Lazy Reference Counting for Transactional Storage Systems

Miguel Castro, Atul Adya, Barbara Liskov

Laboratory for Computer Science,
Massachusetts Institute of Technology,
545 Technology Square, Cambridge, MA 02139
{castro,adya,liskov}@lcs.mit.edu

Abstract

HAC is a novel technique for managing the client cache in a distributed, persistent object storage system. In a companion paper, we showed that it outperforms other techniques across a wide range of cache sizes and workloads. This report describes HAC's solution to a specific problem: how to discard indirection table entries in an *indirect pointer swizzling* scheme. HAC uses *lazy reference counting* to solve this problem. Instead of eagerly updating reference counts when objects are modified and eagerly freeing unreferenced entries, which can be expensive, we perform these operations lazily in the background while waiting for replies to fetch and commit requests. Furthermore, we introduce a number of additionaloptimizations to reduce the space and time overheads of maintaining reference counts. The net effect is that the overhead of lazy reference counting is low.

Keywords: reference counting, pointer swizzling, transactions, caching, performance

1 Introduction

HAC is a novel technique for managing the client cache in a client-server, persistent object storage system [4]. It combines page and object caching to reduce the miss rate in client caches dramatically while keeping overheads low. HAC provides improved performance over earlier approaches — by more than an order of magnitude on common workloads [4]. The contribution of this report is to document *lazy reference counting*: the technique used by HAC to solve the specific problem of discarding indirection table entries in an *indirect pointer swizzling* scheme.

Pointer swizzling [11] is a well-established technique to speed up pointer traversals in object-oriented databases. For objects in the client cache, it replaces contained global object names by virtual memory pointers, thereby avoiding the need to translate the object name to a pointer every time a reference is used. Pointer swizzling schemes can be classified as *direct* [13] or *indirect* [11, 10]. In direct schemes, swizzled pointers point directly to an object, whereas in indirect schemes swizzled pointers point to an indirection table entry that contains a pointer to the object. Indirection allows HAC to efficiently evict cold objects from the cache and compact hot objects to avoid storage fragmentation — when an object is evicted or moved, it is not necessary to find and correct all pointers to the object; instead, we can just fix the object's entry in the indirection table. However, it introduces the problem of reclaiming storage in the indirection table.

Reference counting [5] is a simple incremental automatic storage reclamation technique. It maintains a count of the number of references pointing to an item and reclaims the item when its count drops to zero. Existing reference counting schemes suffer from four problems [14] — space overheads, the problem of cycles, the cost of reclaiming the unreferenced entries, and the cost of changing reference counts at every pointer update. Our lazy reference counting scheme overcomes these problems. First, it uses only one-byte reference counts (with larger counts stored in a separate table). Second, the inability to collect cyclic garbage using reference counts is not a problem here since evicting objects from the cache breaks cycles. Third, freeing

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-91-J-4136. M. Castro is supported by a PRAXIS XXI fellowship.

an unreferenced entry is achieved by simply setting a bit in the entry (reusing the entry is similarly inexpensive). Finally, reference counts are *not* corrected at every pointer update but only once at the end of a transaction to account for all of the transaction's modifications. Furthermore, lazy reference counting performs reference count adjustments and frees unreferenced entries in the background while the client is waiting for replies to fetch and commit requests.

This report describes how lazy reference counting is implemented in HAC [4] and presents the results of experiments that evaluate its performance. The results show that the lazy reference counting scheme has low space and time overheads.

Even though we focus on the implementation of lazy reference counting in HAC, the algorithm and most of the optimizations we introduce could be applied to solve other related problems. For example, assume a client cache management scheme where direct swizzling is used and pages are evicted by read protecting the virtual page frame where they were mapped. In this setting, lazy reference counting can be used to determine when there are no more pointers to the virtual page frame, allowing it to be reused to map a different page.

The rest of the report is organized as follows. Section 2 presents an overview of our system and object naming scheme. Section 3 describes lazy reference couting. The results of our experiments are presented in Section 4.

2 System Overview

HAC and lazy reference counting are part of Thor-1, a new implementation of the Thor object-oriented database system; more information about Thor-1 can be found in [4]. This section begins with a brief overview of the client and server architectures. Next, it describes object naming and indirect swizzling.

