Implementing Orthogonal Persistence: A Simple Optimization Based on Replicating Collection

Scott Nettles

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

James O'Toole

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

Abstract

Orthogonal persistence provides a safe and convenient model of object persistence. We have implemented a transaction system that supports orthogonal persistence in a garbage collected heap. In our system, replicating collection provides efficient concurrent garbage collection of the heap. In this paper, we show how replicating garbage collection can also be used to reduce commit operation latencies in our implementation.

We describe how our system implements transaction commit. We explain why the presence of non-persistent data can add to the cost of these operations. We show how to eliminate these additional costs by using replicating garbage collection. The resulting implementation of orthogonal persistent should provide transaction performance that is independent of the quantity of non-persistent data in use. We expect efficient support for orthogonal persistence to be valuable in operating systems applications which use persistent data.

Authors' addresses: nettles@cs.cmu.edu, otoole@lcs.mit.edu

This research was sponsored by the Avionics Lab, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U. S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, Arpa Order No. 7597, by the Air Force Systems Command and the Defense Advanced Research Projects Agency (DARPA) under Contract F19628-91-C-0168, and by the Department of the Army under Contract DABT63-92-C-0012.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

1 Introduction

Systems in which arbitrary data structures can be made persistent are becoming increasingly important. Such systems form the basis of both persistent programming languages and object-oriented databases. In such systems an important design choice is deciding which objects should be persistent. For both safety and programmer convenience, the most desirable choice is orthogonal persistence [1]. We have built a system that supports orthogonal persistence. In this note we discuss a simple but significant optimization in its implementation.

In a system with orthogonal persistence, an object is persistent if it is reachable by dereferencing pointers starting from a distinguished object, the persistent root. Tracing garbage collectors also use reachability to determine which objects must be retained. Consequently, it is natural to use techniques related to garbage collection to implement orthogonal persistence.

We have used copying garbage collection techniques to implement orthogonal persistence as part of a general purpose multi-threaded transaction system. In closely related work [7] we show how a new garbage collection technique, replicating collection, can be used to provide a simple efficient concurrent garbage collector for our system. This same technique can be used to significantly improve the performance of our implementation of orthogonal persistence.

In the sections that follow we first present our basic implementation and the performance problem it introduces. We then present our solution to this problem and briefly discuss its implementation. The sections that follow assume some familiarity with the technique of copying garbage collection.

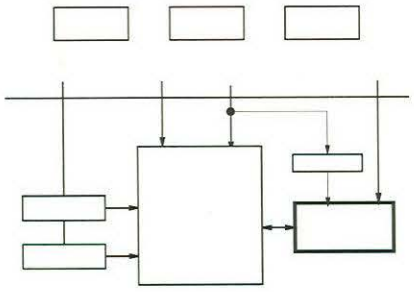

Figure 1: The Transactional Heap Interface

2 The Problem

Figure 1 shows the basic interface to our system. The system supports the operations: read, write, allocate, abort and commit. Objects that are reachable from the persistent root at commit are guaranteed to survive program failures. Objects that are only reachable from the transitory root are lost upon program failure. A concurrent replicating garbage collector provides storage reclamation both of transitory and persistent objects. To support abort, persistence, and generational and replicating collection the location and old value of each write operation is recorded in a write log. Although our system supports multiple clients, this aspect is not relevant to the current subject and will not be further discussed. The commit operation has primary responsibility for maintaining object persistence and its implementation is the focus of the remainder of this section. For more details about our system see O'Toole et al [7].

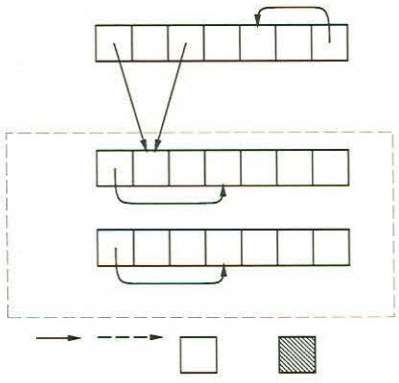

Figure 2: A Committed State

Figure 2 shows a detailed view of the key components of our system when it is in a committed state. Permanent objects are stored in the persistent heap; all other objects are stored in the transitory heap. The persistent heap is composed of two images: the stable image, which is stored on disk and which holds the committed image of the heap, and the volatile image, which is found in main memory and which holds any uncommitted data. The client reads and writes the volatile image. In a committed state all objects reachable from the persistent root must be found in the persistent heap. Thus in a committed state no pointers may point from the persistent heap into the transitory heap. Pointers may point from the transitory heap into the volatile image.

