Concurrent Garbage Collection of Persistent Heaps

Scott Nettles James O'Toole[†] David Gifford[†]

April 1993

CMU-CS-93-137

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

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

Also appears as MIT-LCS-TR-569. Submitted to *The 14th ACM Symposium on Operating Systems Principles.*

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, by 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.

Keywords: persistence, transaction processing, garbage collection, copying garbage collection, concurrent garbage collection, replication copying garbage collection

Abstract

We describe the first concurrent compacting garbage collector for a persistent heap. Client threads read and write the heap in primary memory, and can independently commit or abort their write operations. When write operations are committed they are preserved in stable storage and thus survive system failures. Clients can freely access the heap during a garbage collection because a replica of the heap is created by the *stable replica collector*. A log is maintained to capture client write operations. This log is used to support both the transaction system and the replication-based garbage collection algorithm.

Experimental data from our implementation was obtained from a transactional version of the SML/NJ compiler and modified versions of the TPC-B and OO1 database benchmarks. The pause time latency results show that the prototype implementation provides significantly better latencies than stop-and-copy collection. For small transactions, throughput is limited by the logging bandwidth of the underlying log manager. The results provide strong evidence that the replication copying algorithm imposes less overhead on transaction commit operations than other algorithms.

1 Introduction

Persistent programming environments and object repositories share the need to determine when to deallocate storage so that it can be reused for another purpose. Automatic storage management techniques that offer both high performance access and low interference with client operations are of key importance in these applications.

Three techniques are known for storage reclamation: explicit deallocation, reference counting garbage collection, and tracing garbage collection. Each technique has its disadvantages. The explicit deallocation of storage by programmers can result in dangling pointers and space leaks, and automatic reference counting causes overhead on every pointer creation and destruction operation, and can recover cyclic structures only with great difficulty. Tracing garbage collection offers the most attractive storage reclamation alternative, but existing systems either pause the client program while garbage collecting or do not maintain a heap image that is resilient to system failure.

We have designed and implemented the first compacting garbage collector which permits clients to manipulate a stable heap that is being simultaneously garbage collected. Controlled experiments show that a *stable replica collector* can substantially improve client access to a stable heap when compared with a stop-and-copy collector. The stable replica collector adds little overhead to the execution time of client programs, and significantly reduces the maximum pause times imposed by garbage collection. Based upon these and other experimental measures it appears this system offers a practical approach to storage management in persistent programming environments, object repositories, and similar applications.

The basic idea behind a stable replica collector is to allow clients to directly read and write a stable heap, and to concurrently copy live data from the stable heap into a new stable heap. Clients are free to mutate portions of the heap that have already been copied because the system maintains a mutation log. The log is used by the garbage collector to ensure that the new stable heap contains all pertinent client updates before is it used to replace the old stable heap. The log is processed concurrently which limits garbage collection pause times to brief synchronization points.

Our programming model assumes multiple clients are simultaneously performing allocation, read, and write operations on a shared persistent heap. Sharing of data among clients is controlled by the clients. Every object mutation is part of a transaction, and transactions can either be committed or aborted. The persistent heap is always restored to its most recently committed state if a system failure occurs.

The external interface of our system is shown in Figure 1. From the programmer's perspective the major components of the system are the stable heap and the storage manager.

Client operations on the stable heap are performed on an image that resides in primary memory. Mutations to the image are also recorded in the mutation log by code automatically produced by a compiler. The in-line mutation logging instructions are the source of the overhead placed on the client by stable replica collection.

The storage manager is responsible for providing both transaction and heap compaction

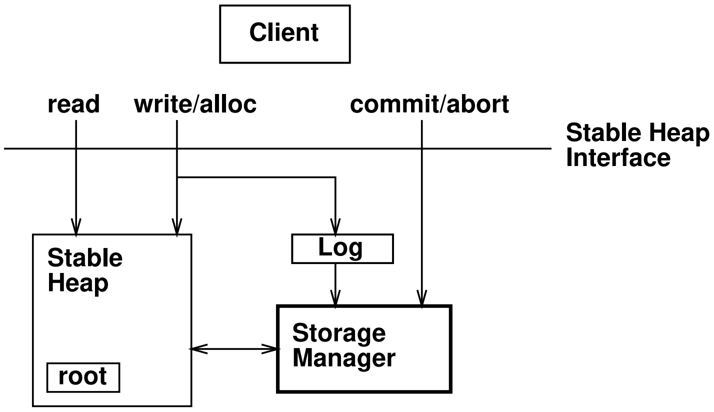

Figure 1: The Transactional Heap Interface

services to multiple clients without disrupting client computations. The storage manager ensures that all heap objects that are reachable from the *persistent root* are retained in the heap, and that unreachable objects are discarded. The storage manager further ensures that all committed mutations are recorded in the stable heap. The stable heap logically only consists of the results of committed transactions. The storage manager learns of mutations to the heap by reading the mutation log.

The remaining five sections discuss the previous work that we have built on (Section 2), describe stable replica collection and the design of the storage manager in detail (Section 3), present an implementation of stable replica collection in Standard ML (Section 4), summarize our experimental results with the implementation (Section 5), and discuss elaborations and applications of the basic algorithm (Section 6).

2 Previous Work

We are aware of only one other implementation of a concurrent collector for a persistent heap, that of Almes [1]. Designed and implemented for use by the Hydra OS for C.mmp, it is a mark-and-sweep collector based on Dijkstra's concurrent mark-and-sweep algorithm [10]. There are two key differences between this work and our own. First, it cannot relocate objects, thus it offers no opportunities for either heap compaction or clustering of objects for fast access. Second, while it works in the context of persistent objects, it is not designed for use in a system with transaction semantics. Because of the rather unusual environment in which it ran, it is also difficult to make any performance comparison between it and our work.

There is a long history of incremental and concurrent copying collectors dating back to Baker[5]. Essentially all of these collectors require the client to access the to-space version of an object during collections. The technique of Ellis, Li, and Appel[3] enforces this restriction by using virtual memory protection to force clients to use only to-space values. Our technique does not require any unusual operating systems support, nor does it constrain the order in which objects are copied as does Ellis, Li, and Appel. We believe that the ability to freely choose the order in which objects are copied and traversed is especially important in a system which may need to optimize access to the disk. As of yet we have not explored this possibility.