2.1 Clients and Servers

The system is made up of clients and servers. Servers provide persistent highly available storage for objects. Clients cache copies of persistent objects. Application transactions run entirely at clients, on the cached copies. More information about the client interface to the database can be found in [8]. We serialize transactions using optimistic concurrency control; the mechanism is described in [1, 7]. Our concurrency control algorithm outperforms the best pessimistic methods on almost all workloads [7].

HAC partitions the client cache into page frames and fetches entire pages from the server. To make room for an incoming page, HAC selects some page frames for *compaction*, discards the cold objects in these frames, and compacts the hot objects to free one of the selected frames. The approach is illustrated in Figure 1. This compaction process is performed in the background while waiting for the fetch reply.

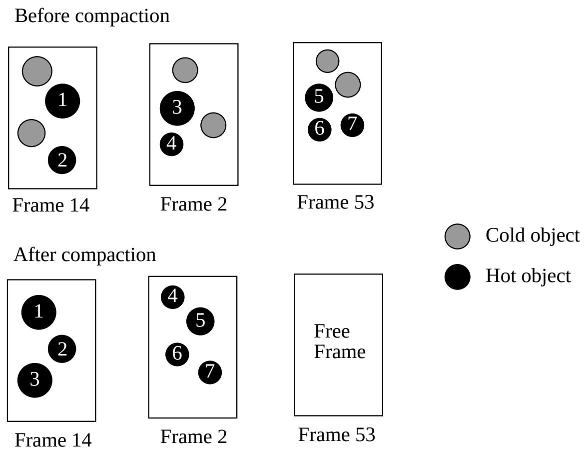

Figure 1: Compaction of pages by HAC

While a transaction runs, the client keeps track of objects it used (in the *read* set), and objects it modified (in the *write* set). Additionally, when a transaction modifies an object for the first time, the client makes a copy of the object in the *undo* log. These copies are used to restore modified objects to their previous state when transactions abort. The copies are important because they speed up the re-execution of aborted transactions. This mechanism is essential for our optimistic concurrency control scheme to outperform locking alternatives [7]. The space dedicated to holding the undo log has a limited size; when it fills up, no more copies are created. We expect a small buffer to be capable of holding copies of all modified objects for most transactions [7].

When a transaction commits, the client sends the new versions of all objects modified or created by the committing transaction to a server, along with the names of objects that were read and written during the current transaction. The server validates the transaction against those that have already committed or prepared (we use a two-phase protocol if the transaction used objects at multiple servers). If validation succeeds, information about the committing transaction, including all its modifications is forced to the transaction log; only after the information has been recorded on stable storage does the server inform the client of the commit.

When a transaction that modified object o commits, it causes all other clients that cache o to contain out of date information. We cause invalid information to be flushed from caches by sending *invalidations* piggybacked on other messages already being sent.

2.2 Object References

Objects stored on disk refer to one another using *names*. The name of an object identifies the disk page where the object is stored, but not the location of the object's data within that page. This is important because it allows storage management at servers to avoid fragmentation by moving objects within their pages independently from other clients or servers, since no other server or client knows the actual location where an object resides.

Using *names* as references is not appropriate for the objects cached at the client, because every dereference requires a translation of the name to the location of the cached object. Instead we use an *indirect* form of *lazy swizzling on discovery* [13]; references are swizzled to point to an entry in an indirection table that points to the object's fields. Swizzling is performed when a reference is loaded from an instance variable into a local variable. This is implemented using *edge marking* [10]; references are marked as swizzled or unswizzled using one bit. When a reference is loaded from an instance variable into a local variable, a check is performed to determine whether the reference is swizzled or not. If the reference is not swizzled, the object name is translated into a pointer to the object's entry in the indirection table (this may require fetching the object if it is not cached), and the resulting value is stored both in the local variable and back in the instance variable.

In our system, the indirection table is also a swizzle table; it is used to translate object names into pointers to indirection table entries. The table is implemented as an overflow chaining hash table. Each entry in the indirection table contains a pointer to the fields of some object (or null if that object has been discarded), a reference count, and a next pointer used to link the entries that make up a hash table bucket.

3 Reference Count Management

Indirection table entries are freed using the lazy reference counting scheme. The scheme takes advantage of the copies maintained by our concurrency control algorithm in the undo log to update reference counts and free entries efficiently at commit time (other schemes maintain similar undo logs [12, 2]).

