Appears in the *Proceedings of the ACM Symposium on Operating System Principles (SOSP '97), Saint-Malo, France, October 1997*

HAC: Hybrid Adaptive Caching for Distributed Storage Systems

Miguel Castro **Atul Adya** **Barbara Liskov** **Andrew C. Myers**

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

Abstract

This paper presents HAC, a novel technique for managing the client cache in a distributed, persistent object storage system. HAC is a hybrid between page and object caching that combines the virtues of both while avoiding their disadvantages. It achieves the low miss penalties of a page-caching system, but is able to perform well even when locality is poor, since it can discard pages while retaining their hot objects. It realizes the potentially lower miss rates of object-caching systems, yet avoids their problems of fragmentation and high overheads. Furthermore, HAC is adaptive: when locality is good it behaves like a page-caching system, while if locality is poor it behaves like an object-caching system. It is able to adjust the amount of cache space devoted to pages dynamically so that space in the cache can be used in the way that best matches the needs of the application.

The paper also presents results of experiments that indicate that HAC outperforms other object storage systems across a wide range of cache sizes and workloads; it performs substantially better on the expected workloads, which have low to moderate locality. Thus we show that our hybrid, adaptive approach is the cache management technique of choice for distributed, persistent object systems.

1 Introduction

In distributed persistent storage systems, servers provide persistent storage for information accessed by applications running at clients [LAC$^+$96, C$^+$89, WD94, LLOW91, BOS91]. These systems cache recently used information at client machines to provide low access latency and good scalability.

This paper presents a new hybrid adaptive caching technique, HAC, which combines page and object caching to reduce the miss rate in client caches dramatically. The approach provides improved performance over earlier approaches — by more than an order of magnitude on common workloads. HAC was developed for use in a persistent object store but could be applied more generally, if provided information about object boundaries. For example, it could be used in managing a cache of file system data, if an application provided information about locations in a file that correspond to object boundaries. HAC could also be used within a server cache.

In persistent object systems, objects are clustered in fixed-size pages on disk, and pages are much larger than most objects [Sta84]. Most systems manage the client cache using *page caching* [LLOW91, WD94, SKW92]: when a miss occurs, the client evicts a page from its cache and fetches the missing object's page from the server. Page caching achieves low miss penalties because it is simple to fetch and replace fixed-size units; it can also achieve low miss rates provided clustering of objects into pages is good. However, it is not possible to have good clustering for all application access patterns [TN92, CK89, Day95]. Furthermore, access patterns may evolve over time, and reclustering will lag behind because effective clustering algorithms are very expensive [TN92] and are performed infrequently. Therefore, pages contain both *hot* objects, which are likely to be used by an application in the near future, and *cold* objects, which are not likely to be used soon. Bad clustering, i.e., a low fraction of hot objects per page, causes page caching to waste client cache space on cold objects that happen to reside in the same pages as hot objects.

Object-caching systems [Ont92, D$^+$90, LAC$^+$96, C$^+$94b, KK90, WD92, KGBW90] allow clients to cache hot objects without caching their containing disk pages. Thus, object caching can achieve lower miss rates than page caching when clustering is bad. However, object caching has two problems: objects are variable-sized units, which leads to storage fragmentation, and there are many more objects than pages, which leads to high overhead for bookkeeping and for maintaining per-object usage statistics.

Dual buffering [KK94] partitions the client cache stati-

This research was supported in part by DARPA contract DABT63-95-C-005, monitored by Army Fort Huachuca, and in part by DARPA contract N00014-91-J-4136, monitored by the Office of Naval Research. M. Castro is supported by a PRAXIS XXI fellowship.

cally into a page cache and an object cache. This technique has been shown to achieve lower miss rates than both page caching and object caching when the fraction of space dedicated to the object cache is manually tuned to match the characteristics of each application. Such tuning is not feasible for real applications; furthermore, these systems do not solve the problems of storage fragmentation and high overheads in object caching systems.

HAC is the first technique to provide an adequate solution to all these problems. It is a hybrid between page and object caching that combines the virtues of each — low overheads and low miss rates — while avoiding their problems. Furthermore, HAC adaptively partitions the cache between objects and pages based on the current application behavior: pages in which locality is high remain intact, while for pages in which locality is poor, only hot objects are retained. To our knowledge, HAC is the first adaptive caching system. O'Toole and Shrira [OS95] present a simulation study of an adaptive caching system, but they focus on avoiding reads to install modifications at servers and ignore storage management issues.

HAC partitions the client cache into page frames and fetches entire pages from the server. To make room for an incoming page, HAC does the following:

- selects some page frames for *compaction*,
- discards the cold objects in these frames,
- compacts the hot objects to free one of the frames.

The approach is illustrated in Figure 1. The policy to select page frames for compaction strives both to achieve low overhead and to discard objects that are unlikely to be reused. HAC maintains usage statistics on a per-object basis using a novel technique with low space and time overheads. The technique combines both recency and frequency information to make good replacement decisions.

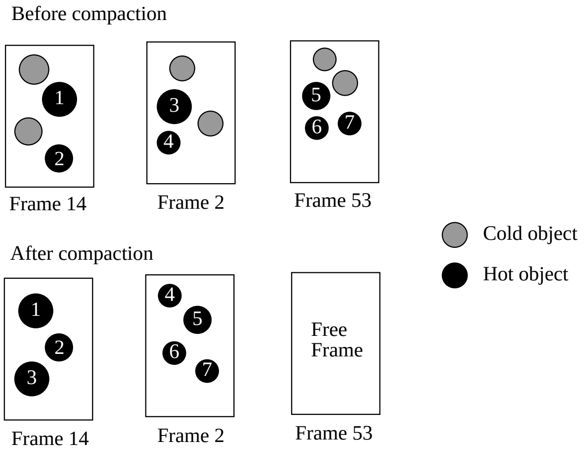


Figure 1: Compaction of pages by HAC

HAC has been incorporated in Thor-1, a new implementation of the Thor object database [LAC$^+$96]; it retains the server storage management and concurrency control techniques of our previous implementation, Thor-0, but uses HAC for client cache management. The high performance of Thor-1 is furthermore achieved by adherence to two design principles: *think small*, and *be lazy*. We were careful about the sizes of all data structures, but particularly about objects; we designed our object format parsimoniously, and the result is better performance in all parts of the system, from the server disk to the client cache. Furthermore, we do work only when absolutely necessary, since this allows us to avoid it entirely in many cases. We will explain how these principles were applied in our implementation and how they contribute to our performance.

The paper presents results of performance experiments that show the benefits of these techniques. The experiments evaluate their impact on performance across a wide range of cache sizes and workloads. Our results show that HAC consistently outperforms page-based approaches and achieves speedups of more than an order of magnitude on memory-bound workloads with typically achievable clustering. We also show that Thor-1 outperforms the best object storage systems.

The rest of the paper is organized as follows. Section 2 gives an overview of our implementation, and describes the object format at both clients and servers. Section 3 describes HAC. We present performance results in Section 4. Section 5 summarizes the main results in the paper.

2 System Overview

In Thor, applications running at client machines make use of persistent objects stored at servers. Applications interact with Thor by calling methods of objects; these calls occur within atomic transactions. The methods are implemented in a type-safe language called Theta [LCD$^+$94].

To speed up applications, client machines cache copies of persistent objects. Clients communicate with servers only to fetch pages or commit transactions; we use optimistic concurrency control [AGLM95, Gru97] to serialize transactions. The fact that computations run as transactions does not preclude the use of HAC in non-transactional systems, since the main impact of transactions is defining when modifications are sent to servers, and this could be defined in a different way, e.g., for file systems, updates could be sent every few seconds.

Good performance for a distributed object storage system requires good solutions for client cache management, storage management at servers, and concurrency control for transactions. Performance studies of our previous implementation, Thor-0, showed its techniques for server storage management [Ghe95] and concurrency control [AGLM95, Gru97] worked well and therefore these were retained in Thor-1.

2.1 Servers

Servers store objects on disk in pages and maintain a page cache in main memory to speedup client fetch requests. The results presented in this paper were obtained using 8 KB pages, but the system can be configured to use a different page size. To simplify cache management, objects are required not to span page boundaries. Pages are large enough that this does not cause significant internal fragmentation. Objects larger than a page are represented using a tree.

When a transaction commits, its client sends information about its modifications to the servers in the commit request; if the commit succeeds, these changes become visible to later transactions. Because client caches in HAC may hold objects without their containing pages, we must ship modified objects (and not their containing pages) to servers at commit time. Earlier work [OS94, WD95] has shown that this leads to poor performance if it is necessary to immediately read the objects' pages from disk to install them in the database. We avoid this cost by using the *modified object buffer* (MOB) architecture [Ghe95]. The server maintains an in-memory MOB that holds the latest versions of objects modified by recent transactions. New versions are written to the MOB when transactions commit; when the MOB fills up, versions are written to their disk pages in the background.

2.2 Page and Object Format

Keeping objects small at both servers and clients is important because it has a large impact on performance [WD94, MBMS95]. Our objects are small primarily because object references (or *orefs*) are only 32 bits. Orefs refer to objects at the same server; objects point to objects at other servers indirectly via *surrogates*. A surrogate is a small object that contains the identifier of the target object's server and its oref within that server; this is similar to designs proposed in [Bis77, Mos90, DLMM94]. Surrogates will not impose much penalty in either space or time, assuming the database can be partitioned among servers so that inter-server references are rare and are followed rarely; we believe these are realistic assumptions.

Object headers are also 32 bits. They contain the oref of the object's class object, which contains information such as the number and types of the object's instance variables and the code of its methods.

