Providing Persistent Objects in Distributed Systems

Barbara Liskov, Miguel Castro, Liuba Shrira*, Atul Adya

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

Abstract. THOR is a persistent object store that provides a powerful programming model. THOR ensures that persistent objects are accessed only by calling their methods and it supports atomic transactions. The result is a system that allows applications to share objects safely across both space and time.
The paper describes how the THOR implementation is able to support this powerful model and yet achieve good performance, even in a wide-area, large-scale distributed environment. It describes the techniques used in THOR to meet the challenge of providing good performance in spite of the need to manage very large numbers of very small objects. In addition, the paper puts the performance of THOR in perspective by showing that it substantially outperforms a system based on memory mapped files, even though that system provides much less functionality than THOR.

1 Introduction

Persistent object stores provide a powerful programming model for modern applications. Objects provide a simple and natural way to model complex data; they are the preferred approach to implementing systems today. A persistent object store decouples the object model from individual programs, effectively providing a persistent heap that can be shared by many different programs: objects in the heap survive and can be used by applications so long as they can be referenced. Such a system can guarantee *safe sharing*, ensuring that all programs that use a shared object use it in accordance with its type (by calling its methods). If in addition the system provides support for atomic transactions, it can guarantee that concurrency and failures are handled properly. Such a platform allows sharing of objects across both space and time: Objects can be shared between applications running now and in the future. Also, objects can be used concurrently by applications running at the same time but at different locations: different processors within a multiprocessor; or different processors within a distributed environment.

A long-standing challenge is how to implement this programming model so as to provide good performance. A major impediment to good performance in persistent object stores is the need to cope with large numbers of very small objects. Small objects

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.

* Current address: Department of Computer Science, Brandeis University, Waltham, MA 02254.

can lead to overhead at multiple levels in the system and affect the cost of main memory access, cache management, concurrency control and recovery, and disk access.

This paper describes how this challenge is met in the THOR system. THOR provides a persistent object store with type-safe sharing and transactions. Its implementation contains a number of novel techniques that together allow it to perform well even in the most difficult environment: a very large scale, wide-area distributed system. This paper pulls together the entire THOR implementation, explaining how the whole system works.

THOR is implemented as a client/server system in which servers provide persistent storage for objects and applications run at client machines on cached copies of persistent objects. The paper describes the key implementation techniques invented for THOR: the CLOCC concurrency control scheme, which provides object-level concurrency control while minimizing communication between clients and servers; the MOB disk management architecture at servers, which uses the disk efficiently in the presence of very small writes (to individual objects); and the HAC client caching scheme, which provides the high hit rates of an object caching scheme with the low overheads of a page caching scheme. In effect, the THOR implementation takes advantage of small objects to achieve good performance, thus turning a liability into a benefit.

The paper also presents the results of experiments that compare the performance of THOR to that of C++/OS. C++/OS represents a well-known alternative approach to persistence; it uses memory mapped files to provide persistent storage for objects, and a 64-bit architecture to allow addressing of very large address spaces. This system does not, however, provide the functionality of THOR, since it supports neither safe sharing nor atomic transactions. The performance comparison between THOR and C++/OS is interesting because the latter approach is believed to deliver very high performance. However, our results show that THOR substantially outperforms C++/OS in the common cases: when there are misses in the client cache, or when objects are modified.

The rest of this paper is organized as follows. We describe the THOR model in Section 2. Section 3 presents an overview of our implementation architecture. Sections 4, 5, and 6 describe the major components of the implementation. Our performance experiments are described in Section 7, and conclusions are presented in Section 8.

2 Thor

THOR provides a universe of persistent objects. Each object in the universe has a unique identity, a state, and a set of methods; it also has a type that determines its methods and their signatures. The universe is similar to the heap of a strongly-typed language, except that the existence of its objects is not linked to the running of particular programs. The universe has a persistent root object. All objects reachable from the root are persistent; objects that are no longer accessible from the root are garbage collected.

Applications use THOR objects by starting a THOR *session*. Within a session, an application performs a sequence of *transactions*; a new transaction is started each time the previous one completes. A transaction consists of one or more calls to methods of THOR objects. The application code ends a transaction by requesting a commit or abort. A commit request may fail (causing an abort); if it succeeds, THOR guarantees

that the transaction is serialized with respect to all other transactions and that all its modifications to the persistent universe are recorded reliably. If the transaction aborts, it is guaranteed to have no effect and all its modifications are discarded.