The lazy reference counting scheme is optimized for the common case — transactions whose base copies of modified objects fit in the undo log and whose accessed objects fit in a small fraction of the client cache [7]. We start by presenting the algorithm that handles the common case and later explain how it is extended to handle transactions that access and modify a large number of objects. We finish the section by describing key implementation details.

3.1 The Base Algorithm

Let e be an entry in the indirection table. The reference count field of e keeps an approximate count of the number of swizzled references to e. We call this count the *current reference count* of e or $CRC(e)$. The *actual reference count* of e or $ARC(e)$ is the exact number of swizzled references that refer to e in the stack, registers or instance variables of cached objects. We define $stack(e)$ as the number of swizzled references to e from the stack or registers. The function $refs(o, e)$ is defined as the number of swizzled references in the instance variables of object o that refer to e, and the function $refd(o)$ maps o to the multiset whose elements are the indirection table entries pointed to by each swizzled reference in o.

Eager reference counting schemes ensure that $CRC(e)$ is always equal to $ARC(e)$ for all e, but in our lazy reference counting scheme they are usually different. Instead, our system satisfies the following invariant:

Invariant 1: *For each entry e in the indirection table and all objects y modified during the current transaction, the actual number of swizzled references to e is given by:*

$$ARC(e) = CRC(e) + stack(e) + \sum(refs(y,e) - refs(y',e))$$

where y' is the copy of y in the undo log.

The invariant captures two important features of our algorithm. First, $CRC(e)$ never accounts for references in the stack or registers. Second, $CRC(e)$ does not account for references in the new versions of objects modified during a transaction until it commits, but accounts for the swizzled references in the objects' copies in the undo log. The first feature is similar to what *deferred reference counting* [6] provides but the second is new.

We now prove that the system preserves Invariant 1 by induction on the length of the execution, and simultaneously explain the various steps in our algorithm. The invariant holds vacuously for the base case because initially there are no objects in the cache. For the induction step, there are six operations that can change the reference counts in the system; we show that the invariant is preserved in all these cases:

(a) Swizzling: When a reference to entry e from object o is swizzled, $ARC(e)$ increases by one. If o has not been modified, our algorithm increments $CRC(e)$ by one and the invariant is preserved. Otherwise, the algorithm leaves $CRC(e)$ unchanged, but $refs(o,e)$ increases by one and the invariant is maintained.

(b) Creation of copy in undo log: Our system creates a copy y' of an object y in the undo log, the first time y is modified during a transaction. The invariant is preserved because initially $refs(y,e) = refs(y',e)$ for all e.

(c) Reference modification: Suppose a reference field is updated in object y such that it now refers to entry f instead of containing a reference to object o [1]. No CRC is modified by our algorithm when this happens. Let us consider two cases:

- If the reference to o was swizzled and pointed to o's corresponding indirection table entry e, $ARC(f)$ increases by one and $ARC(e)$ decreases by one. The invariant is preserved because $refs(y,f)$ increases by one while $refs(y,e)$ decreases by one.
- If the reference to o was not swizzled, only $ARC(f)$ changes; it increases by one. The invariant is preserved because $refs(y,f)$ increases by one. (Note that even if o has an associated entry e, $refs(y,e)$ will not change in this case.)

(d) Eviction: When an object o is evicted, the ARC of entries in $refd(o)$ decreases. Our algorithm decrements the CRC for all these entries and therefore maintains the invariant. (In the current version of the system, we never evict objects modified by the current transaction [4].)

(e) Invalidation: Servers send invalidations to a client when an object o in that client's cache is modified by another client. When such an invalidation is received, o is removed from the cache. Thus, like in the eviction case, the ARC of entries in $refd(o)$ decreases. Two cases may occur:

- If the invalidated object was not modified by the current transaction, the object is simply evicted using procedure (d).
- If the invalidated object o was modified by the current transaction, our algorithm preserves Invariant 1 by decrementing the *CRC(e)* for each e in $refd(o')$ and discarding o' (o's copy).

After processing the invalidation, if any invalidated object was used by the current transaction, the transaction aborts triggering the abort processing described next.

(f) Transaction commit: We distinguish two cases:

- If the commit is successful, our lazy reference counting algorithm computes the value of $refs$ for each modified object y and its copy y'; increments $CRC(e)$ by $\sum(refs(y,e) - refs(y',e))$ for each entry e; and clears the transaction logs. Thus, the invariant is maintained.
- If the transaction aborts, the system reverts all modified objects to their copies and clears the transaction logs. Therefore, the value of $ARC(e)$ decreases by $refs(y,e) - refs(y',e)$ for every entry e and modified object y. The invariant is preserved because y is reverted to the value in its copy.