An oref is a pair consisting of a 22-bit *pid* and a 9-bit *oid* (the remaining bit is used at the client as discussed below). The *pid* identifies the object's page and allows fast location of the page both on disk and in the server cache. The *oid* identifies the object within its page but does not encode its location. Instead, a page contains an *offset table* that maps *oids* to 16-bit offsets within the page. The offset table has an entry for each existing object in a page; this 2-byte extra overhead, added to the 4 bytes of the object header, yields a total overhead of 6 bytes per object. The offset table is important because it allows servers to compact objects within their pages independently from other servers and clients. It also provides a larger address space, allowing servers to store a maximum of 2 G objects consuming a maximum of 32 GB; this size limitation does not unduly restrict servers, since a physical server machine can implement several logical servers.

Our design allows us to address a very large database. For example, a server identifier of 32 bits allows 2^{32} servers and a total database of 2^{67} bytes. However, our server identifiers can be larger than 32 bits; the only impact on the system is that surrogates will be bigger. In contrast, most systems that support large address spaces use very large pointers, e.g., 64 [CLFL95, LAC^{+}96], 96 [Kos95], or even 128-bit pointers [WD92]. In Quickstore [WD94], which also uses 32-bit pointers to address large databases, storage compaction at servers is very expensive because all references to an object must be corrected when it is moved (whereas our design makes it easy to avoid fragmentation).

2.3 Clients

The client cache is partitioned into a set of page-sized *frames*. Some frames are *intact*: they hold pages that were fetched from servers. Others are *compacted*; they hold objects that were retained when their containing server pages were compacted. When a client tries to use an object that is not in its cache, it fetches the object's containing page from the appropriate server and frees a frame. Pages at the client have the same size and structure as at the server to avoid extra copies.

It is not practical to represent object pointers as *orefs* in the client cache because each pointer dereference would require a table lookup to translate the name into the object's memory location. Therefore, clients perform *pointer swizzling* [KK90, Mos92, WD92], i.e., replace the *orefs* in objects' instance variables by virtual memory pointers to speed up pointer traversals. HAC uses *indirect* pointer swizzling [KK90], i.e., the *oref* is translated to a pointer to an entry in an *indirection table* and the entry points to the target object. Indirection allows HAC to move and evict objects from the client cache with low overhead; indirection has also been found to simplify page eviction in a page-caching system [MS95].

HAC uses a novel lazy reference counting mechanism to discard entries from the indirection table. The reference count in an entry is incremented whenever a pointer is swizzled and decremented when objects are evicted, but no reference count updates are performed when objects are modified. Instead, reference counts are corrected lazily when a transaction commits, to account for the modifications performed during the transaction. This scheme is described in detail in [CAL97], where it is shown to have low overheads.

In-cache pointers are 32 bits, which is the pointer size on most machines. HAC uses 32-bit pointers even on 64-bit machines; it simply ensures that the cache and the indirection table are located in the lower 2^{32} bytes of the address space. The results presented in Section 4 were obtained on a machine with 64-bit pointers.

Both pointer swizzling and *installation* of objects, i.e., allocating an entry for the object in the indirection table, are performed *lazily*. Pointers are swizzled the first time they are loaded from an instance variable into a register [Mos92, WD92]; the extra bit in the *oref* is used to determine whether a pointer has been swizzled or not. Objects are installed in the indirection table the first time a pointer to them is swizzled. The size of an indirection table entry is 16 bytes. Laziness is important because many objects fetched to the cache are never used, and many pointers are never followed. Furthermore, lazy installation reduces the number of entries in the indirection table, and it is cheaper to evict objects that are not installed.

3 Hybrid Adaptive Caching

HAC improves system performance by reducing the miss rate in client caches: it retains useful objects without needing to cache their pages, and it can cache more objects than existing object-caching systems because the compaction mechanism greatly reduces storage fragmentation. Additionally, HAC avoids the high overheads associated with existing object-caching systems.

We begin by describing how compaction works. Then we describe how we select frames to compact and objects to discard.

3.1 Compaction

HAC computes usage information for both objects and frames as described in Section 3.2, and uses this information to select a victim frame F to compact, and also to identify which of $F's$ objects to retain and which to discard.

Then it moves retained objects from F into frame T, the current *target* for retained objects, laying them out contiguously to avoid fragmentation (see Figure 2). Indirection table entries for retained objects are corrected to point to their new locations; entries for discarded objects are modified to indicate that the objects are no longer present in the cache; also, the reference counts of the objects referenced by discarded objects are decremented. If all retained objects fit in T, the compaction process ends and F can be used to receive the next fetched page; this case is shown in Figure 2(a). If some retained objects do not fit in T, F becomes the new target and the remaining objects are compacted inside F to make all the available free space contiguous. Then, another frame is selected for compaction and the process is repeated for that frame. This case is shown in Figure 2(b).

This compaction process preserves locality: retained objects from the same disk page tend to be located close together in the cache. Preserving locality is important because it takes advantage of any spatial locality that the on-disk clustering algorithm may be able to capture.

An important issue in hybrid caching is handling the situation where the disk page P of an object o is fetched and o is already in use, cached in frame F. HAC handles this situation in a simple and efficient way. No processing is performed when P is fetched. Since the copy of o in F is installed in the indirection table, o's copy in P will not be installed or used. If there are many such unused objects in P, its frame will be a likely candidate for compaction, in which case all its uninstalled copies will simply be discarded. If instead F is freed, its copy of o is moved to P (if o is retained) instead of being compacted as usual. In either case, we avoid both extra work and foreground overhead. In contrast, the eager approach in [KK94] copies o into P when P is fetched; this increases the miss penalty because the object is copied in the foreground, and wastes effort if P is compacted soon after but o is not evicted.

3.2 Replacement Policy

It is more difficult to achieve low space and time overheads in object-caching systems than in page-caching systems because there are many more objects than pages. Despite this difficulty, our replacement policy achieves low miss rates with low space and time overheads. We keep track of object usage by storing 4 bits per object that indicate both how recently and how frequently the object has been used. This object usage information is used to compute frame usage information, but frame usage is only computed occasionally because its computation is expensive. Frames are selected for compaction based on their last computed usage; objects within the frame are retained or discarded based on how their usage compares to that of their frame.

3.2.1 Object Usage Computation

It has been shown that cache replacement algorithms which take into account both recency and frequency of accesses – 2Q [JS94], LRU-K [OOW93] and frequency-based replacement [RD90] – can outperform LRU because they can evict recently used pages that are used infrequently. However, the algorithms that have been proposed so far have high space and time overheads.

HAC maintains per-object usage statistics that capture information on both recency and frequency of accesses while incurring very low space overheads. Headers of installed objects contain 4 usage bits. The most significant usage bit is set each time a method is invoked on the object. Besides its

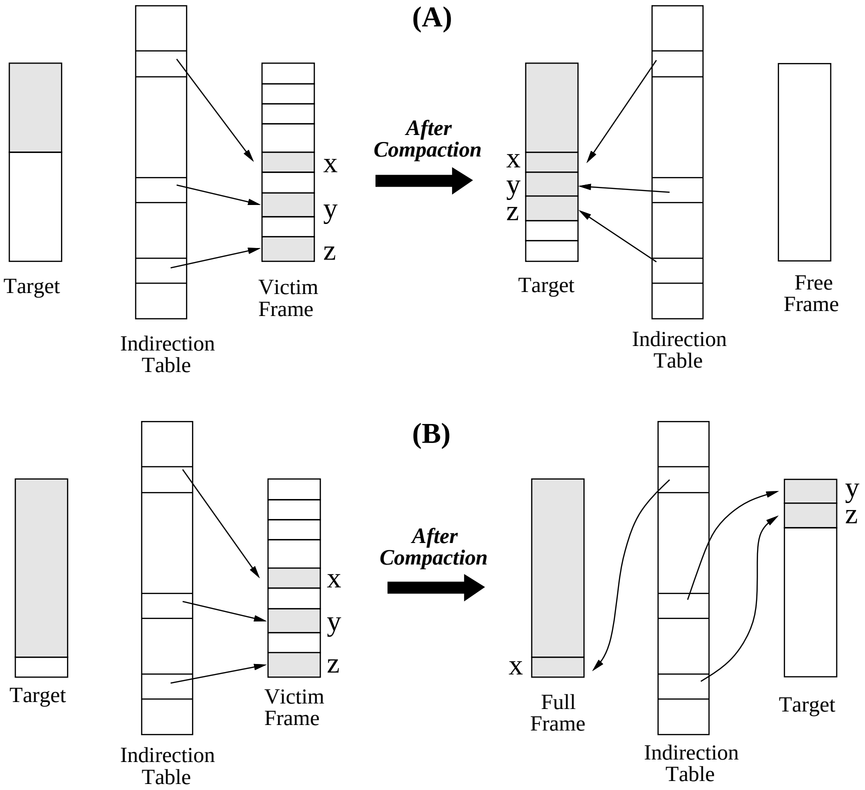


Figure 2: Compacting objects in a frame

low space overhead, this scheme has low impact on hit time; it adds at most two extra instructions and no extra processor cache misses to a method call.

The usage value is decayed periodically by shifting right by one; thus, each usage bit corresponds to a decay period and it is set if the object was accessed in that period. Our scheme considers objects with higher usage (interpreting the usage as a 4-bit integer) as more *valuable*, i.e., objects that were accessed in more recent periods are more valuable and when the last accesses to two objects occurred in the same period, their value is ordered using the history of accesses in previous periods. Therefore, our scheme acts like LRU but with a bias towards protecting objects that were frequently accessed in the recent past. To further increase this bias and to distinguish objects that have been used in the past from objects that have never been used, we add one to the usage bits before shifting; we found experimentally that this increment reduces miss rates by up to 20% in some workloads. Our decay rule is identical to the one used to decay reference counts in [RD90], but its purpose is quite different.

