. The other transitions are allocate, write, commit, the last atomic step

of abort, crash, copy, scan, flush_log, flush_cache, truncate_log and undo_tran. Below we argue that each of these transitions maintain Invariant (1).

In the arguments we rely on the other invariants, so these need to be shown without relying on Invariant (1). That exercise is left for the reader.

A.2.1 Allocate

Allocate creates a new object that is made accessible by inserting it in the VR. The base_commit record written for the object to vl ensures that the invariant holds.

A.2.2 Write

Write does not add to the set of objects that satisfy the accessibility predicate. For the object that it modifies, it appends an update record to vl. The undo information in the record cancels the redo information in it, so if Invariant (1) holds before write, it also holds after write.

A.2.3 Commit

Before commit for t, Invariant (1) holds. During commit no new object becomes accessible, but the os changes state for those objects for which t holds a write lock. Thus, we only need to consider these objects. After commit, os(oidmap(a)) = (sb ++ vc)(a) for each object modified by t, where a is the address of the object. By Invariant (3), (sb ++ vc)(a) = (sb + sl + vl)(a). Since no other transaction holds a write lock on an object locked by t, Invariant (4) tells us that (sb + sl + vl)(a) = (sb + sl + vl - (sl + vl))(a) and Invariant (1) still holds.

A.2.4 Last Atomic Step of Abort

The last transition of abort appends an abort record to vl. This has no effect on the invariant since undoactive does not look at abort records.

A.2.5 Crash

Both invariants (1) and (2) hold before crash. Combining them we derive a property that also holds before crash: commit_accessible(a) ==> os[oidmap(a)] =

state2state((sb + sl - sl)(a)). The crash operation itself is logically divided between in two. The first part models a crash; it adds the transactions in active to aborted, and then discards vl, vc, VR, and active. The second part models recovery. The derived property holds after the first part of crash since the state on which it depends, i.e., sb and sl, is not affected. The only objects satisfying the accessible predicate after the first part of crash are those that satisfied the commit_accessible predicate before crash. Since vl is empty, invariant (1) holds after the first part.

In the second part of crash, the only records appended to vl are abort and compensation log records for transactions that were active just before crash. This work is done by the abort operation, and we have shown that each of the atomic transitions of abort preserve Invariant (1).

A.2.6 Copy

Suppose copy is called for the object at address a, accessible(a) holds, and the object is copied to address b. (Otherwise, the invariant holds trivially.) After copy, accessible(a) no longer holds, so the invariant no longer concerns a. After copy, accessible(b) holds, so we need to show that the invariant holds for b. There are two cases : (1) no transaction holds a write_lock on b, and (2) a transaction holds a write_lock.

First Case

If no transaction holds a write_lock, then no transaction held a write_lock on a before copy. By Invariant (4), we have (sb + sl + vl.pre - (sl + vl.pre))(a) = (sb + sl + vl.pre)(a) before copy. Then after copy, the copy record written to the log ensures that (sb + sl + vl.pre)(a) = (sb + sl + vl.post)(b). Since oidmap(a) = oidmap(b), Invariant (1) holds.

Second Case

The copy record ensures that (sb + sl + vl.pre)(a) = (sb + sl + vl.post)(b). It also ensures that translate(a, sl + vl) = b, so (sb + sl + vl.pre - (sl + vl.pre))(a) = (sb + sl + vl.post - (sl + vl.post))(b). Since oidmap(a) =

oidmap(b), Invariant (1) holds.

A.2.7 Scan

scan replaces the from_space addresses in an object by the corresponding to_space addresses. Assume scan is called on the object at address a, and accessible(a) holds. (Otherwise, the invariant holds trivially.) scan calls copy on each of the objects referenced by (sb + sl + vl.pre)(a), and we just showed that each copy operation preserves the invariant. Since copy updates the oidmap and writes copy records to the vl, we have state2sstate((sb + sl + vl.pre - (sl + vl.pre))(a)) = state2sstate((sb + sl + vl.post - (sl + vl.post))(a)). Thus, Invariant (1) holds.

A.2.8 Flush_log

An atomic transition of flush_log appends vl to sl, and then empties sl. Thus we have sl.pre + vl.pre = sl.post + vl.post, and if Invariant (1) held before the transition, it also holds after the transition.

A.2.9 Flush_cache

An atomic transition of flush_cache modifies sb and vc, such that sb.pre ++ vc.pre = sb.post ++ vc.post. By Invariant (3), we have sb ++ vc = sb + sl + vl; thus, sb.pre + sl + vl = sb.post + sl + vl. So if Invariant (1) held before the transition, it also holds after the transition.

A.2.10 Truncate_log

An atomic transition of truncate_log removes a prefix of sl, but again by Invariant (3) it must be the case that sb + sl.pre + vl = sb + sl.post + vl. So if Invariant (1) held before the transition, it also holds after the transition.

A.2.11 Undo_tran

The first atomic transition does not change sb, sl, or vl. The other atomic transition appends a compensation log record (clr) to the vl that undoes the update recorded in an update record in sl + vl. If the invariant held before the transition, it also holds after since

the new state recorded in the clr record is just the state that **undoactive** would have restored using the old state in the update record (i.e., **`translate_state(update.old_state) = clr.new_state`**).

END