[1] The technique we use to swizzle references ensures that, when a new reference value is assigned to a reference field, the new value is swizzled.

In the current implementation of our system, there are no swizzled references on the stack or registers at commit time, i.e., $stack(e) = 0$ for all entries e. Therefore, just after the commit processing described in (f) is performed $ARC(e) = CRC(e)$ for every e. At this point, our algorithm frees entries e such that $CRC(e) = 0$, ensuring the following basic safety and liveness properties.

Theorem 1 (Safety): *Indirection table entries are freed only if there are no swizzled references referring to them.*

Theorem 2 (Liveness): *Indirection table entries with no swizzled references referring to them are freed when a transaction terminates (unless the entry corresponds to an object that is still cached).*

3.2 Algorithm Extensions

When objects are evicted, the ARC of some indirection table entries may drop to zero. For transactions that access a large amount of data and incur many client cache misses, the amount of storage wasted by unreferenced indirection table entries may be significant. This section explains how the algorithm is extended to allow freeing of indirection table entries during transactions and to handle transactions that modify more data than what fits in the undo log.

Let *eager* be a subset of the objects modified by the current transaction. The extended algorithm updates reference counts eagerly to account for modifications to references contained in objects in the eager set. The algorithm inserts into eager modified objects whose copies do not fit in the undo log. Additionally, when the amount of storage wasted by unreferenced entries is significant relative to the total client cache size (e.g., when the storage used up by entries whose CRC dropped to zero is 1% of the client cache size), the algorithm inserts all modified objects in eager in order to allow freeing of entries during the transactions. This involves incrementing the reference counts of entries pointed to by modified objects (that are still not in eager) and it is performed while waiting for the reply to a fetch request.

The extended algorithm verifies a modified invariant (this can be shown using a proof similar to the one presented in Section 3.1):

Invariant 2: *For each entry e in the indirection table, all objects $y \notin$ eager modified during the current transaction, and all copies x' of objects $x \in$ eager, the actual number of swizzled references to e is given by:*

$$ARC(e) = CRC(e) + stack(e) - \sum refs(x', e) + \sum (refs(y, e) - refs(y', e))$$

where y' is y's copy in the undo log.

The term $\sum \text{refs}(x', e)$ reflects the fact that $CRC(e)$ accounts for both the references that point to e in objects in eager and in their copies. This guarantees that entries e that are pointed to by copies have $CRC(e) > 0$ until the transaction commits (or aborts). Therefore, these entries are not freed during a transaction making it possible to revert the state of modified objects in eager to their copies if the transaction aborts.

The extended algorithm works as follows:

(a_1) Swizzling: When a reference to entry e from an object $y \in$ *eager* is swizzled, the algorithm increments $CRC(e)$ by one. For objects not in eager, it proceeds as described in (a).

(b_1) Creation of copy in undo log: If the undo log is full the first time an object y is modified during a transaction, the algorithm adds y to eager. Otherwise, the algorithm proceeds as in (b).

(c_1) Pointer modification: When a pointer contained in object $y \in$ *eager* is modified to point to entry f instead of containing a reference to object o, the algorithm increments $CRC(f)$ by one. If the reference to o was swizzled and pointed to o's corresponding indirection table entry e, $CRC(e)$ is decremented by one. For objects not in eager, the algorithm behaves as described in (c).

(d_1) Eviction: While waiting for the reply to a fetch request, the algorithm computes the value of *refs* for each modified object $y \notin$ *eager*; increments $CRC(e)$ by $\sum refs(y, e)$; and adds all y to eager. Then for each evicted object o, it decrements $CRC(e)$ for each e in $refd(o)$ and frees e if $CRC(e) = 0 \wedge stack(e) = 0$.

(e_1) Invalidation: If the client receives an invalidation for an object $y \in$ *eager*, the algorithm decrements *$CRC(e)$* for each $e \in refd(y)$. If y has a copy y', the algorithm decrements *$CRC(e)$* for each $e \in refd(y')$. The algorithm also performs the pro-

cessing described in (e).