Usage bits are much cheaper in both space and time than maintaining either an LRU chain or the data structures used by 2Q, LRU-K and frequency based replacement. For example, the system described in [Kos95] uses a doubly linked list to implement LRU. This imposes an overhead of at least an extra 8 bytes per object for the LRU information, which increases the client cache miss rate and also increases hit times substantially by adding up to three extra processor cache misses per object access.

HAC uses fine-grained concurrency control [AGLM95, Gru97] and some objects in a cached page may be invalidated (when they become stale) while the rest of the page remains valid. We set the usage of invalid objects to 0, which ensures their timely removal from the cache. In contrast, page-caching systems that use fine-grained concurrency control (e.g., adaptive callback locking [CFZ94], which does concurrency control by adaptively locking either objects or pages) waste cache space holding invalid objects because they always cache full pages.

3.2.2 Frame Usage Computation

We could implement replacement by evicting the objects with the lowest usage in the cache, but this approach may pick objects from a large number of frames. Since HAC compacts objects retained from these frames, compaction time may be unacceptable, e.g., compacting 126 frames could take up to 1 second in our experimental environment. Therefore, we compute usage values for frames and use these values to select frames to compact and indirectly objects to evict.

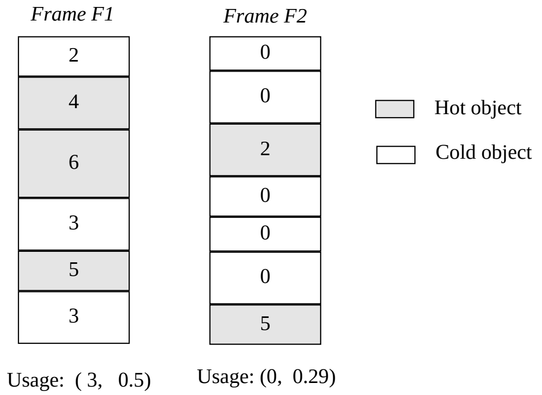


Figure 3: Usage statistics for frames

Our goals in freeing a frame are to retain hot objects and to free space. The frame usage value reflects these goals. It is a pair $<T, H>$. T is the *threshold*: when the frame is discarded, only *hot* objects, whose usage is greater than T, will be retained. H is the fraction of objects in the frame that are hot at threshold T. We require H to be less than the *retention fraction*, R, where R is a parameter of our system. T is the minimum usage value that results in an H that meets this constraint. Frame usage is illustrated (for R = 2/3) in Figure 3. For frame F1, T = 2 would not be sufficient since this would lead to H = 5/6; therefore we have T = 3. For frame F2, T = 0 provides a small enough value for H.

We use object count as an estimate for the amount of space occupied by the objects because it is expensive to compute this quantity accurately in the HAC prototype; it is a reasonable estimate if the average object size is much smaller than a page.

Selecting a value for system parameter R involves a tradeoff between retaining hot objects to reduce miss rate and achieving low replacement overhead by reducing the number of frames whose contents need to be compacted. If R is large, only very cold objects will be discarded but little space may be recovered; if R is small, more space is freed but the miss rate may increase since hot objects may be evicted. We have found experimentally that $R = 2/3$ works well; Section 4.1 describes the sensitivity experiments we used to determine the value of R and other system parameters.

HAC uses a *no-steal* [GR93] cache management policy, i.e., objects that were modified by a transaction cannot be evicted from the cache until the transaction commits. The frame usage is adjusted accordingly, to take into account the fact that modified objects are retained regardless of their usage value: when computing the usage of a frame, we use the *maximum usage value* for modified objects rather than their actual usage value.

3.2.3 The Candidate Set

The goal for replacement is to free the least valuable frame. Frame F is less valuable than frame G if its usage is lower:

$$\text{F.T} < \text{G.T or (F.T = G.T and F.H} < \text{G.H)}$$

i.e., either F's hot objects are likely to be less useful than G's, or the hot objects are equally useful but more objects will be evicted from F than from G. For example, in Figure 3, F2 has lower usage than F1.

Although in theory one could determine the least valuable frame by examining all frames, such an approach would be much too expensive. Therefore, HAC selects the victim from among a *set of candidates*. A few frames are added to this set during each *epoch*, i.e., at each fetch. A frame's usage is computed when it is added to the set; since this computation is expensive, we retain frames in the candidate set, thus increasing the number of candidates for replacement at later fetches without increasing replacement overhead. Since frame usage information that was calculated long ago is likely to be out of date, we remove entries that have remained in the candidate set for E epochs. We use a value of 20 for E (see Section 4.1).

We add new members to the candidate set using a variant of the CLOCK algorithm [Cor69]. We organize the cache as a circular array of frames and maintain a *primary scan pointer* and N *secondary scan pointers* into this array; these pointers are equidistant from each other in the array. When a frame needs to be freed, HAC computes the usage value for S contiguous frames starting at the *primary pointer* and adds those frames to the candidate set; then it increments the *primary scan pointer* by S. We use a value of $S = 3$ (see Section 4.1).

The secondary pointers are used to ensure the timely eviction of uninstalled objects, i.e., objects that have not been used since their page was fetched into the cache. These objects are good candidates for eviction provided they have not been fetched too recently into the cache [OOW93, RD90]. The secondary pointers are used to find frames with a large number of uninstalled objects. For each secondary pointer, HAC determines the number of installed objects in S contiguous frames starting at that scan pointer and then advances the pointer by S. It enters a frame in the candidate set if the fraction of installed objects in the frame is less than R; the thresholds of these frames will always be zero since uninstalled objects have a usage value of zero. In each epoch, at most $S \times N$ members are added to the candidate set from the secondary scan pointers. The secondary scan pointers introduce low overhead because HAC keeps track of the number of installed objects in each frame and no scanning through the usage values of objects is needed.

Since scan pointers are equidistant from one another, a newly fetched page will be protected until approximately another $\frac{1}{(N+1)S}$ of the frames in the cache has been freed. The

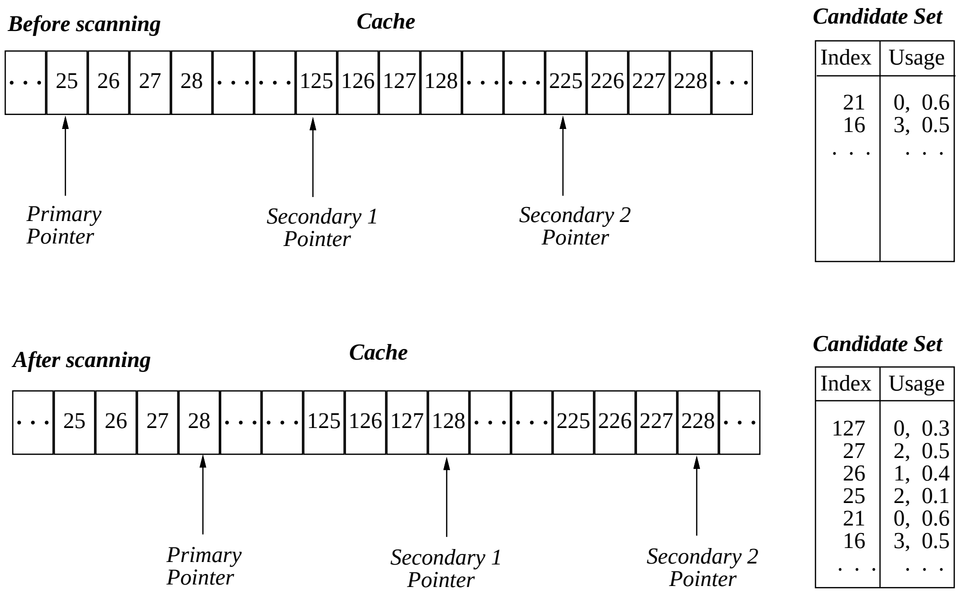


Figure 4: Adding new members to the candidate set

choice of N is thus important: a low number of secondary scan pointers will lead to wasting cache space with uninstalled objects but a larger number can increase miss rates by prematurely evicting recently fetched objects. We found experimentally that a value of 2 for N works well (see Section 4.1); it protects recently fetched objects from eviction until approximately $1/9$ of the frames in the cache have been freed.

Figure 4 shows an example for a 300-page cache. The three scan pointers are spaced 100 frames apart. After scanning, four frames (25, 26, 27, and 127) have been added to the set; the other frames at the secondary pointers were not added since they did not have enough uninstalled objects.

HAC decays object usage when it computes frame usage at the primary scan pointer: it needs to scan the objects at this point, and therefore decaying object usage adds almost no overhead. However, if there are no fetches, usage values will not be decayed and it will be difficult to predict which objects are less likely to be accessed. If this were a problem, we could perform additional decays of object usage when the fetch rate is very low. These decays would not impose a significant overhead if performed infrequently, e.g., every 10 seconds.

3.2.4 Selection of Victims

The obvious strategy is to free the lowest-usage frame in the candidate set. However, we modify this strategy a little to support the following important optimization.

As discussed earlier, HAC relies on the use of an indirection table to achieve low cache replacement overhead. Indirection can increase hit times, because each object access may require dereferencing the indirection entry's pointer to the object, and above all, may introduce an extra cache miss. This overhead is reduced by ensuring the following invariant: *an object for which there is a direct pointer in the stack or registers is guaranteed not to move or be evicted*. The Theta compiler takes advantage of this invariant by loading the indirection entry's pointer into a local variable and using it repeatedly without the indirection overhead; other compilers could easily do the same. We ran experiments to evaluate the effect of pinning frames referenced by the stack or registers and found it had a negligible effect on miss rates.

To preserve the above invariant, the client scans the stack and registers and conservatively determines the frames that are being referenced from the stack. It frees the lowest-usage frame in the candidate set that is not accessible from the stack or registers; if several frames have the same usage, the frame added to the candidate set most recently is selected, since its usage information is most accurate. To make the selection process fast, the candidate set was designed to have $O(\log\ E)$ cost for removing the lowest-usage frame.