THOR objects are implemented using a type-safe language called Theta [LCD+94, DGLM95]. A different type-safe language could have been used, and we have subsequently provided a version of THOR in which objects are implemented using a subset of Java. A number of issues arise when switching to Java, e.g., what to do if an object that cannot be made persistent, such as a thread in the current application, becomes reachable from the persistent root. These issues are discussed in [Boy98].

Applications that use THOR need not be written in the database language, and in fact can be written in many different languages, including unsafe ones like C and C++. THOR supports such *heterogeneous sharing* by providing a small layer of code called a *veneer*. A veneer consists of a few procedures that the application can call to interact with THOR (e.g., to start up a session or commit a transaction), together with a *stub* for each persistent type; to call a method on a THOR object, the application calls the associated method on a stub object. More information about veneers can be found in [LAC+96, BL94].

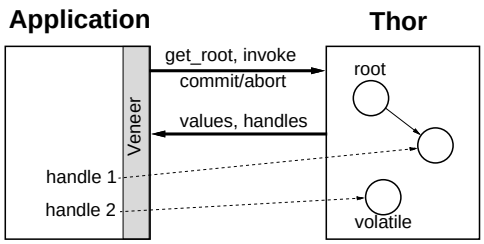

Fig. 1. The THOR Interface

Figure 1 illustrates the THOR interface. Note that THOR objects remain inside THOR; this is an important way in which THOR differs from other object-oriented databases [LLOW91, C+]. Furthermore, the distinction is critical to safe sharing because this way we can ensure that objects are accessed properly, even though the language used in the application may not be type safe.

Theta is a strongly-typed language and therefore once a method call starts running within THOR, any calls it makes will be type safe. However, no similar guarantee exists for calls coming into THOR from the application. Therefore all calls into THOR are type-checked, which is relatively expensive. To reduce the impact of this overhead, each call to THOR should accomplish quite a bit of work. This can be achieved by *code shipping*: portions of the application are pushed into THOR and run on the THOR side of the boundary. Of course such code must first be checked for type-safety; this can be accomplished by verifying the code using a bytecode or assembler verifier [MWCG98, M+99]. Once verified the code can be stored in THOR so that it can be used in the future without further checking. Information about the benefit of code shipping and other optimizations at the THOR boundary can be found in [LACZ96, BL94].

3 Implementation Architecture

The next several sections describe how we achieve good performance. Our implementation requirements were particularly challenging because we wanted a system that would perform well in a large-scale, wide-area, distributed environment. We wanted to support very large object universes, very small objects, very large numbers of sessions running concurrently, and world-wide distribution of the machines concurrently accessing THOR.

Our implementation makes use of a client/server architecture. Persistent objects are stored at servers; each object resides at a particular server, although objects can be moved from one server to another. We keep persistent objects at servers because it is important to provide continuous access to objects and this cannot be ensured when persistent storage is located at client machines (e.g., the machine's owner might turn it off). There can be many servers; in a large system there might be tens of thousands of them. Furthermore, an application might need to use objects stored at many different servers.

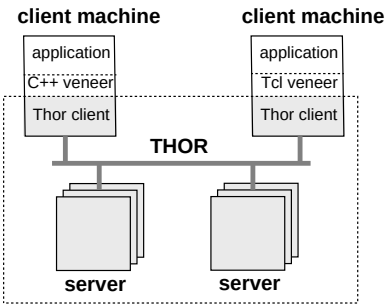

Fig. 2. Architecture of THOR Implementation

Applications run at clients on cached copies of persistent objects. This architecture is desirable because it supports scalability; it reduces the load on servers by offloading work to clients, thus allowing servers to handle more clients. Figure 2 shows the THOR architecture. The figure shows each server with a number of replicas; THOR uses replication to provide highly-available access to persistent objects.

3.1 Object Format

Servers store objects on disk in pages. To simplify cache management, objects are required not to span page boundaries. Pages are large enough that this does not cause significant internal fragmentation; for example, the average size of objects accessed by most traversals of the OO7 benchmark [CDN93] in THOR is 29 bytes, while our current page size is 8 KB. Objects larger than a page are represented using a tree.

Our design for the format of objects had the goal of keeping objects small; this 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.