(f_1) Transaction commit: We distinguish two cases:

- *If the commit is successful:* The algorithm decrements *CRC(e)* for each $e \in refd(y')$ (where y' is a copy of some object in eager. No processing is performed for objects in eager without a copy. For objects not in eager, the algorithm proceeds as described in (f).
- *If the transaction aborts:* For each $y \in eager$, the algorithm decrements *CRC(e)* for all e in $refd(y)$. If y has a copy y', the algorithm also decrements *CRC(e)* for each $e \in refd(y')$ and reverts y to its copy. No processing is required for objects not in eager.

Once more, just after the commit processing described in (f_1) is performed $ARC(e) = CRC(e)$ for every e and the algorithm frees entries with $CRC(e) = 0$. Using invariant 2 it is possible to show that the extended algorithm also verifies the safety and liveness theorems.

3.3 Implementation and Optimizations

This section describes the key overheads in the implementation of lazy reference counting and presents the techniques used to reduce them.

3.3.1 Space Overheads

The reference count field in the indirection table is only 1 byte, which should be enough to store the *CRC* of most entries. If an overflow occurs, the field takes the value 255 and the actual reference count is stored in a hash table keyed by the index of the entry in the indirection table. The algorithm also uses 2 bits in the object header: the *pointer-update* bit which is set if and only if a pointer in the object was modified during the current transaction; and the *eager* bit which is set if and only if the object is in the *eager* set. The pointer-update bit is set by code that is automatically generated by our Theta [9] compiler.

3.3.2 Time Overheads

We now discuss the time overhead introduced by each step in the lazy reference counting algorithm and the optimizations used to reduce it. We focus on the common case where all data modified by a transaction fits in the undo log and no entries are freed during a transaction (i.e., the eager set is empty).

Adjusting reference counts when a reference is swizzled can potentially add an extra cache miss. To avoid this overhead, our swizzle table is also the indirection table. Therefore, when a reference is swizzled its corresponding entry in the indirection table is accessed and the reference count adjustment is inexpensive.

When an object is evicted, its references are scanned and for each swizzled reference, the reference count of the target entry is decremented and if it drops to zero the entry's index is appended to the *zero* log (if the corresponding object was discarded from the cache). Similarly, if the entry being discarded also has a null reference count it is added to the *zero* log. This process is memory intensive and relatively expensive. To reduce the overhead of adjusting reference counts when objects are evicted, our system performs this processing while waiting for the reply to a fetch request. Unless the client is multi-threaded or multi-programmed, this allows us to completely overlap eviction time adjustment of reference counts with the fetch delay. Therefore, eviction time processing usually introduces no overhead.

Commit time processing is divided in two steps. The first step adjusts reference counts to reflect the current state of modified objects. It compares modified objects with their copies to determine which references were modified, and adjusts reference counts accordingly. As an optimization, the comparison is only performed if the pointer-update bit is set because otherwise it is known that no adjustments are required.

In a system that performs pointer swizzling, the client must unswizzle references in modified objects before it sends their new states to the server in a commit request. Our implementation takes advantage of this by dividing the first step into an *increment phase* and a *decrement phase*. The increments are all performed when references are unswizzled and introduce a negligible overhead because the unswizzling code accesses the entries whose counts need to be incremented. The decrement phase is performed in the background while waiting for the reply to the commit request and introduces no additional delay (unless the client is multi-threaded or multi-programmed in which case it may introduce some delay). Entries whose reference counts drop to zero during the decrement phase are marked as free. However, since these entries are pointed to by the copies of modified objects, the changes performed while freeing them are logged such that they can be undone in case the transaction aborts.

The second step, iterates over the zero log and marks as free any entry whose reference count is still zero and whose corresponding object was discarded from the cache. The second step is also performed in the background while waiting for the reply to the commit request. No logging is required in this case because the objects that referred to these entries were discarded from the cache.

If marking an entry as free required removing the entry from its hash bucket chain in the swizzle table and inserting it in some free list, commit time processing could be excessive. We avoid this overhead by simply setting a bit in the entry. The free entry remains linked to its hash bucket chain and will be reused when a new object (whose name hashes to the same bucket) is installed in the indirection table. This technique works well in practice because our multiplicative hashing function spreads the entries across buckets very uniformly.

4 Performance Evaluation

This section shows that the overheads introduced by our lazy reference counting scheme are low.