When a frame fills up with compacted objects, we compute its usage and insert it in the candidate set. This is desirable because objects moved to that frame may have low usage values compared to pages that are currently present in the candidate set.

The fact that the usage information for some candidates is old does not cause valuable objects to be discarded. If an object in the frame being compacted has been used since that frame was added to the candidate set, it will be retained, since its usage is greater than the threshold. At worst, old frame-usage information may cause us to recover less space from that frame than expected.

3.3 Replacement in the Background

Although our replacement overhead is low, we also have designed the system to allow replacement to be done in the background. HAC always maintains a free frame, which is used to store the incoming page. Another frame must be freed before the next fetch, which can be done while the client waits for the fetch response.

3.4 Summary

The client maintains usage information for each object to differentiate between hot and cold objects. It periodically scans frames in the cache, computes summary usage values for these frames, and enters this information in the candidate set. The candidate set maintains the summary usage information of frames that have been scanned during the last few fetches. When a cache miss occurs, the client fetches the missing page into a free frame, and frees a new frame for the next fetch by compacting the least valuable frames from the candidate set: it copies their hot objects into the current target frame and evicts their cold objects.

4 Performance Evaluation

This section evaluates the performance of HAC based on the usual analytical model of cache performance:

$$\textit{Access time} = \textit{Hit time} + \textit{Miss rate} \times \textit{Miss penalty}$$

Here, *hit time* is the average time to access an object that is present in the client cache and is fully converted to the in-cache format, *miss rate* is the average number of page fetches per object access, and *miss penalty* is the average time to service a client cache miss.

We analyze the performance of HAC by evaluating its effect on each of the terms in the formula. Section 4.2 shows that HAC achieves lower miss rates than page-caching systems and dual-buffering schemes proposed in the literature. Sections 4.3 and 4.4 examine the overhead introduced by HAC on hit time and miss penalty. Section 4.5 analyzes the overall performance of our system, and Section 4.6 shows how it performs when objects are both read and written.

4.1 Experimental Setup

Before presenting the analysis, we describe the experimental setup. Our workloads are based on the OO7 benchmark [CDN94]; this benchmark is intended to match the characteristics of many different CAD/CAM/CASE applications, but does not model any specific application. 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. The *small* database has 20 atomic parts per composite part; the *medium* has 200. In our implementation, the small database takes up 4.2 MB and the medium database takes up 37.8 MB.

The objects in the databases are clustered into 8 KB pages using *time of creation* as described in the OO7 specification [CDN94]. The databases were stored by a server 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 [Sea97].

The databases were accessed by a single client. Both the server and the client ran on DEC 3000/400 Alpha workstations, each with a 133 MHz Alpha 21064 processor, 128 MB of memory and OSF/1 version 3.2. They were connected by a 10 Mb/s Ethernet and had DEC LANCE Ethernet interfaces. The server had a 36 MB cache (of which 6 MB were used for the modified object buffer); we experimented with various sizes for the client cache.

The experiments ran several database traversals that are described below. Both the C code generated by the Theta compiler for the traversals and the system code were compiled using GNU's gcc with optimization level 2.

4.1.1 Traversals

The OO7 benchmark defines several database traversals; these perform a depth-first traversal of the assembly tree and execute an operation on the composite parts referenced by the leaves of this tree. Traversals T1 and T6 are read-only; T1 performs a depth-first traversal of the entire composite part graph, while T6 reads only its *root atomic part*. Traversals T2a and T2b are identical to T1 except that T2a modifies the root atomic part of the graph, while T2b modifies all the atomic parts. Note that T6 accesses many fewer objects than the other traversals.

In general, some traversals will match the database clustering well while others will not, and we believe that on average, one cannot expect traversals to use a large fraction of each page. For example, Tsangaris and Naughton [TN92] found it was possible to achieve good average use by means of impractical and expensive clustering algorithms; an $O(n^{2.4})$ algorithm achieved average use between 17% and 91% depending on the workload, while an $O(n \log n)$ algorithm achieved average use between 15% and 41% on the same workloads. Chang and Katz [CK89] observed that real CAD applications had similar access patterns. Furthermore, it is also expensive to collect the statistics necessary to run good clustering algorithms and to reorganize the objects in the database according to the result of the algorithm [GKM96, MK94]. These high costs bound the achievable frequency of reclusterings and increase the likelihood of mismatches between the current workload and the workload used to train the clustering algorithm; these mismatches can significantly reduce the fraction of a page that is used [TN92].

The OO7 database clustering matches traversal T6 poorly but matches traversals T1, T2a and T2b well; our results show that on average T6 uses only 3% of each page whereas the other traversals use 49%. We defined a new traversal $T1^-$ that uses an average of 27% of a page to represent a more likely clustering quality; we believe that in real workloads, average use would fall between T6 and $T1^-$. We also defined a traversal $T1^+$ that uses 91% of the objects in a page; it allows us to evaluate the impact of HAC in a very unlikely worst case. $T1^+$ and $T1^-$ are similar to T1; $T1^+$ visits all the sub-objects of atomic parts and connections, whereas $T1^-$ stops traversing a composite part graph after it visits half of its atomic parts. Our methodology fixes the way objects are physically clustered in the database and simulates different qualities of clustering by using different traversals.

To analyze the sensitivity of HAC to different workloads, we also designed our own *dynamic traversals* of the OO7 database, which are similar to the multi-user OO7 benchmark [C^+94a], but execute a more mixed set of operations. The dynamic traversals perform a sequence of operations in two medium-sized databases; each operation selects one of the databases randomly, follows a random path down from the root of the assembly tree to a composite part, and randomly executes a $T1^-$, T1 or $T1^+$ traversal of the composite part graph. To model a working set, operations are preferentially executed in one of the two databases. At any given point, one of the databases is the *hot database*, to which 90% of the operations are directed; the remainder go to the *cold database*. Each traversal runs 7500 operations and we time only the last 5000. After the 5000-th operation, there is a shift of working set, and the roles of hot and cold database are reversed.

4.1.2 Parameter Settings

Table 1 shows the values we used for HAC's parameters in all the experiments described in the next sections. To choose these values, we ran various hot traversals including static traversals T1 and T1-, the dynamic traversal with 80% of object accesses performed by $T1^-$ operations and 20% by T1 operations, and another very dynamic *shifting* traversal described by Day [Day95]. The experiments varied the cache sizes and the values of the parameters across the *studied range* reported in the table. The *stable range* column shows the range of parameter values that (for each traversal and each cache size) resulted in an elapsed time within 10% of that obtained for the value we chose.

The performance of HAC improves monotonically within the range studied for R and E, varying little towards the high end of the range. Its behavior relative to changes in N is also monotonic for traversals $T1^-$, T1, and the dynamic traversal, but performance on the shifting traversal degrades for values of N larger than 2.

The only parameter that is not trivial to set is S, the number of frames scanned, because it controls both the speed at which object usage is decayed and the number of frames inserted in the candidate set in each epoch. We picked a value $S = 3$ because it results in good performance for all the experimental points and because the replacement overhead increases quickly with S.

Description	Chosen Value	Studied Range	Stable Range
Retention fraction (R)	0.67	0.5 – 0.9	0.67 – 0.9
Candidate set epochs (E)	20	1 – 500	10 – 500
Secondary scan ptrs (N)	2	0 – 20	2
Frames scanned (S)	3	2 – 20	3

Table 1: Parameter Settings for HAC

4.2 Miss Rate

This section shows that HAC achieves lower miss rates than the best page-caching, and dual-buffering systems in the literature.

4.2.1 Systems Studied

We show that HAC achieves lower miss rates than page-caching systems by comparing it to QuickStore [WD94], which is the best page-caching system in the literature. QuickStore uses a CLOCK algorithm to manage the client cache. Like HAC, it uses 8 KB pages and 32-bit pointers in objects in both the client cache and on disk; its database size is approximately the same size as HAC's, and it is also clustered using *time of creation*, as specified in the OO7 benchmark. It uses an interesting pointer-swizzling technique that stores pointers on disk in their swizzled form, together with a *mapping object* that maps the virtual memory page frame indices of the swizzled pointers to logical page identifiers. When a page is fetched, QuickStore also fetches the page's mapping object and attempts to map the pages that are referenced by the fetched page at the virtual memory addresses specified in the mapping object. If it succeeds, no format conversions are necessary; otherwise, it corrects the pointers in the page to point to a different virtual memory frame. This technique allows QuickStore to use small 32-bit object pointers without severely restricting the maximum database size or introducing any overhead on hit time.

We also compare HAC to a system we created called FPC (fast page caching). FPC is identical to HAC except that it uses a perfect LRU replacement policy to select pages for eviction and always evicts entire pages. We implemented FPC because we wanted to compare the miss rate of HAC and page-caching systems over a wide range of cache sizes and traversals, and only limited experimental results were available for QuickStore. We explain in Section 4.2.3 why experiments showing that HAC outperforms FPC allow us to conclude that it would do even better in a comparison with QuickStore.

To show HAC achieves lower miss rates than dual-buffering systems, we compare its miss rate with that of GOM [KK94], the dual-buffering system with the lowest miss rates in the literature. GOM partitions the client cache *statically* into object and page buffers, each managed using a perfect LRU replacement policy. When there is a cache miss, the missing page is fetched into the page buffer and the least-recently used page P is chosen for eviction. Rather than evicting all the objects in P, GOM copies the recently-used objects in P to the object buffer. If P is refetched, the objects from P that are cached in the object buffer are immediately put back in P. Pages with a large fraction of recently used objects are protected from eviction. To make room for retained objects, the least recently used objects in the object buffer are evicted. To reduce fragmentation, storage is managed using a buddy system. GOM's database is clustered (like ours) using *time of creation*. GOM uses 96-bit pointers and has 12-byte per-object overheads at the server.