Garbage collection algorithms based on replication appear in [15, 16], and an instance of the basic technique is described in by Huelsbergen and Larus[11]. Related work on concurrent collection appears in [6]. The basic literature on uniprocessor garbage collection techniques is surveyed in [18], and discussions of persistent heaps and language support for transactions appears in [4, 13].

There are two earlier designs of concurrent garbage collectors for persistent heaps. Both of these designs are based on the concurrent garbage collection algorithm of Ellis, Li, and Appel. Detlefs[9] described how to apply this algorithm in a C++ environment which included transactions. Kolodner[12] pioneered the work on atomic garbage collection and partially implemented an atomic and concurrent version of the same algorithm. In Detlefs' design, the programmer must explicitly manage data persistency at the time of object allocation, whereas Kolodner specified that persistence of data was determined by its reachability from a set of persistent roots.

Neither of these designs were completely implemented, owing, we believe, to their complexity. Both designs focus on making steps of the garbage collection process recoverable. This is essential in their designs because the transaction system must be able to commit data which the collector is currently manipulating. We believe an important advantage of our design is that it allows the transaction manager to be decoupled from the garbage collector to a much greater extent than these previous designs. This has implications both on ease of implementation and on performance. In our current implementation collections themselves are not recoverable, in the event of a crash they must be restarted. This is not a fundamental limitation of our approach, and we describe an algorithm for garbage collector recovery that our implementation could employ.

The unique contributions of our work include:

- A method of performing compacting garbage collection concurrently on a persistent heap. None of the previous attempts to address this problem have been fully implemented.

- The decoupling of transaction processing from the garbage collection of persistent storage. Unlike existing designs, our approach allows a separate Transaction Manager and Garbage Collector to share a log without necessarily burdening transaction commit delays.

3 Design

The interface to a persistent heap consists of three client operations: allocate a block of storage, read a word of storage, and write a word of storage. Unlike a volatile heap that is destroyed upon system failure, a persistent heap must survive any type of hardware failure and be properly recovered to its last committed state. A committed state consists of the cumulative effect of all committed update operations. Storage that is not reachable from the persistent root is automatically deallocated concurrent with client access to the heap, and reachable storage is compacted at regular intervals to improve performance.

Our persistent heap design admits a wide variety of commit models. At one extreme we can commit each write individually so upon restart all completed writes will be represented. We can also group updates into transactions, and upon failure recover a persistent heap contents that represents the results of all committed transactions. In this section we present a transaction based commit model, but we will describe the invariants that a commit model needs to guarantee to work with our storage manager design.

We will present the design of the storage manager as a series of three refinements to a basic design. Each refinement will either add functionality or performance to the basic implementation:

- *Basic Design: Replication Based Collection on Stable Spaces.* Our initial design uses a replication based collector on stable spaces. Clients operate on *stable from-space*, and the collector concurrently copies from-space into *stable to-space*. In this initial design each write is individually committed, and the client must access stable storage for every read and write operation.

- *Refinement 1: Transactions Group Updates.* Our first refinement adds transactions to the design. Space flips are independent of the lifetime of transactions, and thus the transaction undo log must be preserved across space flips.

- *Refinement 2: Volatile Images Improve Performance.* Our second refinement adds volatile main-memory images of from-space and to-space to improve the performance of client programs and the collector. Committed from-space operations must be recovered upon failure. Thus it is necessary for the transaction manager to ensure that all committed operations are recorded in stable from-space.

- *Refinement 3: Generations Reduce Stable Heap Collection Frequency.* Our third and final refinement adds a *transitory heap* to keep short-lived objects out of the persistent heap. All objects are initially allocated in the transitory heap, and are automatically promoted to the persistent heap when they are made reachable from the persistent root. The transitory heap is garbage collected with respect to the *transitory root*, and is not recovered upon failure.

3.1 Replication Based Collection on Stable Heaps

If the stable heap semispaces are stored in stable storage, then the basic replication copying garbage collection algorithm [15] provides a persistent heap with a concurrent compacting garbage collector. High speed stable storage could be implemented using random access memory, suitably protected by an battery backed power supply. The persistent root is

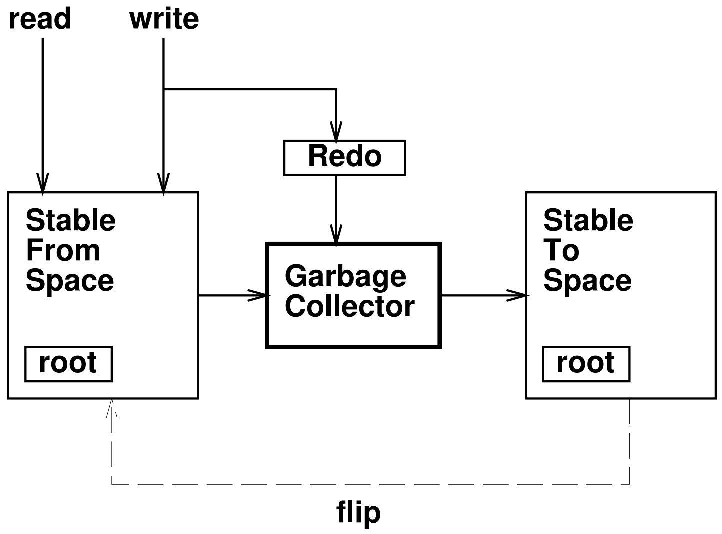

Figure 2: Replication Collection of a Non-Transactional Heap

stored in the heap as shown in Figure 2. In this design each mutation is individually committed and becomes permanent as soon as it is performed.

As shown in Figure 2, replication based garbage collection allows clients to freely operate on from-space while the garbage collector creates a replica of from-space in to-space. Other concurrent copying collectors require that the client access only to-space objects, and a read barrier is used to ensure that the client does not see any pointers to from-space. Our approach is based on a *from-space invariant* which ensures that the client can access only from-space objects.

Clients record their write operations in a *redo log* that is read by the garbage collector to bring to-space up to date before a space flip occurs. The invariant that the collector satisfies is that at the instant a space flip occurs the reachable storage in to-space is isomorphic in address and identical in other content to the storage in from-space. Reachable storage is defined as any object that can be reached from the persistent root pointer.