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 in Section 5.1). 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, e.g., when doing garbage collection. 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-bit [CLFL94, 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).

3.2 Implementation Overview

Good performance for a distributed object storage system requires good solutions for client cache management, storage management at servers, and concurrency control for transactions. Furthermore, all of our techniques have to work properly when there are crashes; in particular there must not be any loss of persistent information, or any incorrect processing of transactions when there are failures. Our solutions in these areas are described in subsequent sections. Here we discuss our overall strategy.

When an application requests an object that is not in the client cache, the client fetches that object's page from the server. Fetching an entire page is a good idea because there is likely to be some locality within a page and therefore other objects on the page are likely to be useful to the client; also page fetches are cheap to process at both clients and servers.

However, an application's use of objects is not completely matched to the way objects are clustered on disk and therefore not all objects on a page are equally useful within an

application session. Therefore our caching strategy does not retain entire pages; instead it discards unuseful objects but retains useful ones. This makes the effective size of the client cache much larger and allows us to reduce the number of fetches due to capacity misses. Our cache management strategy, hybrid adaptive caching, or HAC, is discussed in Section 5.

To avoid communication between clients and servers, we use an optimistic concurrency control scheme. This scheme is called *Clock-based Lazy Optimistic Concurrency Control* or CLOCC. The client performs the application transaction and tracks its usage of objects without doing any concurrency control operations; at the commit point, it communicates with the servers, and the servers decide whether a commit is possible. This approach reduces communication between clients and servers to just fetches due to cache misses, and transaction commits. Our concurrency control scheme is discussed in Section 4.

When a transaction commits, we send the new versions of objects it modified to the servers. We cannot send the pages containing those objects, since the client cache does not necessarily contain them, and furthermore, sending entire pages would be expensive since it would result in larger commit messages. Therefore we do *object shipping* at commit time. The server ultimately needs to store these objects back in their containing pages, in order to preserve spatial locality; however, if it accomplished this by immediately reading the containing pages from disk, performance would be poor [OS94]. Therefore we have developed a unique way of managing storage, using a modified object buffer, or MOB, that allows us to defer writing back to disk until a convenient time. The MOB is discussed in Section 6.

In addition to CLOCC, HAC, and the MOB, THOR also provides efficient garbage collection, and it provides support for high availability via replication. Our garbage collection approach partitions the heap into regions that are small enough to remain in main memory while being collected; in addition we record the information about inter-partition references in a way that avoids disk access or allows it to be done in the background. More information about our techniques, and also about how we do distributed collection, can be found in [ML94, ML97b, ML97a, ML97c]. Our replication algorithm is based on that used in the Harp file system [LGG+91, Par98].

4 Concurrency Control

Our approach to concurrency control is very fine-grained; concurrency control is done at the level of objects to avoid false conflicts [CFZ94]. In addition our approach maximizes the benefit of the client cache by avoiding communication between clients and servers for concurrency control purposes.

We avoid communication by using optimistic concurrency control. The client machine runs an application transaction assuming that reads and writes of objects in the cache are permitted. When the transaction attempts to commit, the client informs a server about the reads and writes done by that transaction, together with the new values of any modified objects. The server determines whether the commit can occur by seeing whether it can serialize the transaction relative to other transactions; if the transaction used objects at multiple servers, a two-phase commit protocol will be used[GR93]. If

the commit can occur, the modifications are made persistent; otherwise the client is told to abort the transaction.

When a commit happens, if copies of objects modified by this transaction exist in other client caches, those copies will become stale. Transactions running at those other client machines will be unable to commit if they use the stale objects. Therefore, servers track such modifications and send *invalidations* to the affected clients. An invalidation identifies objects that have been modified since they were sent to the client; the client discards these objects from its cache (if they are still there) and aborts its current transaction if it used them. Invalidations are piggybacked on other messages sent to the client, so that extra communication is not needed for them.

Now we describe in more detail how our concurrency control scheme works. Even more detail can be found in [Gru97, Ady94, AGLM95].

4.1 Clients

As an application transaction runs, the client keeps track of the objects it reads and writes in the ROS (read object set) and MOS (modified object set), respectively; the MOS is always a subset of the ROS (modified objects are entered in both sets). We also store a copy of the current (pre-transaction) state of each modified object in an *undo log*. This tracking is done as part of running methods on the object. The ROS, MOS, and undo log are cleared at the start of each transaction.