Kemper and Kossmann [KK94] show that the cache management strategy used in GOM leads to a lower miss rate than the *eager copying* strategy used by object-caching systems [C$^+$94b, KK90, WD92, KGBW90] which fetch pages from the server. The eager copying strategy copies objects from the page buffer to the object buffer on first use and copies modified objects back to their home pages when a transaction commits. In contrast to GOM, objects can be accessed only when they are in the object buffer; therefore, this buffer is much larger than the page buffer. Since GOM achieves lower miss rates than these object-caching systems, any miss rate reduction HAC achieves relative to GOM would be larger relative to these systems.

4.2.2 Comparison with QuickStore

Table 2 shows the number of fetches for HAC, FPC, and QuickStore for cold traversals T6 and T1 of the medium database. The QuickStore results were obtained with a 12 MB client cache and were reported in [WD94]. HAC and FPC use a smaller cache size, adjusted to account for the size of the indirection table in traversal T1: HAC used a 7.7 MB cache, FPC used a 9.4 MB cache. The indirection table is large because the entries in the table are 16 bytes and the objects accessed by this traversal are small: 29 bytes on average. The overhead for HAC is higher (55%) because most objects in the cache are installed in the indirection table, whereas for FPC it is lower (27%) because only approximately half of the objects in the cache are installed. The unrealistically small object sizes of the OO7 database represent nearly a worst case for the indirection-table overhead.

	T6	T1
QuickStore	610	13216
HAC	506	10266
FPC	506	12773

Table 2: Misses, Cold traversals, Medium database

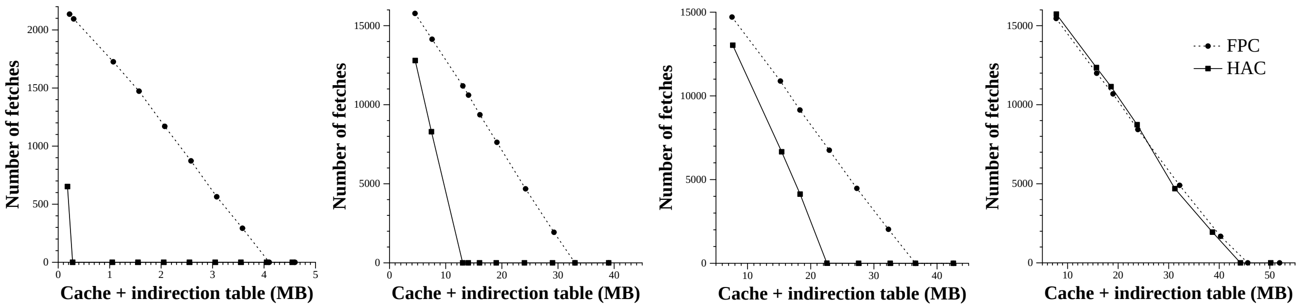
HAC and FPC have the same number of misses in traversal T6 because they are all cold cache misses. In traversal T1, HAC has 24% fewer fetches than FPC, because it has 38% fewer capacity misses (there are 3662 cold cache misses), due to the use of object caching. QuickStore has a higher miss rate than both systems in both traversals: HAC and FPC have 21% fewer misses than QuickStore in T6 because of the extra misses required to fetch the mapping objects, and FPC has 3% fewer fetches than QuickStore in T1 — mainly for the same reason — but also because FPC uses perfect LRU whereas QuickStore uses CLOCK.

The miss rate reduction achieved by HAC relative to the two page-caching systems is not very impressive for this particular traversal and cache size because T1 is a traversal with very good clustering (as discussed in Section 4.1.1), and the cache can fit only 55% of the objects accessed by the traversal.

4.2.3 Additional Experiments

This section presents results of experiments comparing the miss rates of HAC and FPC. We argue that for all traversals with equal or lower clustering quality than T1, any miss rate reductions HAC achieves relative to FPC would be even higher relative to QuickStore. This is true because FPC outperforms QuickStore in traversal T1, and as clustering becomes worse, FPC's performance becomes relatively better. As clustering degrades, FPC's overhead decreases because traversals with worse clustering access a lower fraction of objects per page, resulting in a smaller number of indirection table entries and more space to store objects; QuickStore's overhead remains constant because it always fetches one mapping object per page, regardless of the quality of clustering. For traversals with better clustering than T1, QuickStore might outperform FPC, but we expect these traversals to be very uncommon, as discussed in Section 4.1.1.

The first set of additional experiments ran traversals T6, T1$^-$, T1, and T1$^+$ on the medium database with varying client cache sizes starting both from cold and hot caches. Figure 5 shows the number of misses of HAC and FPC for the hot traversals. The x-axis shows the sum of the client cache size and the indirection table size; since FPC uses less space in the indirection table than HAC, at any particular size, it has more space in the cache than HAC does. We do not present the graphs for the cold traversals because they are



Bad clustering (T6) Average clustering (T1-) Good clustering (T1) Excellent clustering (T1+)

Figure 5: Client cache misses, Hot traversals, Medium database

similar; the main difference is that they include the constant cost of cold cache misses (506 for T6 and 3662 for the other traversals).

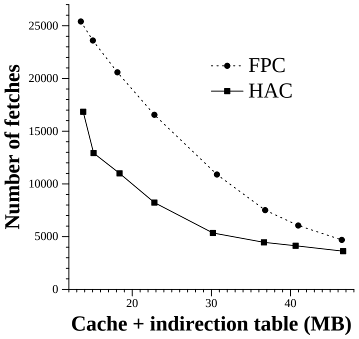
For the excellent-clustering case of traversal T1$^+$, HAC and FPC have an almost identical miss rate, because HAC behaves like a page-caching system when clustering is very good. This traversal accesses an average fraction of objects per page that is larger than the retention fraction R and, since there is good spatial locality, these objects have identical usage values. Therefore, the threshold value selected by HAC to compact a page will frequently cause all its objects to be discarded.

For the other traversals, when the cache is very small or very large, the miss rates of HAC and FPC are similar: in the first case, neither of the two systems is able to retain many useful objects in the cache resulting in a high miss rate for both of them; and if the cache is large enough to hold all the pages used by the traversal, both HAC and FPC perform equally well since there are no cache misses. HAC achieves much lower miss rates than FPC in the middle range because it can cache useful objects without needing to cache their pages; e.g., for the average clustering case (T1$^-$), HAC achieves a maximum reduction in the number of fetches relative to FPC of 11192 (FPC has 11192 fetches and HAC has none).

HAC is space-efficient: it needs only 11% more cache space than the bare minimum to run T1$^-$ without cache misses, and it needs only 1% more than the minimum if the number of secondary scan pointers is increased to 23. HAC requires 20 times less memory than FPC to run traversal T6 without cache misses, 2.5 times less memory to run traversal T1$^-$ and 62% less memory to run traversal T1; as expected, these gains decrease as the quality of clustering increases.

The second set of experiments ran our dynamic traversals. The results were qualitatively similar to the ones described above: the performance of HAC and FPC was similar when clustering was excellent (i.e., when all operations executed T1$^+$ traversals of the composite parts), and HAC achieved much lower miss rates when most of the operations executed traversals T6, T1$^-$ and T1.

Figure 6 presents miss rates for a dynamic traversal where each operation randomly executes T1$^-$ or T1 such that 80% of the object accesses are performed by T1$^-$ operations and 20% by T1 operations (above-average clustering quality).



Good clustering (80% T1-, 20% T1)

Figure 6: Client cache misses, Dynamic traversal

4.2.4 Comparison with GOM

We now compare HAC's performance with that of GOM. Figure 7 presents fetch counts for GOM, HAC, and HAC-BIG running a cold T1 traversal of the small OO7 database with varying client cache sizes. Since GOM partitions the client cache into object and page buffers *statically*, all data for GOM were obtained by manual tuning of the buffer sizes to achieve "the best possible" [KK94] performance; the sizes of the buffers were tuned for each cache size and for each traversal (e.g., tuning was different for T1 and T2b). The results for GOM were obtained from Kossmann [Kos97].

We introduce HAC-BIG to separate the performance effects of smaller objects and better cache management, which in combination cause HAC to outperform GOM at all cache sizes. HAC-BIG is like HAC except that we padded its objects to be approximately the same size as GOM's (HAC-BIG's database is actually 6% larger than GOM's). The differences between HAC and HAC-BIG result from using smaller

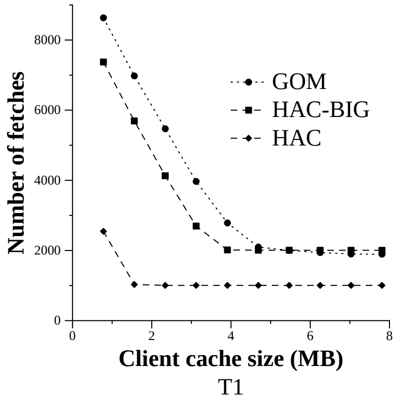


Figure 7: Client cache misses, Cold T1 Traversal, Small database

objects, and the differences between HAC-BIG and GOM result from the better cache management in HAC-BIG. (Despite using page caching, even FPC has a lower miss rate than GOM for all cache sizes, because it uses small objects.)

Both HAC and HAC-BIG use 4 KB pages like GOM. We conservatively did not correct the cache sizes for HAC and HAC-BIG to account for our 16-byte indirection table entries because GOM uses a resident object table that introduces a 36-byte overhead for each object [Kos95, Kos97].

The most important point is that HAC-BIG outperforms GOM even though GOM's results required manual tuning of the sizes of the object and page buffers. Of course, in a real system, such tuning would not be possible, and a poor adjustment can hurt performance. By contrast, HAC adapts dynamically to different workloads with no user intervention.