Upon failure the client can be immediately restarted on stable from-space without any recovery processing. The stable nature of from-space guarantees it will survive failures and thus no content recovery is necessary. Upon crash recovery all that is necessary is to locate the stable from-space in memory. Because the storage backing from-space and to-space is renamed on space flips at least one bit of information must be stored to identify the current representation of the stable from-heap. The act of updating the from-space identifier and free pointer is atomic, and defines the actual moment of space flip. The from-space free pointer must be updated in stable storage on every allocate unless it is recovered by scanning from-space on restart.

Upon failure the garbage collector can be resumed if sufficient state is recorded in stable storage. To enable garbage collector resumption the to-space free pointer, the to-space scan

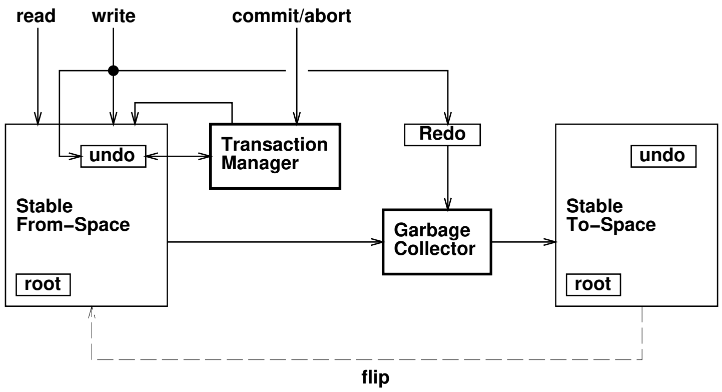

Figure 3: A Transactional Heap

pointer, and the redo log must be in stable storage. These items will collectively be called the garbage collector state. Alternatively the garbage collector state can be kept in volatile storage, and the garbage collector can be restarted with an empty to-space upon failure.

3.2 Transactions Group Updates

Figure 3 shows how transactions can be added to the initial design. In this model, each update operation is part of a transaction, and transactions can be independently committed or aborted. In Figure 3, all write operations are directly applied to from-space, and an *undo log* describing uncommitted mutations is maintained. If a transaction is aborted, all of its updates are undone by using the undo log. If a transaction is committed, then its writes are atomically removed from the undo log.

As shown in Figure 3 the undo log is maintained in the heap and is reachable from the persistent root. Thus the undo log is moved from from-space to to-space as a normal consequence of garbage collection. Carrying the undo log across space flips allows flips to occur when transactions are in progress. The undo log may refer to otherwise unreachable data structures, and these data structures will be preserved across space flips. These otherwise unreachable data structures must be preserved in the event that the transaction that made them unreachable aborts.

Upon recovery clients can be resumed on stable from-space once the undo log found in the heap is applied. Undoing the mutations in the undo log ensures that all uncommitted updates are erased before clients resume processing.

Upon recovery the garbage collector can be resumed if the garbage collector state has been preserved in stable storage.

3.3 Volatile Images Improve Performance

Figure 4 shows how images of from-space and to-space can be maintained in fast volatile memory. In this scheme clients read, write, and allocate in a volatile image of from-space,

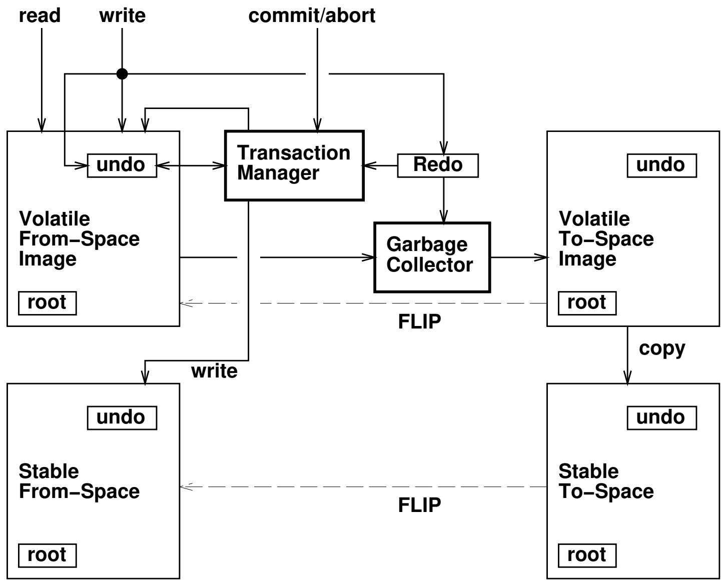

Figure 4: Using Volatile Images

and the garbage collector reads the from-space image and writes the to-space image.

When a transaction commits, its updates to volatile from-space must be made stable. This is accomplished by keeping a redo log for each transaction and using it to identify portions of the from-space image that have been updated. Upon commit the transaction manager writes the updated portions of the from-space image to stable from-space. The updates to stable from-space must be performed as an atomic act in order to guarantee a consistent version of stable from-space in the presence of failures.

The garbage collector directly writes the to-space image, and a background process copies the to-space image onto a slow stable storage media. Thus a slow stable storage medium can be used for to-space without a major performance reduction of the garbage collector. The key invariant that is maintained is that at a space flip stable to-space is identical to the volatile to-space image, all of from-space has been copied, and the redo log is empty. The flip causes to-space to replace both stable from-space and the volatile from-space image.

Upon recovery clients can be resumed on the from-space image once it is restored from stable from-space and the undo log is applied.

Upon recovery the garbage collector can be resumed if its state has been preserved in stable storage.

3.4 Generations Reduce Stable Heap Collection Frequency

Figure 5 shows how a single generation of transitory heap can be added in front of the stable heap. The transitory heap has its own replication based garbage collector that copies data

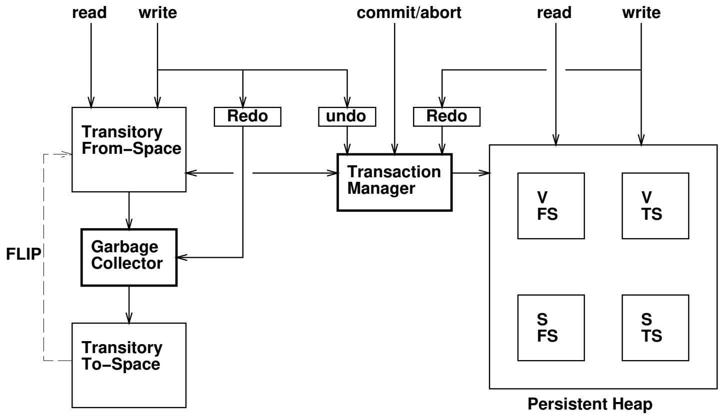

Figure 5: Using a Transitory Heap

reachable from the transitory root.