If the application requests a commit, the MOS, ROS, and copies of all modified objects are sent to one of the servers that stores objects used by that transaction. The server will either accept or reject the commit. If the commit is rejected, or if the application requests an abort, the client uses the undo log to restore the pre-transaction states of modified objects. The undo log is an optimization; it may be discarded by the cache replacement algorithm. If a transaction aborts after its undo log is discarded, the client invalidates the cached copies of the objects modified by the transaction.

The client also processes invalidation messages. It discards the invalid objects if they are present in the cache, and then uses the MOS and ROS to determine whether the current transaction used any invalid objects. In this case, the transaction is aborted but the states of modified invalidated objects are not restored.

The client notifies servers when it has processed invalidations and when pages are discarded from its cache. This information is sent in the background, piggybacked on other messages (e.g., fetches) that the client is sending to the server.

4.2 Servers

When a server receives a commit request, it assign the transaction a timestamp. This timestamp is obtained by reading the time of the server's clock and concatenating it with the server's id to obtain a unique number. We assume that server clocks are loosely synchronized to within a few tens of milliseconds of one another. This assumption is not needed for correctness, but improves performance since it allows us to make time-dependent decisions, e.g., we are able to discard old information. The assumption about loosely-synchronized clocks is a reasonable one for current systems [Mil92].

The server then acts as *coordinator* of a two-phase commit protocol; all servers where objects used by the transaction reside act as *participants*. The coordinator sends prepare messages to the participants, which *validate* the transaction by checking locally whether the commit is acceptable; participants then send an acceptance or refusal to the coordinator. If all participants accept, the transaction commits and otherwise it aborts; in either case the coordinator notifies the client about the decision. Then the coordinator carries out a second phase to inform the participants about the decision. (If only one server is involved, we avoid the two-phase protocol entirely, and read-only participants never participate in phase two.)

With this protocol, the client learns of the commit/abort decision after four message delays. In fact we run an optimized protocol in which the client selects the transaction's timestamp and communicates with all participants directly; this reduces the delay to three messages for read/write transactions. For read-only transactions, participants send their decision directly to the client, so that there are just two message delays. Furthermore, we have developed a way to commit read-only transactions entirely at the client almost all the time [Ady99].

Now we discuss how validation works. We use backward validation[Hae84]: the committing transaction is compared with other committed and committing transactions but not with active transactions (since that would require additional communication between clients and servers).

A transaction's timestamp determines its position in the serial order and therefore the system must check whether it can commit the transaction in that position. For it to commit in that position the following conditions must be true:

1. for each object it used (read or modified), it must have used the latest version, i.e., the modification installed by the latest committed transaction that modified that object and that is before it in the serialization order.
2. it must not have modified any object used by a committing or committed transaction that follows it in the serialization order. This is necessary since otherwise we cannot guarantee condition (1) for that later transaction.

A server validates a transaction using a *validation queue*, or VQ, and *invalid sets*. The VQ stores the MOS and ROS for committed and committing transactions. For now we assume that the VQ grows without bound; we discuss how its entries are removed below. If a transaction passes validation, it is entered in the VQ as a committing transaction; if it aborts later it is removed from the VQ while if it commits, its entry in the VQ is marked as committed.

The invalid set lists pending invalidations for a client. As soon as a server knows about a commit of a transaction, it determines what invalidations it needs to send to what clients. It makes this determination using a *directory*, which maps each client to a list of pages that have been sent to it. When a transaction commits, the server adds a modified object to the invalid set for each client that has been sent that object's page; then it marks the VQ entry for the transaction as committed. As mentioned, information about invalid sets is piggybacked on messages sent to clients. An object is removed from a client's invalid set when the server receives an ack for the invalidation from the client. A page is removed from the page list for the client when the server is informed by the client that it has discarded the page.