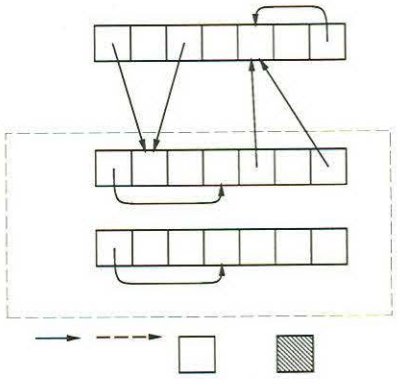

Figure 3: Before Commit

Figure 3 shows the system when uncommitted data is present. Assignments have created pointers from the volatile image into the transitory heap. Commit must guarantee that any object that is now reachable from the persistent root is in the volatile image and that the stable image is atomically updated to reflect all changes to the volatile image. The only changes to the stable image are those explicitly requested by the system. Other details about of how the stable image is updated are irrelevant to this discussion.

We assume that all pointers from the volatile image into the transitory heap refer to objects that are now reachable from the persistent root. (This is a conservative assumption.) The system traverses its write log to identify such pointers and uses them as the roots of a copying garbage collection. This collection moves all objects that are reachable from these roots into the volatile image. Figure 4 shows the state of the system after the collection has been done and the stable image updated. The cost of updating the persistent heap is proportional to the number of writes and the amount of data that must transferred to the volatile and stable images.

However, the work of the commit operation is not yet complete. Figure 4 shows that there remain pointers in the transitory heap that refer to objects that have been moved into the volatile image. An obvi-

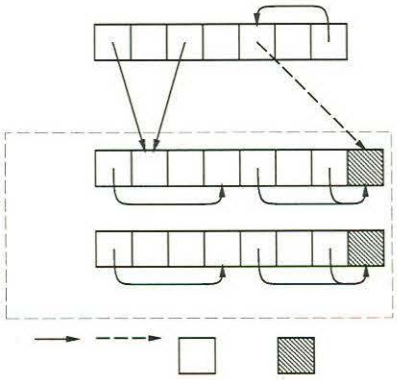

Figure 4: Before Scan

ous way to ensure that all such pointers are updated properly is to scan the entire transitory heap. The forwarding pointers left by the copying collection allow these pointers to be identified and updated. Figure 5 shows the result of such a scan. Another option is to immediately garbage collect the transitory heap; during the collection these references will be redirected to the copies in the volatile image.

Both of these method add a cost to commit which is proportional to the total size of the transitory heap. Unfortunately, we know of no way to selectively track the pointers in the transitory heap that will require updating at the time of transaction commit. It seems inevitable that the cost of updating these pointers will depend on the size of the transitory heap. Yet, we would like the cost committing a single transaction to be independent of the amount of transitory data. Ideally, the latency of an individual commit operation should depend only on the number of write operations performed and the amount of data that must transferred to the volatile and stable images.

3 The Solution

To solve this problem we must perform the commit operation without immediately updating the transitory heap. The key insight is that the semantic requirement of the commit operation is that the stable image must contain the committed data. There is no fundamental requirement that the stable and volatile images be identical, nor that the transitory heap be updated. However, if we delay updating the transitory heap pointers, then we must also avoid updating the pointers in the volatile image. This suggests the following commit strategy:

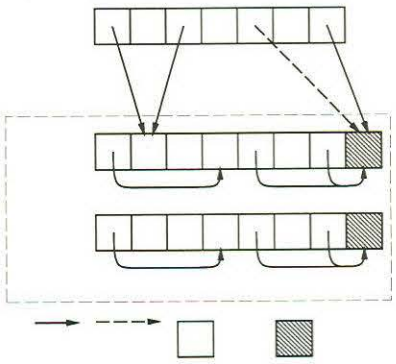

Figure 5: After Scan

1. Do the commit up to the point of the scan, but retain enough information to reverse the effects of this step.
2. Update the stable image using the volatile image.
3. Rollback the effects of the first step.