All objects are initially created in the transitory heap and are promoted to the stable heap when they become reachable from a persistent object. Promotion to the stable heap is performed by the transaction manager during commit processing. When a transaction that has written a pointer to a transitory object into the stable heap commits, the corresponding transitory object is promoted. The transaction manager uses the redo log at commit time to detect such updates to the persistent heap.

Upon restart the client is restarted on the recovered persistent heap using the algorithm outlined in the previous section. The transitory heap is initialized to be empty. Thus upon restart the transitory garbage collector begins with an empty heap and does not need to be restarted.

4 Implementation

The prototype implementation provides concurrent compacting garbage collection within the Standard ML of New Jersey runtime system. We added a single-threaded transaction manager to the runtime system and reimplemented the garbage collector using the concurrent replication algorithm. The purpose of the prototype implementation was to test the feasibility of using concurrent replication to garbage collect the stable heap.

Standard ML of New Jersey (SML/NJ) is a type-safe programming language implementation which includes an optimizing compiler, a runtime system, and a garbage collector[2]. We chose to test our garbage collection algorithm in the SML/NJ programming environment primarily for reasons of convenience. The SML/NJ source code is freely available and is amenable to modifications for experimental purposes. Previous work on persistence[17] and replication-based garbage collection[16, 15] using ML provided us with several of the components needed for our prototype.

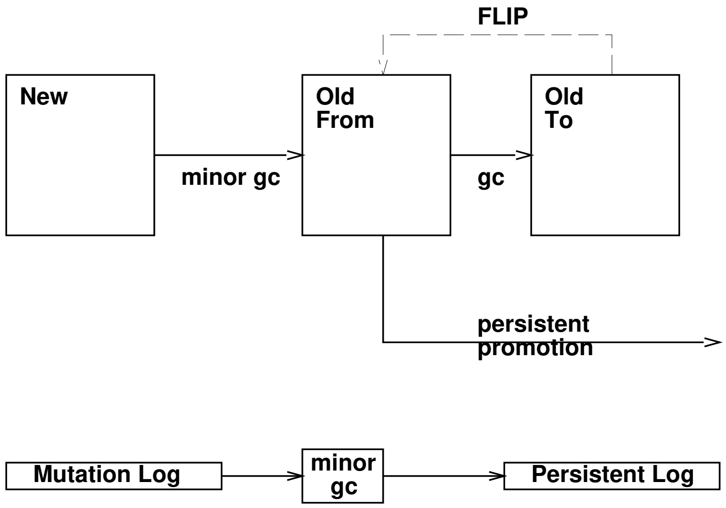

Figure 6: The Transitory Heap

4.1 The Transitory Heap

The transitory heap consists of the two generational heaps present in the SML/NJ implementation. It contains temporary data which will be lost in the event of a crash. The transitory heap is maintained independently of the stable replica collector. Figure 6 shows the primary data structures used by the runtime system to maintain and garbage collect the transitory heap:

- **New-space** — All objects are allocated and initialized by the client in this space, including programming language structures such as environment frames, control frames, continuations, and procedures. The programming language implementation does not use a stack, and the allocation rate in new-space is therefore very high.

- **Old-From** — A minor garbage collection moves all reachable objects in new-space into old-from space. New-space is normally quite small, and is garbage collected very frequently. Short-lived objects do not survive a minor collection.

- **Old-To** — A major garbage collection moves all reachable objects from old-from space into old-to space. When the collection completes, the two spaces exchange roles.

- **Mutation log** — The client records mutations it performs in a log. The code necessary to implement the logging operations is emitted by the compiler. Log records are allocated from new-space in the form of a linked list. This log is used by the transitory heap garbage collector to locate mutated objects in old-from space which point into new-space and thus must be used as roots for minor collections. Because it only needs to track mutations which might create such cross heap references, the SML/NJ system logs only pointer mutations.

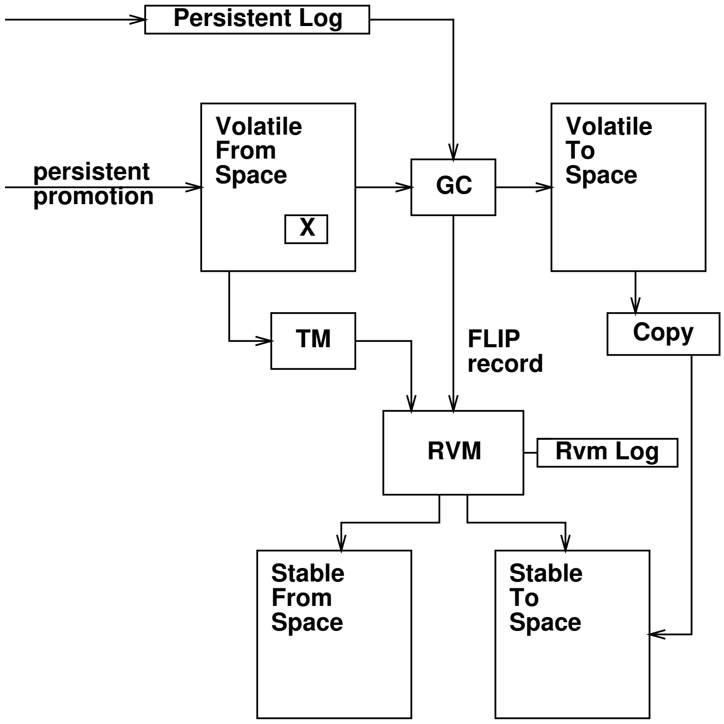

Figure 7: The Persistent Heap

- **Persistent log** — When the mutation log is processed by the transitory heap collector, it filters the log for records which identify mutations to objects in the persistent heap. These log records are moved to the persistent log. The persistent log serves as an undo/redo log for the transaction manager, and also as a redo log for the stable replica collector.