GOM requires a cache size of 4.7 MB to hold all the objects accessed by the traversal whereas HAC-BIG only requires 3.9 MB. The difference is due to two problems with GOM's cache management scheme: storage fragmentation and static partitioning of the cache between page and object buffers (which causes space to be wasted by useless objects contained in cached pages). Note that reducing the size of GOM's page buffer does not necessarily improve performance, because GOM's page buffer is already too small to hold some pages long enough for the traversal to access all their objects of interest. This causes GOM to refetch some pages even when all the objects accessed by the traversal fit in the cache, e.g., for a cache size of 4.7 MB, GOM refetches 11% of the pages.

Finally, GOM incurs overheads that we did not account for in this comparison: it uses perfect LRU, which introduces a large overhead on hit time, and its resident object table entries are 20 bytes larger than ours.

4.3 Hit Time

This section evaluates the overhead that HAC adds to hit time. Our design includes choices (such as indirection) that penalize hit time to reduce miss rate and miss penalty; this section shows that the price we pay for these choices is modest.

We compare HAC with C++ running hot T6 and T1 traversals of the medium database. HAC runs with a 30 MB client cache to ensure that there are no misses and no format conversions during these traversals. The C++ program runs the traversals without paging activity on a database created on the heap. The C++ version does *not* provide the same facilities as our system; for example, it does not support transactions. However, the C++ and HAC codes are very similar because both follow the OO7 specification closely, and both are compiled using GNU's gcc with optimization level 2.

	T1 (sec)	T6 (msec)
Exception code	0.86	0.81
Concurrency control checks	0.64	0.62
Usage statistics	0.53	0.85
Residency checks	0.54	0.37
Swizzling checks	0.33	0.23
Indirection	0.75	0.00
C++ traversal	4.12	6.05
Total (HAC traversal)	7.77	8.93

Table 3: Breakdown, Hot Traversals, Medium Database

Table 3 shows where the time is spent in HAC. This breakdown was obtained by removing the code corresponding to each line and comparing the elapsed times obtained with and without that code. Therefore, each line accounts not only for the overhead of executing extra instructions but also for the performance degradation caused by code blowup. To reduce the noise caused by conflict misses, we used *cord* and *ftoc*, two OSF/1 utilities that reorder procedures in an executable to reduce conflict misses. We used cord on all HAC executables and on the C++ executable; as a result the total traversal times for HAC are better than the ones presented in Section 4.5.

The first two lines in the table are not germane to cache management. The *exception code* line shows the cost introduced by code to generate or check for various types of exceptions (e.g., array bounds and integer overflow). This overhead is due to our implementation of the type-safe language Theta [LAC$^+$96]. The *concurrency control checks* line shows what we pay for providing transactions.

The next four lines are related to our cache management scheme: *usage statistics* accounts for the overhead of maintaining per-object usage statistics, *residency checks* refers to the cost of checking indirection table entries to see if the

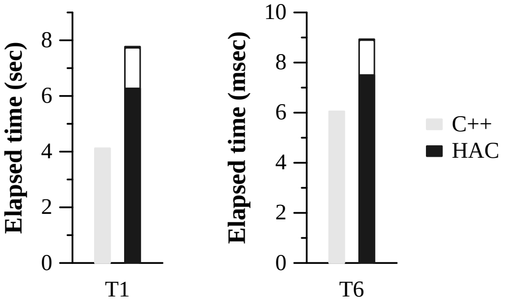


Figure 8: Elapsed time, Hot traversals, Medium database

object is in the cache; *swizzling checks* refers to the code that checks if pointers are swizzled when they are loaded from an object; and *indirection* is the cost of accessing objects through the indirection table. The indirection costs were computed by subtracting the elapsed times for the C++ traversals from the elapsed times obtained with a HAC executable from which the code corresponding to all the other lines in the table had been removed. Since HAC uses 32-bit pointers and the C++ implementation uses 64-bit pointers, we also ran a version of the same HAC code configured to use 64-bit pointers. The resulting elapsed times were within 3% of the ones obtained with the 32-bit version of HAC for both traversals, showing that the different pointer size does not affect our comparison. Note that we have implemented techniques that can substantially reduce the residency and concurrency control check overheads but we do not present the results here.

Figure 8 presents elapsed time results; we subtract the costs that are not germane to cache management (shown in white). The results show that the overheads introduced by HAC on hit time are quite reasonable; HAC adds an overhead relative to C++ of 52% on T1 and 24% on T6. Furthermore, the OO7 traversals exacerbate our overheads, because *usage statistics*, *residency checks* and *indirection* are costs that are incurred once per method call, and methods do very little in these traversals: assuming no stalls due to the memory hierarchy, the average number of cycles per method call in the C++ implementation is only 24 for T1 and 33 for T6. HAC has lower overheads in T6 mainly because the indirection overhead is negligible for this traversal. This happens because all the indirection table entries fit in the processor's second level cache and because our optimizations to avoid indirection overheads are very effective for this traversal.

The overheads introduced by HAC on hit time are even lower on modern processors; we ran the two traversals in a 200 MHz Intel Pentium Pro with a 256 KB L2 cache and the total overheads relative to the C++ traversals (on the same processor) decreased by 11% for T1 and 60% for T6.

It is interesting to note that our compaction scheme can actually speed up some traversals; running T6 with the minimum cache size for which there are no cache misses reduces the elapsed time by 24%. This happens because all objects accessed by the traversal are compacted by HAC into 19 pages, eliminating data TLB misses and improving spatial locality within cache lines. This effect does not occur with T1 because it accesses a much larger amount of data.

Another interesting observation, which puts our indirection table space overhead in perspective, is that the use of 64-bit pointers in the C++ program (and a small amount of space wasted by the memory allocator) result in a database that is 54% larger than ours; this overhead happens to be two times larger than the space overhead introduced by our indirection table during traversal T1.

4.4 Miss Penalty

This section shows that our miss penalty is dominated by disk and network times. To better characterize the techniques that affect miss penalty, we break it down as:

$$\begin{aligned}\textit{Miss penalty} &= \textit{Fetch time} + \textit{Replacement overhead}\\ &\quad + \textit{Conversion overhead}\end{aligned}$$

Here, *fetch time* is the average time to fetch a page from the server, *replacement overhead* is the average time to free a page frame in the client cache to hold a fetched page, and *conversion overhead* is the average time per fetch to convert fetched objects from their on-disk to their in-cache format (i.e., install them in the indirection table and swizzle their pointers).

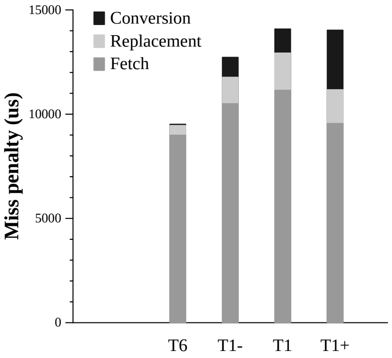


Figure 9: Client cache miss penalty

Figure 9 presents a breakdown of HAC's miss penalty for the static traversals. The miss penalty was measured for the experimental point where replacement overhead was maximal for each traversal. This point corresponded to hot traversals with cache sizes of 0.16 MB for T6, 5 MB for $T1^{-}$, 12 MB for T1 and 20 MB for $T1^{+}$.

Conversion overhead is the smallest component of the miss penalty for all traversals but $T1^{+}$, which we believe is not a realistic workload. The conversion overhead grows

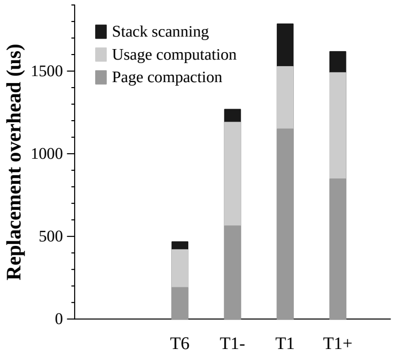


Figure 10: Worst case replacement overhead

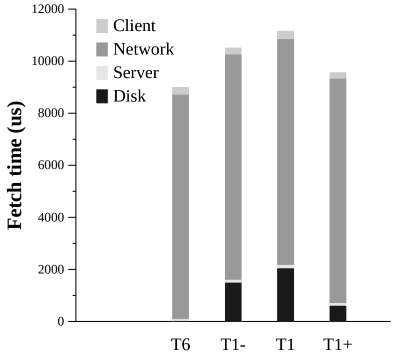


Figure 11: Fetch time

with the quality of clustering because both the number of installed objects per page and the number of swizzled pointers per page increase. The conversion overhead is relatively low: it is 0.7% of the fetch time for T6, 9% for T1$^-$, and 10% for T1. Let us compare this conversion overhead to that in QuickStore. QuickStore's conversion overhead includes the time to process its mapping objects, and can easily exceed HAC's conversion overhead when clustering is poor and mapping objects are large. Furthermore, QuickStore's conversion overhead is much higher than HAC's when it is not possible to map disk pages to the virtual memory frames described by its mapping objects; the authors of [WD94] present results that show an increase in the total time for a cold T1 traversal of 38% in this case.

Figure 9 shows the overhead when replacement is performed in the foreground; it is 5% of the fetch time for T6, 12% for T1$^-$, 16% for T1, and 17% for T1$^+$. Since this overhead is much lower than the fetch time, HAC is able to perform replacement with no cost if the processor is idle waiting for the fetch reply; otherwise, the actual replacement cost will be between zero and the foreground replacement overhead reported in Figure 9.