A participant validates a transaction as follows. First, it checks whether any objects used by a transaction T are in the invalid set for T's client; if so T must abort because it has used a stale version of some object. Otherwise, the participant does the following VQ checks:

1. For each uncommitted transaction S with an earlier timestamp than T, if S's MOS intersects T's ROS, T must abort. We abort T only if S is uncommitted since otherwise T could be aborted unnecessarily (recall that the check against the invalid set ensures that T read the last versions produced by committed transactions). However, S might commit and if it did we would not be able to commit T since it missed a modification made by S. Of course, S might abort instead, but since this is an unlikely event, we simply abort T immediately rather than waiting to see what happens to S.
2. For each transaction S with later timestamp than T, T must abort if:
   (a) T's MOS intersects S's ROS. T cannot commit because if it did, a later transaction S would have missed its modifications. Again, we abort T even if S has not yet committed because S is highly likely to commit.
   (b) T's ROS intersects S's MOS. If S is committed, the abort is necessary since in this case T has read an update made by a later transaction (we know this since T passed the invalid-set test). If S is uncommitted, we could allow T to commit; however, to ensure that external consistency [Gif83] is also provided, we abort T in this case as well.

   If validation fails because of test 2b (when S is committed) or test 2a, it is possible that T could commit if it had a later timestamp. Therefore, we retry the commit of T with a later timestamp.

We use time to keep the VQ small. We maintain a *threshold timestamp*, VQ.t, and the VQ does not contain any entries for committed transactions whose timestamp is less than VQ.t. An attempt to validate a transaction whose timestamp is less than VQ.t will fail, but such a situation is unlikely because of our use of synchronized clocks. We keep VQ.t below the current time minus some delta that is large enough to make it highly likely that prepare messages for transactions for which this server is a participant will arrive when their timestamp is greater than the threshold. For example, a threshold delta of five to ten minutes would be satisfactory even for a widely-distributed network.

When transactions prepare and commit we write the usual information (the ROS, MOS, and new versions of modified objects) to the transaction log. This is necessary to ensure that effects of committed transactions survive failures. The log can be used during recovery to restore the VQ; communication with clients is necessary to restore the directories, and the invalid sets can then be recovered using information in the log. Recovery of the invalid sets is conservative and might lead to some unnecessary aborts, but failures are rare so that this possibility is not a practical problem.

4.3 Discussion

Our optimistic scheme is efficient in terms of space and processing time because our data structures are small (the VQ and the invalid sets). Directories could be large, but in

this case we can simply keep coarser information; the cost would be larger invalid sets, and more invalidations piggybacked on messages sent to the client, but no extra aborts would result.

We have shown that our scheme performs well in practice by comparing its performance to that of other schemes; the results are reported in [AGLM95, Gru97]. In particular we compared its performance to that of adaptive callback locking [CFZ94, ZCF97], which is considered to be the strongest competitor. Our results show that our approach outperforms adaptive callback locking in all reasonable environments and almost all workloads. We do better both on low contention workloads, which are likely to be the common case, and also on high contention workloads. These experiments were performed under simulation [AGLM95, Gru97]; they allowed us to show that our results scale to systems with lots of clients and servers.

When there is high contention, our optimistic scheme has more aborts while a locking scheme has more delay. Our scheme performs well because the cost of aborts is low: we abort early (e.g., when an invalidation arrives at the client), and when we rerun the transaction we are able to run quickly because much of what is needed is already in the client cache, including the previous states of modified objects. Furthermore, most of the extra work due to aborts occurs at the clients, rather than at the servers, which are the scarce resource. Locking schemes have a larger impact on the servers, and the delays due to lock contention can be longer than our delays due to rerunning transactions after aborts.

Our scheme assumes that a transaction's modifications fit in the client cache; we believe this is reasonable for today's machines given the very efficient way we manage the cache (see Sections 5 and 7). Our results apply only to client/server systems in which transactions run at the clients; if transactions ran at servers, the extra work due to aborts would slow down all clients rather than just the client whose transaction aborted, so that optimism may not be the best approach.

5 Hybrid Adaptive Caching

Most persistent object systems manage the client cache using *page caching* [LLOW91, WD94, SKW92]; such systems fetch and discard entire pages. These systems have low miss penalties because it is simple to fetch and replace fixed-size units. Also, page caching can 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 [TN91, CS89, Day95]. Furthermore, access patterns may evolve over time, and reclustering will lag behind because effective clustering algorithms are very expensive [TN91] 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+, TG90, WD92, K+89] allow clients to cache hot objects without caching their containing disk pages and can thus 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.

Our cache management scheme is called HAC, for hybrid adaptive caching. HAC is a hybrid between page and object caching that combines the virtues of each — low overheads and low miss rates — while avoiding their problems. It partitions the cache between objects and pages adaptively based on the current application behavior: pages in which locality is high remain intact, while only hot objects are retained for pages in which locality is poor. Hybrid object and page caching was introduced in [OS95, Kos95] but this earlier work did not provide solutions to the crucial problems of cache partitioning and storage fragmentation.