We modified the SML/NJ compiler to generate code which performs the mutation logging required for our system, all mutations are logged. The transitory heap garbage collections are performed using the replication-based garbage collector described in [15]. The persistent log is used by the transitory collector to locate objects in the transitory heap which are reachable from objects in the stable heap. When the transitory heap is collected, these objects are relocated, and the references stored in the stable heap are updated by the transitory heap collector.

4.2 Operations on the Persistent Heap

The persistent heap is maintained both in virtual memory and on stable storage through the cooperation of the transaction manager and the stable replica collector, as described in Section 3. Figure 7 shows an implementation of the design shown in Figure 5. The implementation uses the Recoverable Virtual Memory (RVM) implementation described

in [14]. RVM represents the stable semi-spaces on disk using two data files and a log file containing redo records; each committing transaction atomically updates the log file, which is applied asynchronously to the data files.
The prototype contains three threads of control:

- The **Client** thread executes the application program and periodically commits transactions by executing as the Transaction Manager. The current Transaction Manager implementation does not operate concurrently with the client. The Client thread also traps into the runtime system when it must synchronize with the garbage collector and perform a flip.

- The **Collector** thread executes the replication garbage collector algorithm, copying reachable objects from the volatile from-space into the volatile to-space. When the Collector thread has constructed a complete replica of the volatile from-space, it signals the Client thread that a flip should take place.

- The **Copy** thread writes the contents of volatile to-space onto the stable to-space, either directly to the disk or via the log manager.

Commit

When the client commits a transaction, it calls into the Transaction Manager, which performs the following operations synchronously:

1. Garbage collect the new-space transitory heap. While not strictly required, this is fast, simplifies the promotion of newly persistent objects, and ensures that the client mutation log has been filtered and the relevant transaction log entries have been moved to the persistent log.

2. Complete any concurrent garbage collection of old-space if one is in progress.

3. Scan the transaction log to locate references to objects in the transitory heap which must be promoted to the persistent heap.

4. Promote all newly persistent objects based on their reachability from the persistent log. This is done using the same basic copying algorithm as the collectors.

5. Scan the persistent log as a redo log to bring replicas of modified objects in the volatile to-space up-to-date. This work is done on behalf of the persistent garbage collector so that the persistent log can be discarded.

6. Empty the persistent log and atomically update the stable from-space by logging and committing an RVM transaction containing modifications to volatile from-space.

7. Update all of the transitory heap (old-space) objects to point to the newly promoted persistent objects.

Of the seven steps performed during commit in our prototype, all but steps 5 and 6 are a consequence of using a transitory heap and providing the data persistent model of object persistence by reachability. Step 5 is necessary because a replication copying collection may be active, but this work need not occur during commit. In the prototype it is convenient to discard the log at commit time, so the gc related log processing is performed as part of commit. Step 6 maintains the invariant relationship between the stable heap and the volatile image by atomically updating the contents of the stable heap and its recovery log.

Persistent GC Flip

When a commit adds enough new data to the volatile from space to cross a predetermined threshold, a garbage collection of the persistent heap is initiated. At this point the Collector performs these steps:

1. Copy all objects in volatile from-space which are reachable from the persistent root into volatile to-space.
2. Copy all objects in volatile from-space which are reachable from the transitory heap into volatile to-space.
3. Scan the persistent log as a redo log to bring replicas of modified objects in the volatile to-space up-to-date.
4. Copy the volatile to-space into stable to-space.
5. Halt the client.
6. Update the client's roots to point to volatile to-space.
7. Update all of the transitory heap (Old-space) to point to volatile to-space instead of volatile from-space.
8. Write the flip record via RVM to indicate that the stable to-space is now the stable from-space

Steps 1–3 are performed asynchronously by the Collector thread. Step 4 is performed by the Copy thread. The Copy thread is used to perform the actual disk writes because the Client thread occasionally synchronizes with the Collector thread to manipulate shared data structures, and it is undesirable to have the Collector thread be blocked on disk write operations at those moments.

When steps 1–4 are complete, the Collector halts the Client thread in order to access to the current register values and process the most recent entries in the mutation log. The most recent mutations to objects in volatile from-space are propagated to volatile to-space and any resulting changes to volatile to-space are written to stable to-space, including the recent entries on the current transaction's undo log. These final updates to stable to-space are currently performed synchronously by our implementation, but they need not be. The final updates of stable to-space could be written asynchronously with the flip record. The flip record labels the stable to-space as the most recent committed semi-space, and must be delivered to the log manager and sequenced into the log ahead of any subsequent client commit operations which occur after the flip.

The Copy thread can write stable to-space either via the RVM log manager or by writing directly to the disk file. It is convenient to use the log manager, but we found that the added RVM log traffic interfered with client commit operations (see Section 5.3). When the Copy thread bypasses the RVM log and writes directly to the to-space disk file, it must first ensure that RVM's log contains no pending updates to the to-space. The Copy thread does this by requesting RVM to empty its log by applying all pending updates.

Recovery

When a crash occurs and the system restarts, most of the recovery processing is handled by RVM. The recovery algorithm is as follows:

1. RVM applies pending committed redo records to bring the from-space and to-space files to their most recently committed state.
2. The garbage collector examines the flip records stored in the stable semi-spaces to determine which disk file contains the active heap.
3. The transaction manager uses the persistent log stored in the committed heap as an undo log to reverse the uncommitted operations of any partially complete transactions. In the prototype the undo recovery log stored in the committed heap will be non-empty only if a crash occurs shortly after the stable heap is flipped, because the transaction manager in use supports a single client thread of control.

To maintain the persistent heap in a recoverable state, the stable replica collector ensures that the transaction undo log is preserved in the volatile to-space at the time of a flip. In the prototype, this is accomplished very easily because the transaction manager stores the log in the volatile from-space. Therefore, it is preserved in volatile to-space by the garbage collection algorithm.

5 Performance

We designed and ran a series of experiments to test the thesis that our algorithm reduces the interference caused by garbage collection pauses and reduces the overall cost of storage management when compared with a stop-and-copy approach. The experiments compare our implementation of a concurrent collector to our implementation of a stop-and-copy collector for the same environment. We are unable to directly compare our work to other concurrent collector designs.