4.1 Experimental Setup

The experiments ran a single client on a DEC 3000/400 workstation, and a server on another, identical workstation. Both workstations had 128 MB of memory and ran OSF/1 version 3.2. They were connected by a 10 Mb/s Ethernet and had DEC LANCE Ethernet interfaces. The database was stored on a Seagate ST-32171N disk, with a peak transfer rate of 15.2 MB/s, an average read seek time of 9.4 ms, and an average rotational latency of 4.17 ms. The server cache was 36 MB and the client cache was 12 MB. We used 8 KB pages.

The experiments ran the single-user OO7 benchmark [3]. The OO7 database contains a tree of *assembly* objects, with leaves pointing to three *composite parts* chosen randomly from among 500 such objects. Each composite part contains a graph of *atomic parts* linked by *connection* objects; each atomic part has 3 outgoing connections. We used the medium database which has 200 atomic parts per composite part and a total size of 37.8 MB. We implemented the database in Theta [9], following the specification [3] closely.

We ran traversals T1 and T2b. These traversals perform a depth-first traversal of the assembly tree and execute an operation on the composite parts referenced by the leaves of this tree. T1 is read-only; it reads the entire graph of a composite part. T2b is identical to T1 except that T2b modifies all the atomic parts. All pointer updates in T2b are due to pointer swizzling; therefore, we added a new traversal called *R2b* that has pointer updates due to both pointer swizzling and assignments to pointer variables. Traversal R2b is similar to T2b except that it swaps the second and third connections in the set of incoming connections of an atomic part.

The code was compiled with GNU's gcc with optimization flag -O2. Each traversal was run in a single transaction as specified in [3]. The traversals were run on an isolated network.

4.2 Experiments

Figure 2 shows the time spent managing reference counts and freeing indirection table entries for T1, T2b and R2b. The bars labeled *Configuration 2* correspond to a configuration with a 512 KB undo log and the lazy reference counting algorithm configured to start freeing indirection table entries during a transaction when the storage used up by entries with null reference counts reaches 1% of the client cache size. This is our default configuration which was used to obtain the results presented in [4].

T1, T2b and R2b run in a single transaction, access significantly more data than what fits in the client cache, and T2b and R2b modify more data than what fits in the undo log (T2b modifies approximately 4 MB of data and R2b modifies approximately 3.5 MB of data). Therefore, the bars labeled *Configuration 2* correspond to transactions that do not fall in the common case for which we designed our lazy reference counting algorithm. We believe these transactions are uncommonly large [7]. To simulate the behavior of the system with smaller transactions, we also ran T1, T2b and R2b with an undo log large enough to hold copies of all modified objects and without freeing entries in the middle of transactions. The bars labeled *Configuration 1* correspond to the latter configuration.

The tops of the bars in Figure 2 were erased to represent the portion of the time spent managing reference counts that is overlapped with fetch and commit delays.

The experimental methodology we followed to obtain the overheads presented in Figure 2 consisted in comparing the execution times of our system and a version of our system from which all reference counting code was removed. Therefore, the numbers we present include the overheads due to increased instruction cache and TLB misses. In fact, the foreground overhead of reference counting in traversals T1 and T2b is significantly higher than we expected and we believe this is due to conflicts in our processor's direct-mapped code cache.

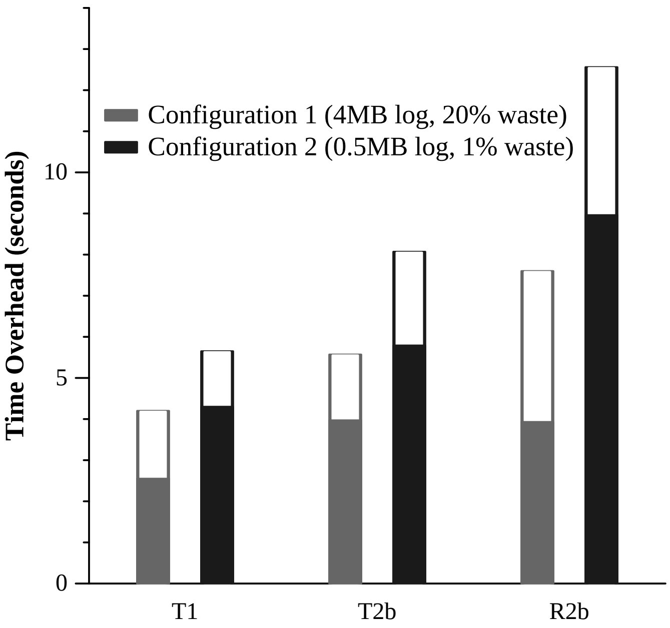

Figure 2: Reference Count Management Overhead (the white bars represent overhead that was overlapped with fetch and commit delays)

The results in Figure 2 show that the system performs better in *Configuration 1* which we expect will be the behavior in the common case. This happens because reference count updates and freeing entries can be delayed until the end of a transaction which reduces the total amount of work to be performed. The results also show that performing reference count management operations in the background reduces the overhead.