HAC partitions the client cache into page-sized *frames* and fetches entire pages from the server. To make room for an incoming page, it

- 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 3.

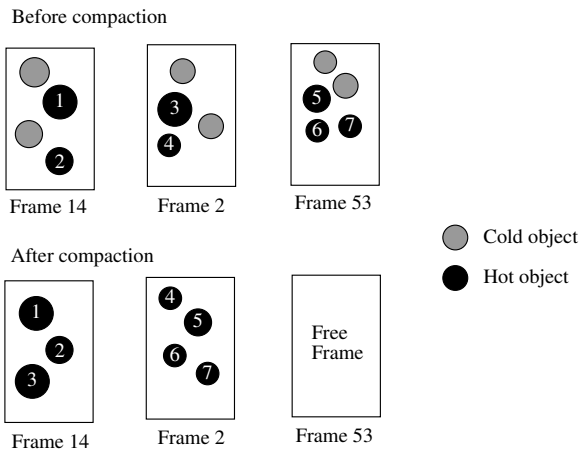

Fig. 3. Compaction of pages by HAC

Now we explain the scheme in more detail; even more information can be found in [CALM97].

5.1 Client Cache Organization

Pages at the client have the same size and structure as at the server to avoid extra copies. However, it is not practical to represent object pointers as *orefs* in the client cache because each pointer dereference would require an expensive computation to obtain the object's

memory location. Therefore, clients perform *pointer swizzling* [TG90, 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 [TG90]; the *oref* is translated to a pointer to an entry in an *indirection table* and the entry points to the target object. (In-cache pointers are 32 bits just like orefs. On 64-bit machines; HAC simply ensures that the cache and the indirection table are located in the lower 2^{32} bytes of the address space.) 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].

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.

HAC uses a novel lazy reference counting mechanism to discard entries from the indirection table [CAL97]. 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.

5.2 Compaction

HAC computes usage information for both objects and frames as described in Section 5.3, and uses this information to select a victim frame V to compact, and also to identify which of $V's$ objects to retain and which to discard. Then it moves retained objects from V into frame T, the current *target* for retained objects, laying them out contiguously to avoid fragmentation. Indirection table entries for retained objects are corrected to point to their new locations; and entries for discarded objects are modified to indicate that the objects are no longer present in the cache. If all retained objects fit in T, the compaction process ends and V can be used to receive the next fetched page. If some retained objects do not fit in T, V becomes the new target and the remaining objects are compacted inside V to make all the available free space contiguous. Then, another frame is selected for compaction and the process is repeated for that frame.

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.

When disk page P is fetched, some object o in P may already be in use, cached in frame F. HAC handles this 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.

5.3 Replacement

Now we discuss how we do page replacement. We track object usage, use this to compute frame usage information from time to time, select frames for removal based on the frame usage information, and retain or discard objects within the selected frame based on how their usage compares to that of their frame. Replacement is 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.

Object Usage Computation Our object usage calculation takes both recency and frequency of access into account, since this has been shown to outperform LRU [JS94, OOW93, RD90], but we do this with very low overhead. Headers of installed objects contain 4 usage bits. The most significant usage bit is set each time a method is invoked on the object. Usage bits are cheap in both space and time; only two extra instructions and no extra processor cache misses are needed to do a method call. They are much cheaper in both space and time than maintaining either an LRU chain or the data structures used in [JS94, OOW93, RD90].

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. We set the usage of invalid objects to 0, which ensures their timely removal from the cache.

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, which means we would need to compact all these frames. Therefore, we compute usage values for frames and use these values to select frames to compact.

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; we have found

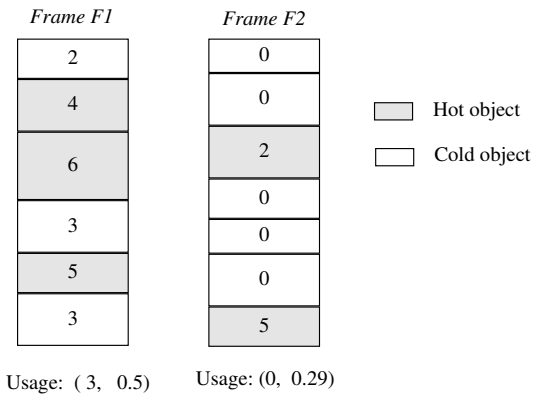

Fig. 4. Usage statistics for frames

experimentally that $R = 2/3$ works well. 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 4. 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; it is a reasonable estimate if the average object size is much smaller than a page.