There are several problems with this strategy. One problem is that typically copying collectors destroy the original version of the object that they are copying by overwriting it with a forwarding pointer. However it is not enough to simply repair this damage during rollback. The problem is that subsequent commits will need to modify the stable image in a manner consistent with the current placement of data in the stable image. Thus it is necessary to retain complete placement information in the volatile image.

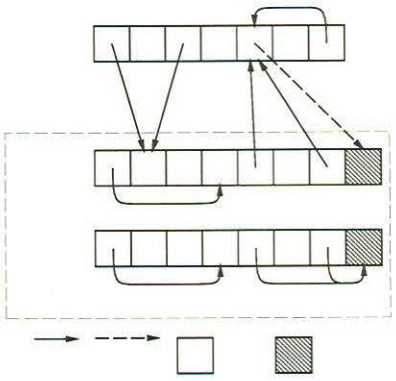

Figure 6: After Rollback

To do this, we use replicating garbage collection. Replicating collection was originally developed for incremental and concurrent garbage collection [5, 6]. Replicating collection allows the client to continue using the original objects in the transitory heap and pre-

serves the required placement information so that it can be used by subsequent commit operations.

The key idea of replicating collection is to perform the basic copy operation non-destructively. Non-destructive copying requires that the object not be overwritten by the forwarding pointer. The existence of more than one potentially valid copy of mutable object implies the possibility that the copies might become inconsistent. Our system addresses this object consistency issue in the following way:

- The client reads and writes the original version of the object.
- All write operations are recorded in a log.
- When convenient, the system reads the log and applies the writes to the new version of the object.
- Clients are permitted to switch to the new version of the object only if all log entries have been processed.

This method enables our system to replicate objects but bring the replicas "into service" with the client at a later time.

This technique is ideally suited to solving the problem at hand. After the replicating collection is completed, the changes to the volatile image are written to the stable image. Then the rollback step is carried out by setting the values of the roots (modified locations in the volatile image) back to the values they had at the start of the collection. Figure 6 shows the state of the system after this step.

After the commit operation has been completed, the client continues to use the original objects in the transitory heap. Although the replicating collection is non-destructive, it does leave forwarding pointers to the objects it copied, it simply does not overwrite the original object with them. Subsequent commit operations will not recopy these objects because they are already marked with forwarding pointers that indicate that the object has been moved into the persistent heap. The write operation logging used by replicating collection will ensure that the replicas are kept up to date.

Later, when the system eventually garbage collects the transitory heap, all of the pointers in the transitory heap will be updated just as in Figure 5. At that time, the pointers in the volatile image that were reset to their original values will also be updated.

4 The Implementation

We are currently implementing this optimization in our system. The implementation is straightforward because the system already includes all of the mechanism needed to support replicating collection. To support the rollback step of this optimization and the associated bookkeeping, the following changes to the implementation are required:

- When the commit operation updates a root pointer, the location and old value of this pointer is saved in a log. This *commit log* has the same format as the write log already in use by the system.
- When the commit operation has updated the stable image, the commit log is used to restore the pointers into the transitory heap. Existing support for rollback makes this easy.
- The write log processing code that supports replicating garbage collection is extended to create transaction log entries when it reapplies write operations to replicas of persistent objects.
- After a transitory heap garbage collection is completed, the commit log is used to redo the pointer updates that were undone in the rollback step.

Because of space limitations as well as their highly technical nature, we have omitted a complete discussion of some difficult modifications that involve coordinating this optimization with the operations of the concurrent persistent heap garbage collector.

We plan to measure the performance of the new system soon. We expect the performance improvement due to this optimization to be substantial when the transitory heap is large. Because the extra bookkeeping required by this optimization is not expensive, we expect performance to be good even if the transitory heap is small.

5 Related Work

The presentation of this optimization has been closely tied to the implementation of our system. Because of the ease of implementing orthogonal persistence using copying collection we expected this issue to arise in other systems. For example, Kolodner [3] discusses this problem in the context of his Argus design, but does not offer a satisfactory solution. We are unaware of any other solution.