Table 1 shows the reference counting overhead relative to the total traversal time. This table shows *best case* and *worst case* overhead for each traversal. The best case corresponds to *Configuration 1* with overlapping and the worst case corresponds to *Configuration 2* without overlapping. The important fact to retain is that the overhead is low; it is lower than 8% in the worst case, lower than 4% in the best case, and we expect most traversals to fall in the best case.

	T1	T2b	R2b
Worst Case	6%	8%	8%
Best Case	3%	4%	3%

Table 1: Reference Counting Overhead Relative to Total Traversal Time

5 Conclusion

In a companion paper [4], we introduced a novel cache management technique, HAC, and showed that it outperforms other techniques across a wide range of cache sizes and workloads. This report's main contribution is a detailed description of lazy reference counting; a technique that solves a specific problem in HAC: how to discard indirection table entries in an *indirect pointer swizzling* scheme. This report also shows that lazy reference counting has a low cost because it reduces the amount of reference count management work and performs a significant portion of this work in the background while waiting for the replies to fetch and commit requests.

References

[1] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. In *Proceedings of the SIGMOD International Conference on Management of Data*, pages 23–34. ACM Press, May 1995.

[2] M. Carey et al. The EXODUS extensible DBMS project: An overview. In *Readings in Object-Oriented Database Systems*, pages 474–499. Morgan Kaufmann, 1990.

[3] M. J. Carey, D. J. DeWitt, and J. F. Naughton. The OO7 benchmark. Technical Report; Revised Version dated 7/21/1994 1140, University of Wisconsin-Madison, 1994. At `ftp://ftp.cs.wisc.edu/OO7`.

[4] Miguel Castro, Atul Adya, Barbara Liskov, and Andrew C. Myers. HAC: Hybrid Adaptive Caching for Distributed Storage Systems. In *16th ACM Symp. on Operating System Principles*, Saint-Malo, France, October 1997.

[5] G. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 2(12), December 1960.

[6] L. Deutsch and D Bobrow. An efficient, incremental, automatic garbage collector. *Communications of the ACM*, 19(9), October 1976.

[7] R. Gruber. *Optimism vs. Locking: A Study of Concurrency Control for Client-Server Object-Oriented Databases*. PhD thesis, M.I.T., Cambridge, MA, 1997.

[8] Barbara Liskov, Atul Adya, Miguel Castro, Mark Day, Sanjay Ghemawat, Robert Gruber, Umesh Maheshwari, Andrew C. Myers, and Liuba Shrira. Safe and efficient sharing of persistent objects in Thor. In *Proc. SIGMOD Int'l Conf. on Management of Data*, pages 318–329, June 1996.

[9] Barbara Liskov, Dorothy Curtis, Mark Day, Sanjay Ghemawat, Robert Gruber, Paul Johnson, and Andrew C. Myers. *Theta Reference Manual*. Programming Methodology Group Memo 88, MIT Laboratory for Computer Science, Cambridge, MA, February 1994. Available at http://www.pmg.lcs.mit.edu/papers/thetaref/.

[10] J. Eliot B. Moss. Working with persistent objects: To swizzle or not to swizzle. *IEEE Transactions on Software Engineering*, 18(3), August 1992.

[11] Kaehler T. and Krasner G. *LOOM—Large Object-Oriented Memory for Smalltalk-80 Systems*, pages 298–307. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1990.

[12] S. J. White and D. J. Dewitt. Quickstore: A high performance mapped object store. In *SIGMOD '94*, pages 187–198, 1994.

[13] Seth J. White and David J. Dewitt. A performance study of alternative object faulting and pointer swizzling strategies. In *Proceedings of the Eighteenth International Conference on Very Large Data Bases*, pages 419–431, Vancouver, BC, Canada, 1992.

[14] P. Wilson. Uniprocessor garbage collection techniques. *ACM Computing Surveys*, 1996.