HAC uses a *no-steal* [GR93] cache management policy: objects that were modified by a transaction cannot be evicted from the cache until the transaction commits. (Objects read by the current transactions, and old versions in the undo log, can be discarded.) 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.

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

F.T $<$ G.T or (F.T = G.T and F.H $<$ 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 4, $F2$ 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*. Frames are added to this set at each fetch; we select frames to add using a variant of the clock algorithm [Cor69]. 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. We remove frames from the

candidate set if they survive enough fetches; we have found experimentally that removing frames after 20 fetches works well.

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.

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.

6 The Modified Object Buffer

Our use of HAC, and also the fact that we discard invalid objects from the client cache, mean that it is impossible for us to send complete pages to the server when a transaction commits. Instead we need to use object shipping. Ultimately, however, the server must write objects back to their containing page in order to preserve clustering. Before writing the objects it is usually necessary to read the containing page from disk, since it is unlikely in a system like ours that the containing page will be in memory when it is needed [MH92]. Such a read is called an *installation read* [OS94]. Doing installation reads while committing a transaction performs poorly [WD95, OS94] so that some other approach is needed.

Our approach is to use a volatile buffer in which we store recently modified objects; the buffer is called the MOB (for modified object buffer). When modified objects arrive at the server they are stored in the MOB instead of being installed in a page cache. The modifications are written to disk lazily as the MOB fills up and space is required for new modifications. Only at this point are installation reads needed.

The MOB architecture has several advantages over a conventional page buffer. First, it can be used in conjunction with object shipping, and yet installation reads can be avoided when transactions commit. Second, less storage is needed to record modifications in the MOB than to record entire modified pages in a page cache. Therefore the MOB can record the effects of more transactions than a page cache, given the same amount of memory; as a result, information about modifications can stay in the MOB longer than in a page cache. This means that by the time we finally move an object from the MOB to disk, there is a high probability that other modifications have accumulated for its page than in the case of a page cache. We call this effect *write absorption*. Write absorption leads to fewer disk accesses, which can improve system performance.

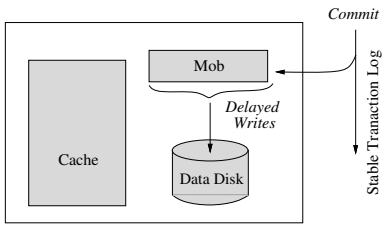

Fig. 5. Server Organization

Now we describe how the MOB works; more information can be found in [Ghe95]. The server contains some volatile memory, disk storage, and a stable transaction log as shown in Figure 5. The disk provides persistent storage for objects; the log records modifications of recently committed transactions. The volatile memory is partitioned into a page cache and a MOB. The page cache holds pages that have been recently fetched by clients; the MOB holds recently modified objects that have not yet been written back into their pages on disk.

Modifications of committed transactions are inserted into the MOB as soon as the commit has been recorded in the log. They are not written to disk immediately. Instead a background *flusher* thread lazily moves modified objects from the MOB to disk. The flusher runs in the background and does not delay commits unless the MOB fills up completely with pending modifications.

The MOB does not replace the transaction log, which is used to ensure that transactions commit properly in spite of failures. After a failure the MOB can be restored from the log. When objects are removed from the MOB, the corresponding records can also be removed from the log.

Pages in the cache are completely up-to-date: they contain the most current versions of their objects. However, pages on disk do not reflect modifications of recent transactions, i.e., modifications in the MOB. Therefore a fetch request is processed as follows: If the needed page is not in the page cache, it is read into the cache and then updated to reflect all the recent modifications of its objects in the MOB. Then the page is sent to the client. When the page is evicted from the cache, it is not written back to disk even if it contains recent modifications; instead we rely on the flusher to move these

modifications to disk.