Figure 10 presents a breakdown of the foreground replacement overhead for HAC. As the figure shows, replacement overhead grows with the quality of clustering, except that it is lower for T1$^+$ than for T1. This happens mainly because the fraction of a fetched page that survives compaction grows, causing an increase in the page compaction time. The exception for T1$^+$ occurs because HAC tends to evict entire pages for traversals with extremely good clustering. The cost of stack scanning is significantly lower than the cost of usage computation and page compaction. However, it is high in traversal T1 (14% of the total) because the OO7 traversals are implemented using recursion, which results in large stack sizes; we expect the cost of stack scanning to be lower for most applications.

Figure 11 presents a breakdown of the fetch time. The most important observation is that fetch time is completely dominated by the time spent transmitting the fetch request and reply over the network, and by the time spent reading pages from disk at the server. The cost labeled *client* corresponds to the overhead of registering pages in the client cache and *server* corresponds to the cache bookkeeping cost at the server and the additional overhead to support transactions.

For most system configurations, the cost of foreground replacement and format conversions as a fraction of fetch time will be lower than the values we report. This is clearly true when a slower network is used, e.g., client and server connected by a wireless network or a WAN, but the use of a faster network could presumably increase the relative cost. We believe that the reduction in network time would be offset by the following factors. First, in our experimental configuration, the disk time is only a small fraction of the total fetch time because the hit rate in the server cache is unrealistically high; T1 has the lowest server cache hit rate and it is still 86%. In a configuration with a realistic hit rate of 23% (measured by Blaze [Bla93] in distributed file systems), the foreground replacement overheads in T1$^-$ and T1 would be 11% and 16% of the disk time alone. Second, our server was accessed by a single client; in a real system, servers may be shared by many clients, and contention for the servers' resources will lead to even more expensive fetches. Third, our client machine is a DEC 3000/400 with a 133 MHz Alpha 21064 processor, which is slow compared to more recent machines; our overheads will be lower in today's machines. For example, we reran the experiments described in this section on a 200 MHz Intel Pentium Pro with a 256 KB L2 cache. The conversion overheads on this machine relative to the overheads on the Alpha workstation were 33% for T1$^-$ and 42% for T1; the foreground replacement overheads were 75% for T1$^-$ and 77% for T1. Note that the speedup of the replacement overhead should be larger in a machine with a larger second level cache.

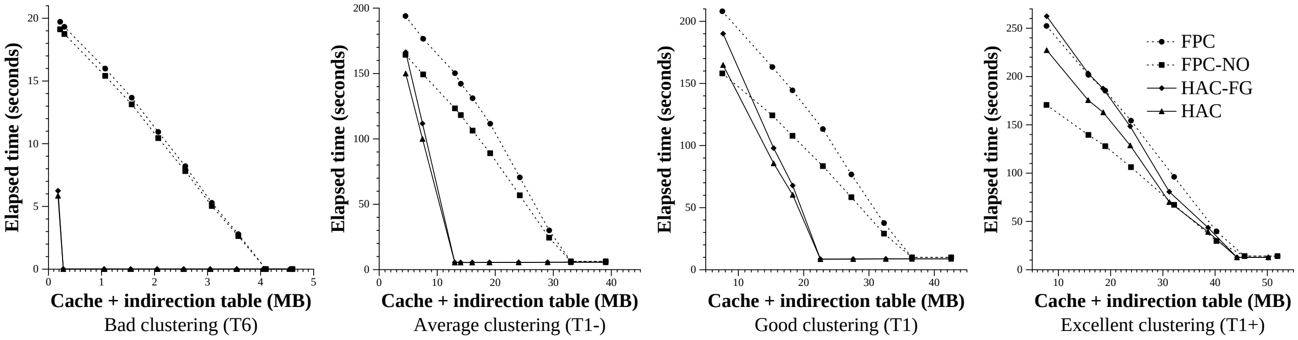


Figure 12: Elapsed time, Hot traversals, Medium database

4.5 Overall Performance

The previous sections showed that HAC achieves low miss rates, hit times and miss penalties. This section shows the effect on overall performance by comparing elapsed times (not including commit time) for HAC and FPC. We are not able to compare HAC with QuickStore directly because QuickStore runs on different equipment. However, our results show that Thor-1 with HAC will outperform QuickStore on realistic workloads, as we argue below.

We present results for both the base version of HAC, which performs replacement in the background while waiting for replies to fetch requests, and HAC-FG, a version of HAC modified to perform replacement in the foreground. In all our experiments, the time to perform replacement was lower than the idle time during fetches and, therefore, HAC had zero replacement overhead. HAC-FG corresponds to the worst case replacement overhead, i.e., when there is no idle time between fetches because of multithreading or multiprogramming in the client machine.

We also present results for two versions of FPC that bound the range of performance we would expect from a fast page-caching system: the base version and FPC-NO, which is an *unrealistic* system that incurs no overheads. The elapsed times of FPC-NO were obtained by subtracting the replacement and conversion overheads from the elapsed time of the base version. Both versions use a previously collected execution trace to implement perfect LRU without any overhead.

We claim that any performance gains HAC or HAC-FG achieve relative to FPC-NO, when running traversals with equal or lower clustering quality than T1, would be even higher relative to QuickStore. This is true because FPC-NO has low hit time (as low as HAC's), no replacement and conversion overheads, and lower miss rates than QuickStore for traversals with equal or lower clustering quality than T1. In the analysis that follows, we concentrate on the worst-case comparison for HAC — HAC-FG vs. FPC-NO; the gains of HAC relative to a more realistic page-caching system would be higher.

Figure 12 shows elapsed times we measured running hot T6, $T1^-$, T1, and $T1^+$ medium traversals. The performance of HAC-FG is worse than the performance of FPC-NO for the traversal $T1^+$, because the miss rates of both systems are almost identical and FPC-NO has no overheads. However, the lines for the two versions of HAC are contained within the range defined by the two FPC lines. Therefore, the performance of HAC should be similar to the performance of realistic implementations of FPC even for this worst case.

For more realistic traversals, when the cache is very large, the performance of HAC-FG and FPC-NO is similar because the cache can hold all the pages touched by the traversal and the hit times of both systems are similar. When the cache is very small, FPC-NO can outperform HAC-FG because the miss rates of the two systems are similar and FPC-NO has no overheads, but this region is not interesting because both systems are thrashing. The gains of HAC-FG relative to FPC-NO are substantial in the middle region because it has much lower miss rates. The maximum performance difference between HAC-FG and FPC-NO occurs for the minimum cache size at which all the objects used by the traversal fit in the client cache; HAC-FG performs up to approximately 2400 times faster than FPC-NO in T6, 23 times faster in $T1^-$, and 10 times faster in T1.

Figure 13 shows the elapsed times we measured for the dynamic traversal that executes 80% of the object accesses in $T1^-$ operations and 20% in T1 operations. HAC-FG performs up to 61% faster than FPC-NO in this workload.

We also compared the performance of Thor-1 with Thor-0, our previous implementation, which had been shown to outperform all other systems as long as the working set of a traversal fit in the client cache [LAC$^+$96]. We found that HAC enabled Thor-1 to outperform Thor-0 on all workloads, and to do substantially better on traversals where the working set did not fit in the client cache.

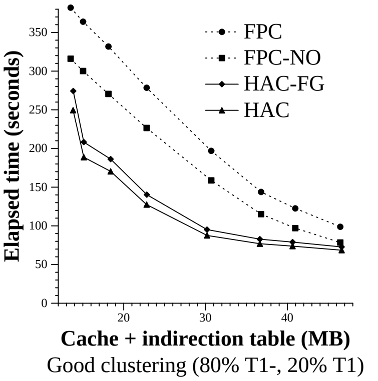


Figure 13: Elapsed time, Dynamic traversal

4.6 Traversals With Updates

All the experiments we presented so far ran read-only traversals. For completeness, we now show results for traversals with updates. Figure 14 presents elapsed times for cold traversals T1, T2a and T2b of the medium database running with a 12 MB client cache. The figure also shows the time to commit the transactions.

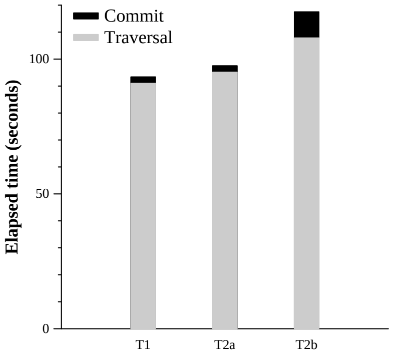


Figure 14: Elapsed time, Cold traversals, Medium database

HAC uses a *no-steal* [GR93] cache management policy: modifications are sent to the server only when a transaction commits, and the modified objects cannot be evicted from the client cache until this happens. We claim there is no significant loss in functionality or performance in our system due to the lack of a steal approach. This approach may be needed with page caching; otherwise, the cache could fill up with pages containing modified objects, leaving no room to fetch more pages needed by a large transaction. However, since object caching retains only the modified objects and not their pages, it is unlikely that the cache will fill up. Our claim is supported by the results presented in Figure 14: HAC is capable of running traversal T2b in a single transaction even though this transaction reads and writes an extremely large number of objects (it reads 500000 objects and writes 100000). Furthermore, T2b runs only 26% slower than T1 in HAC, whereas in QuickStore T2b runs 3.5 times slower than T1 (mostly because of the recovery overhead of shipping updates to the server due to insufficient client cache space) [WD94].

Object-caching systems must ship modified objects to servers at commit time. This can lead to poor performance if it is necessary to immediately read the objects' pages from disk in order to install the objects in their pages. We avoid this performance problem using the *modified object buffer* architecture [Ghe95], which allows these installations to be performed in the background.