Our experiments demonstrate that a stable replica collector interferes with a client less than a stop-and-copy collector. Stable replica collector pauses are of the same general magnitude as fast commits. For heap sizes in the megabyte range, the pause times achieved by our technique are a factor of eight shorter than stop-and-copy collection. We estimate that for larger heaps, such as those which might be found in a production object database, the difference would be even greater. We are also able to demonstrate definite performance advantages deriving from our approach.

Although it is not the focus of our current work, we also measured the commit performance of our system. The commit performance of the system depends mostly on the choice of transaction model, the particulars of the transactions, and the performance of the underlying log manager. We measured the transaction performance primarily to show how good and bad the commit pauses can be, given the simple log representation used by the system.

Throughout this section we attempt to consider the design implications of our results. We reflect on the issues with respect to our own work, the design of log managers which support garbage collection, the implementation of persistence models and the implementation of other designs for concurrent collection of persistent heaps.

5.1 Benchmarks

Three benchmarks were used to test our implementation. Each was chosen to measure and stress different aspects of our system. Two of the benchmarks performed a significant number of garbage collections, while the third was used to measure transaction throughput. Although we did not measure recovery performance we did crash (sometimes inadvertently!) and recover each of our benchmarks to verify that they were indeed recoverable.

- The *Compiler* benchmark is the Standard ML of New Jersey compiler compiling a portion of the SML/NJ implementation. We modified the compiler to store all of its data in the persistent heap and commit its state every time a module (file) was compiled, modeling the behavior of a persistent programming environment. This 100,000 line program is compute intensive and contains long-running transactions.

- The *TP-OO1-V* benchmark is a variant of the OO1 Engineering Database benchmark described in [7]. We implemented the algorithm described as OO1 in order to have a representative object-oriented database application. However, the OO1 benchmark, as specified, does not require garbage collection, so we added deletion operations to the benchmark to make it a more realistic application for our system.

- The *TPC-B* benchmark performs a large number of bank teller operations which perform transfers among various bank accounts. This benchmark is our implementation in Standard ML of the TPC-B benchmark from [8].

All benchmarks were executed on a Silicon Graphics 4D/340 equipped with 256 megabytes of physical memory. The clock resolution on this system is approximately 10 milliseconds. The machine contains four MIPS R3000 processors clocked at 33 megahertz. Each processor has a 64 kilobyte instruction cache, a 64 kilobyte primary data cache, and

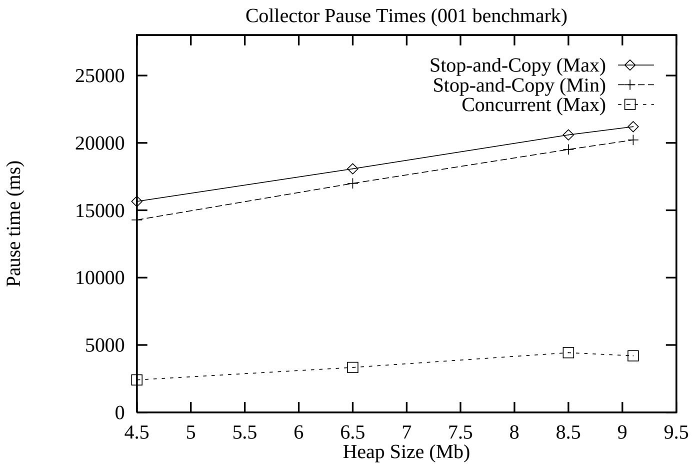

Figure 8: Flip Pauses vs. Heap Size

a 256 kilobyte secondary data cache. The secondary data caches are kept consistent via a shared memory bus watching protocol, and there is a five word deep store buffer between the primary and the secondary caches.

5.2 Methodology

All benchmarks were executed using a transaction manager which implements a "transactional process" model. Each time the application requests a commit, its entire process state is committed to the persistent heap, including processor register contents. We chose to run the benchmarks using this model because it generally minimizes the size of the transitory heaps and maximizes the workload on the stable heap.

5.3 Pause Times

The most important property of our collector is that the pause times it imposes on the client are short and bounded. Very large stable heaps imply very large stop-and-copy garbage collection pauses. We wanted our prototype to work well enough to demonstrate the usefulness of short pause times in configurations where transaction commit pauses are present.

We ran the OO1 Engineering Database benchmark in Standard ML with varying amounts of live data in the heap. This allowed us to measure the duration of collection pauses as a function of heap size. Figure 8 shows the maximum length pauses caused by collection of the persistent heap as a function of heap size for both the concurrent and stop-and-copy collectors as well as the minimum length pauses for the stop-and-copy collector. (The minimum pause for the concurrent collector is to small to measure.) Even for the modest

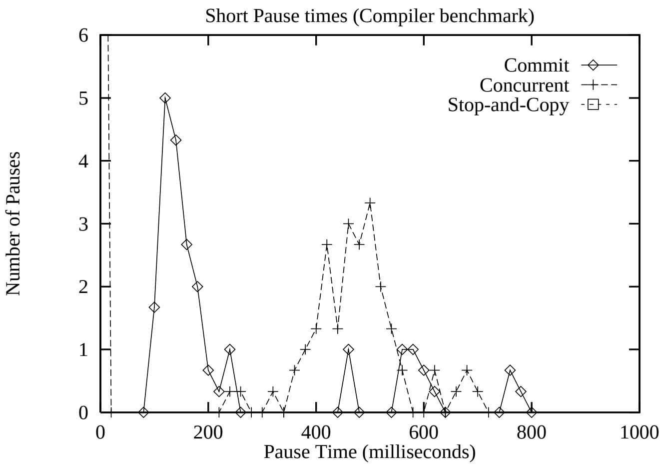

Figure 9: Distribution of All Pauses (<1 second)

heap sizes used here, the pauses created by the stop-and-copy collector are probably not acceptable.

In contrast, the maximum pauses for the concurrent collector are almost all below 500 milliseconds. However, we were disappointed by the maximum pause time results of the concurrent collector because the remaining outlying pause times are an order of magnitude higher. We investigated the source of these pauses and found that they were all caused due to the final writes of volatile to-space to stable storage, which are performed synchronously in the current implementation. More careful coordination between the Copy and the Collector threads will allow us to initiate the final writes of volatile to-space asynchronous but release the Client thread before these writes complete. This change should eliminate these delays.

To explore how collection pauses compared to commit pauses, we ran the *Comp* benchmark using both the concurrent collector and the stop-and-copy collector. We measured both commit pauses and pauses from both collectors. The results are shown in Figures 9 and 10. Figure 9 shows only the pauses below 1 second. The remaining pauses are shown in Figure 10, except for a long tail of commit pauses which extends to 53 seconds. The commit pauses shown here were produced using the concurrent collector, and are essentially the same as those produced by the stop-and-copy collector.

As shown in Figure 9, the longest pauses due to concurrent collection are comparable to the typical commit pause, and are much smaller than the pauses due to stop-and-copy collection. Most of the stop-and-copy pauses appear in Figure 10. We investigated the source of the longer concurrent collector pauses and found that for this test the typical reason was that the post-flip log truncation was still active when a new collection needed to be initiated. This is partially a product of the use of a small collection threshold, which forces frequent collections. These pauses could be eliminated if RVM allowed us to

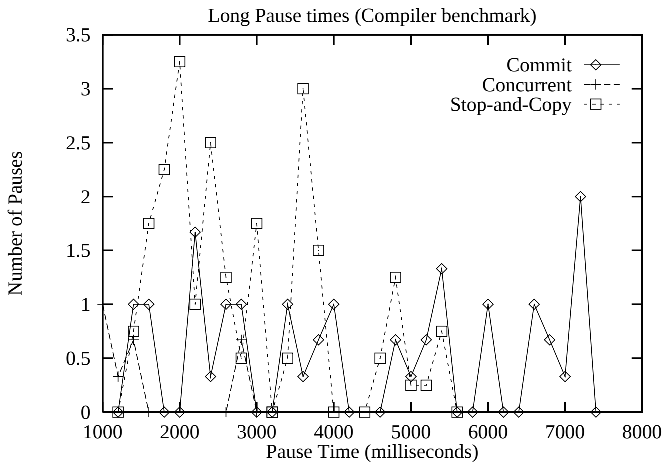

Figure 10: Distribution of All Pauses (continued >1 second)

invalidate all from-space data in the log upon a flip. Instead, the current implementation must wait for RVM to apply all pending log records. Adding this feature to RVM could yield a substantial performance improvement as well, because truncating the RVM log is an expensive operation. However, even with the current RVM interface we believe that more careful coordination between the truncation and the collector thread will allow this problem to be avoided.

While studying the performance of the prototype we also ran the compiler benchmark using a configuration in which all collector modifications of to-space were logged using RVM. In other words, the log manager was used to manage to-space in addition to from-space. In this configuration, the extra log traffic from the garbage collector process introduced substantial additional commit delays. In the concurrent replication algorithm, the write traffic from the garbage collector need not slow down the commit traffic from the client because the writes are to different spaces. A suitable modification to RVM might make it possible for us to take advantage of this fact by allowing some transactions to move through the log independent of others. In an Appel, Ellis, Li-style algorithm, where the garbage collector and the client both write to-space, we expect that it would be difficult or impossible to avoid interference between the two. This is because both the client and the collector are modifying the same spaces and much more careful coordination is needed between the two. We believe that this coordination is a source of substantial complexity in other designs.

5.4 Transaction Throughput

In addition to measuring the pause related behavior, we were also interested in examining the throughput of our system. Two questions were asked. First, could the system provide reasonable performance for small simple transactions and second, can concurrent replication

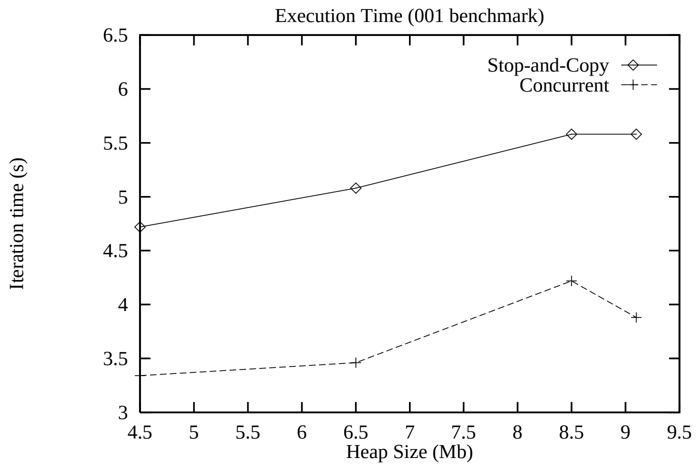

Figure 11: Modification Elapsed Time

copying provide increased throughput for systems which garbage collect. The answer to both of these questions is yes.

In order to test the fast paths for transaction commit, we executed the TPC-B program using a 12 megabyte database which contained bank account records for 1 branch covering 100,000 customers. During this test, our implementation ran 14 transactions per second. The limiting factor was the synchronous disk write required by each transaction. Note that this database size is smaller than that stated in the TPC-B guidelines. We believe that we have complied with all other requirements of the benchmark standard.

We measured the cost of the various phases of commit processing, including separate measurements of the promotion of committed data from the transitory heap to the stable heap, the garbage collection of the transitory heap, the logging of stable heap modifications through the RVM interface, and the commit of the RVM transaction to update the stable heap. For very small transactions as in TPC-B, essentially all of the delay during commit occurs when committing the RVM transaction, because RVM must synchronously write to the disk.

To study the effect of our system when garbage collection is needed we used our modified version of OO1. We measured the elapsed time to perform the standard engineering modification described by the benchmark with the addition of the deletion of one hundred parts per transaction. This allowed the transaction to maintain a constant heap size since one hundred insertions were also performed. Figure 11 shows the elapsed time to perform the transaction as a function of heap size. The use of concurrent collection is clearly advantageous.

5.5 Improving Stable Storage Access

For large transactions, such as in the compiler benchmark, we examined the detailed components of both commit and collection closely. Our examination shows that a significant fraction of the time is spent processing the log, primarily the cost of logging these mutations through RVM. This suggests several possible modifications which should improve commit performance or reduce total overhead or both.