5 Conclusions

This paper has described a new technique for managing the client cache in a distributed persistent object system. HAC is a hybrid between page and object caching that combines the virtues of both while avoiding their disadvantages: it achieves the low overheads of a page-caching system, but does not have high miss rates when spatial locality is poor; and it achieves the low miss rates of object-caching systems, while avoiding their problems of storage fragmentation and high overheads for managing the cache at a fine granularity. Furthermore, HAC is adaptive: when spatial locality is good, it behaves like a page-caching system, avoiding the overheads of object caching where it has low benefit. If spatial locality is poor, it behaves like an object-caching system, taking advantage of the low miss rates to offset the increased overheads. It is able to dynamically adjust the amount of cache space devoted to pages so that space in the cache can be used in the way that best matches the needs of the application.

The paper compares HAC to QuickStore, the best page-caching system in the literature, and shows that HAC significantly outperforms QuickStore for the range of spatial locality that can be expected using practical clustering algorithms. The paper also shows HAC outperforms GOM, the best dual-buffering system in the literature. Since GOM was shown to have lower miss rates than object-caching systems and our overheads are very low, we claim HAC will also outperform the best object-caching systems. Therefore, we believe hybrid adaptive caching is the cache management technique of choice for distributed, persistent object storage systems.

Acknowledgements

We would like to thank Jeff Chase (our shepherd), Darrell Anderson, Kavita Bala, Phillip Bogle, Chandrasekhar Boyapati, John Chapin, Markos Zaharioudakis, and the anonymous referees for their helpful comments. Donald Kossmann provided us with additional data for his dual-buffering system. We would also like to thank other members of the Programming Methodology group who contributed to the Thor project — Dorothy Curtis, Paul Johnson and Umesh

Maheshwari. Liuba Shrira was involved in the initial discussions of the design.

References

[AGLM95] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. In *ACM SIGMOD Int. Conf. on Management of Data*, pages 23–34, San Jose, CA, May 1995.

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

[Bla93] M. Blaze. Caching in Large-Scale Distributed File Systems. Technical Report TR-397-92, Princeton University, Jan. 1993.

[BOS91] P. Butterworth, A. Otis, and J. Stein. The GemStone database management system. *Comm. of the ACM*, 34(10):64–77, Oct. 1991.

[C+89] M. Carey et al. Storage management for objects in EXODUS. In W. Kim and F. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*. Addison-Wesley, 1989.

[C+94a] M. J. Carey et al. A Status Report on the OO7 OODBMS Benchmarking Effort. In *ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 414–426, 1994.

[C+94b] M. J. Carey et al. Shoring up persistent applications. In *ACM SIGMOD Int. Conf. on Management of Data*, pages 383–394, Minneapolis, MN, May 1994.

[CAL97] M. Castro, A. Adya, and B. Liskov. Lazy reference counting for transactional storage systems. Technical Report MIT-LCS-TM-567, MIT Lab for Computer Science, June 1997.

[CDN94] 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. ftp://ftp.cs.wisc.edu/OO7.

[CFZ94] M. Carey, M. Franklin, and M. Zaharioudakis. Fine-Grained Sharing in a Page Server OODBMS. In *ACM SIGMOD Int. Conf. on Management of Data*, pages 359–370, may 1994.

[CK89] E. E. Chang and R. H. Katz. Exploiting inheritance and structure semantics for effective clustering and buffering in an object-oriented dbms. In *ACM SIGMOD Int. Conf. on Management of Data*, pages 348–357, Portland, OR, May 1989.

[CLFL95] J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska. Sharing and protection in a single-address-space operating system. *ACM Transactions on Computer Systems*, 12(1):271–307, Feb. 1995.

[Cor69] F. J. Corbato. A Paging Experiment with the Multics System, *in Festschrift: In Honor of P. M. Morse*, pages 217–228. MIT Press, 1969.

[D+90] O. Deux et al. The story of O2. *IEEE Trans. on Knowledge and Data Engineering*, 2(1):91–108, Mar. 1990.

[Day95] M. Day. Client cache management in a distributed object database. Technical Report MIT/LCS/TR-652, MIT Laboratory for Computer Science, 1995.

[DLMM94] M. Day, B. Liskov, U. Maheshwari, and A. C. Myers. References to remote mobile objects in Thor. *ACM Letters on Programming Languages and Systems (LOPLAS)*, pages 115–126, Mar. 1994.

[Ghe95] S. Ghemawat. *The Modified Object Buffer: a Storage Manamement Technique for Object-Oriented Databases*. PhD thesis, Massachusetts Institute of Technology, 1995.

[GKM96] C. Gerlhof, A. Kemper, and G. Moerkotte. On the cost of monitoring and reorganization of object bases for clustering. *Sigmod Record*, 25(3):22–27, September 1996.

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

[Gru97] R. Gruber. *Optimism vs. Locking: A Study of Concurrency Control for Client-Server Object-Oriented Databases*. PhD thesis, MIT, Feb. 1997.

[JS94] T. Johnson and D. Shasha. A low overhead high performance buffer replacement algorithm. In *Proceedings of International Conference on Very Large Databases*, pages 439–450, 1994.

[KGBW90] W. Kim, J. F. Garza, N. Ballou, and D. Woelk. Architecture of the ORION next-generation database system. *IEEE Trans. on Knowledge and Data Engineering*, 2(1):109–124, Mar. 1990.

[KK90] T. Kaehler and G. Krasner. *LOOM—Large Object-Oriented Memory for Smalltalk-80 Systems*, pages 298–307. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1990.

[KK94] A. Kemper and D. Kossmann. Dual-buffer strategies in object bases. In *20th Int. Conf. on Very Large Data Bases (VLDB)*, pages 427–438, Santiago, Chile, 1994.

[Kos95] D. Kossmann. *Efficient Main-Memory Management of Persistent Objects*. Shaker-Verlag, 52064 Aachen, Germany, 1995. PhD thesis, RWTH Aachen.

[Kos97] D. Kossmann. Private communication. June 30, 1997.

[LAC+96] B. Liskov, A. Adya, M. Castro, M. Day, S. Ghemawat, R. Gruber, U. Maheshwari, A. C. Myers, and L. Shrira. Safe and efficient sharing of persistent objects in Thor. In *ACM SIGMOD Int. Conf. on Management of Data*, pages 318–329, June 1996.

[LCD+94] B. Liskov, D. Curtis, M. Day, S. Ghemawhat, R. Gruber, P. Johnson, and A. C. Myers. Theta reference manual. Programming Methodology Group Memo 88, MIT Lab. for Computer Science, Feb. 1994. Also available at http://www.pmg.lcs.mit.edu/papers/thetaref/.

[LLOW91] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore database system. *Comm. of the ACM*, 34(10):50–63, Oct. 1991.

[MBMS95] J. C. Mogul, J. F. Barlett, R. N. Mayo, and A. Srivastava. Performance Implications of Multiple Pointer Sizes. In *USENIX 1995 Tech. Conf. on UNIX and Advanced Computing Systems*, pages 187–200, New Orleans, LA, 1995.

[MK94] W. J. McIver and R. King. Self adaptive, on-line reclustering of complex object data. In *ACM SIGMOD Int. Conf. on Management of Data*, pages 407–418, Minneapolis, MN, May 1994.

[Mos90] J. E. B. Moss. Design of the Mneme persistent object store. *ACM Transactions on Information Systems (TOIS)*, 8(2):103–139, Apr. 1990.

[Mos92] J. E. B. Moss. Working with persistent objects: To swizzle or not to swizzle. *IEEE Transactions on Software Engineering*, 18(3):657–673, August 1992.

[MS95] M. McAuliffe and M. Solomon. A trace-based simulation of pointer swizzling techniques. In *Int. Conf. on Data Engineering (ICDE)*, pages 52–61, Mar. 1995.

[Ont92] Ontos, Inc., Lowell, MA. *Ontos Reference Manual*, 1992.

[OOW93] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K page replacement algorithm for database disk buffering. In *ACM SIGMOD Int. Conf. on Management of Data*, pages 297–306, Washington, D.C., May 1993.

[OS94] J. O'Toole and L. Shrira. Opportunistic log: Efficient installation reads in a reliable object server. In *Proceedings of the Symp. on Operating System Design and Implementation (OSDI)*, pages 39–48, Monterey, CA, 1994.

[OS95] J. O'Toole and L. Shrira. Shared data management needs adaptive methods. In *Proceedings of IEEE Workshop on Hot Topics in Operating Systems*, Orcas Island, 1995.

[RD90] J. Robinson and N. Devarakonda. Data cache management using frequency-based replacement. In *Proceedings of ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 134–142, 1990.

[Sea97] Seagate Technology, Inc. http://www.seagate.com/, 1997.

[SKW92] V. Singhal, S. V. Kakkad, and P. R. Wilson. Texas: An efficient, portable persistent store. In *5th Int. Workshop on Persistent Object Systems (POS)*, pages 11–33, San Miniato, Italy, Sept. 1992.

[Sta84] J. W. Stamos. Static grouping of small objects to enhance performance of a paged virtual memory. *ACM Trans. on Programming Languages and Systems (TOPLAS)*, 2(2):155–180, May 1984.

[TN92] M. Tsangaris and J. F. Naughton. On the performance of object clustering techniques. In *ACM SIGMOD Int. Conf. on Management of Data*, pages 144–153, San Diego, CA, June 1992.

[WD92] S. White and D. DeWitt. A performance study of alternative object faulting and pointer swizzling strategies. In *18th Int. Conf. on Very Large Data Bases (VLDB)*, pages 419–431, Vancouver, British Columbia, Aug. 1992.

[WD94] S. J. White and D. J. DeWitt. QuickStore: A high performance mapped object store. In *ACM SIGMOD Int. Conf. on Management of Data*, pages 395–406, Minneapolis, MN, May 1994.

[WD95] S. J. White and D. J. DeWitt. Implementing crash recovery in QuickStore: A performance study. In *ACM SIGMOD Int. Conf. on Management of Data*, pages 187–198, San Jose, CA, June 1995.