Many of the mutations on the log are to the same location. Compressing the log to eliminate these entries would reduce the logging cost substantially. We make a call to RVM for each entry on the log. RVM must validate each such call independently. An interface which allowed a group of related modifications to be batched would reduce this expense greatly. We have discussed the addition of such an interface with the designers of RVM.

RVM delays capturing the value of a redo record until the transaction is is part of commits. In general this is the desired semantics, but for our application it would be acceptable to capture the value when the modification was first logged. This would allow RVM to be more aggressive about the use of write ahead logging. Another use of write ahead logging which would be beneficial would to be more eager about promoting newly persistent values to the persistent heap. Currently no promotions are done until commit. Early promotion would allow the cost to be absorbed in existing collection work.

We have not touched upon the issue of the cost of recovery . In our prototype implementation there are two phases of recovery processing. First, the RVM log manager must recover the last committed physical heap image. We have not studied RVM recovery performance. Second, the transaction manager must undo any uncommitted modifications which are present in the persistent heap. Neither of our benchmarks uses multiple client threads, so the only time there is uncommitted data in the heap is immediately after a garbage collector flip.

The recovery processing of the prototype works and has been tested. Our experience with recovery makes it obvious that essentially the entire time spent doing recovery is attributable to RVM reconstructing the committed contents of the stable heap from its log structures. Most of the time spent by RVM doing recovery and log-truncation operations during the operation of our collector could be avoided if the persistent garbage collector could inform RVM when a heap semi-space is being recycled.

6 Future Work

We expect to reduce the current pause times of the concurrent collector by improving the synchronization among the various threads so that the client thread is never blocked waiting for synchronous disk activity generated by the garbage collector. We also plan to experiment with a few simple optimizations that may greatly compress the mutation logs. Our current prototype system uses a single-threaded transaction manager, but we expect to be able to add support for a variety of transaction semantics, including multiple client threads, using [17]. One lesson learned from the performance measurements is that there are several desirable features which are candidates for addition to the RVM log manager.

The current implementation does not support the restart of a partially completed garbage collection, but the changes required to our implementation are minimal. Because the client uses only from-space, the recovered state of to-space is of little importance. The state of the replication garbage collection algorithm can be recovered as long as its volatile data structures are periodically flushed to stable storage.

We also expect that replication copying collection will prove to be valuable for very large persistent heaps which are accessed using varying internal and external representations (swizzling), via a cache, or in a distributed programming environment containing multiple volatile heaps. The primary advantages of replication copying collection in these configurations is that the client and the garbage collector need not be as tightly coupled as in other garbage collection algorithms, and system features which depend upon knowledge of object mutations and object locations are therefore easier to build.

7 Conclusions

We have implemented the first concurrent compacting garbage collector for a transactional persistent heap. The design is based on replication garbage collection, and uses a mutation log which is shared with the transaction manager. The prototype implementation demonstrates that client activity can continue during the garbage collection of stable data.

The experimental measurements show that the concurrent algorithm offers garbage collection pauses which are much shorter than a stop-and-copy collection, and largely independent of stable heap size. Although transaction performance is limited by the underlying log manager implementation, the algorithm promises garbage collection which is usable in real-time operating systems applications requiring large pointer-rich persistent storage.

Acknowledgements

Jim O'Toole and Scott Nettles thank the DEC Systems Research Center for support as summer interns in 1990, at which time replication-based garbage collection was originally conceived. Thanks also to our readers Brian Reistad and Franklyn Turbak.

References

[1] Guy T. Almes. Garbage collection in a object-oriented system. Technical Report CMU-CS-80-128, Carnegie Mellon University, June 1980.

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

[3] Andrew W. Appel, John R. Ellis, and Kai Li. Real-time Concurrent Garbage Collection on Stock Multiprocessors. In *SIGPLAN Symposium on Programming Language Design and Implementation*, pages 11–20, 1988.

[4] M. P. Atkinson, K. J. Chisolm, and W. P. Cockshott. PS-Algol: an Algol with a persistent heap. *SIGPLAN Notices*, 17(7):24–31, July 1982.

[5] H. G. Baker. List Processing in Real Time on a Serial Computer. *Communications of the ACM*, 21(4):280–294, 1978.

[6] Hans-Juergen Boehm, Alan J. Demers, and Scott Shenker. Mostly Parallel Garbage Collection. In *SIGPLAN Symposium on Programming Language Design and Implementation*, pages 157–164, 1991.

[7] R. G. G. Cattell. An engineering database benchmark. In Jim Gray, editor, *The Benchmark Handbook for Database and Transaction Processing Systems*, pages 247–281. Morgan-Kaufmann, 1991.

[8] Transactions Processing Council. Tpc-b. In Jim Gray, editor, *The Benchmark Handbook for Database and Transaction Processing Systems*, pages 79–114. Morgan-Kaufmann, 1991.

[9] David L. Detlefs. Concurrent, atomic garbage collection. Technical Report CMU-CS-90-177, Carnegie Mellon University, October 1990.

[10] E. Dijkstra, L. Lamport, A. Martin, C. Scholten, and E. Steffens. On-the-fly Garbage Collection:An Exercise in Cooperation. *Communications of the ACM*, 21(11):966–975, November 1978.

[11] Lorenz Huelsbergen and James R. Larus. A concurrent copying garbage collector for languages that distinguish (im)mutable data. In *Proceedings of the 1993 ACM Symposiym on Principles and Practice of Parallel Programming*, 1993. To appear.

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

[13] B. Liskov and R. Scheifler. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Transactions on Programming Language and Systems*, 5(3):382–404, July 1983.

[14] Hank Mashburn and M. Satyanarayanan. RVM: Recoverable virtual memory. Note in progress, March 1991.

[15] Scott M. Nettles and James W. O'Toole. Real-Time Replication Garbage Collection. In *SIGPLAN Symposium on Programming Language Design and Implementation*. ACM, June 1993.

[16] Scott M. Nettles, James W. O'Toole, David Pierce, and Nicholas Haines. Replication-Based Incremental Copying Collection. In *Proceedings of the SIGPLAN International Workshop on Memory Management*, pages 357–364. ACM, Springer-Verlag, September 1992.

[17] Scott M. Nettles and J.M. Wing. Persistence + Undoability = Transactions. Technical Report CMU-CS-91-173, Carnegie Mellon University, August 1991.

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