The page cache is managed using a LRU policy, but the MOB is managed in a FIF0 order to allow the log to be truncated in a straightforward way. The flusher scans the MOB and identifies a set of pages that should be written to disk to allow a prefix of the log to be truncated. This set of pages is read into the page cache if necessary (these are the installation reads). Then *all* modifications for those pages are installed into the cached copies and removed from the MOB. Then the pages are written to disk, and finally a prefix of the log is truncated. Therefore the operation of the flusher replaces the checkpointing process used for log truncation in most database systems [GR93]. The log may contain records for modifications that have already been installed on disk but this is not a problem: the modifications will be re-installed if there is a failure (and re-entered in the MOB), but failures are rare so that the extra cost to redo the installs is not an issue and correctness is not compromised since installation is idempotent.

The MOB is implemented in a way that makes both fetches and the flusher run fast. The flusher needs to identify pages to be flushed; it also needs to find all modifications for these pages, and remove them from the MOB without leaving any holes. Fetching requires a quick way to find any modifications for the requested page. We accommodate these requirements by storing the modified objects in a heap that is managed using a standard malloc/free memory allocator. The contents of the heap are chained together in a *scan list* that can be used to identify the pages that need to be written out so that the MOB contents can be discarded in log order. Finally, a hash table maps from page identifiers to the set of objects in the MOB that belong to that page. This hash table is used to handle client fetch requests, and also by the flusher to find the set of all objects belonging to a page that is being written to disk.

Writes to disk are actually done using units called segments, which are larger than pages and correspond roughly in size to a disk sector. Using segments instead of pages allows us to take advantage of disk characteristics. Furthermore, segments are useful from an application perspective since they provide a larger unit for clustering, and when such clustering exists there can be more write absorption using segments than pages. Thus segments improve performance in two ways: by using the disk more efficiently and by improving write absorption.

7 Performance Evaluation

This section evaluates the performance of THOR. It compares THOR with a system, called C++/OS, that does not implement safe sharing or transactions and is directly based on abstractions and mechanisms offered by current operating systems. This system does not provide the full functionality of THOR. It provides a form of transparent persistency for objects but it does not support transactions or safe sharing.

The comparison between the two systems is interesting because C++/OS indicates the kind of performance that could be obtained by running a persistent object store directly on current operating systems; also, C++/OS is believed to perform extremely well because the mechanisms it uses have been the focus of extensive research and development and are fairly well optimized. Our results show that THOR significantly outperforms C++/OS in the common case when there are cold or capacity misses in the

client cache, or when objects are modified. Furthermore, THOR outperforms C++/OS even though C++/OS implements a much less powerful programming model.

7.1 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. 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. All our experiments ran on the *medium* database, which has 200 atomic parts per composite part. The traversals we ran and the environment are described below.

C++/OS The C++/OS system uses a C++ implementation of OO7. This implementation uses a modified memory allocator that creates objects in a memory-mapped file in the client machine. This file is stored by a server and it is accessed by the client operating system using the NFS Version 3 distributed file system protocol.

The operating system manages the client cache by performing replacement at the page granularity (8 KB) and it also ships modified pages back to the server. We added an `msync` call at the end of each traversal to force any buffered modifications back to the server.

C++/OS uses 64-bit object pointers so that more than 4 GB of data can be accessed; this is the same approach taken in [CLFL94]. The result is a database that is 42% larger than THOR's, but the system does not incur any space or time overhead for swizzling pointers.

Environment The objects in the databases in both systems are clustered into 8 KB. In THOR, the database was stored by a replicated server with two replicas and a witness [LGG+91], i.e., a server that can tolerate one fault. Each replica stored the database 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 THOR database takes up 38 MB in our implementation. The C++/OS database is also stored on a Seagate ST-32171N disk; it takes up 54 MB since it uses 64-bit pointers.

The database was accessed by a single client. Both the server and client machines were DEC 3000/400 Alpha workstations, each with a 133 MHz Alpha EV4 (21064) processor, 160 MB of memory and Digital Unix. They were connected by a 10 Mb/s Ethernet and had DEC LANCE Ethernet interfaces. In THOR, each server replica had a 36 MB cache (of which 6 MB were used for the MOB); in C++/OS, the NFS server had 137 MB of cache space. We experimented with various sizes for the client cache.

All the results we report are for *hot* traversals: we preload the caches by running a traversal twice and timing the second run. There are no cold-cache misses in either the server or the client cache during a hot traversal. This is conservative; THOR outperforms C++/OS in cold cache traversals because its database is much smaller.

The C code generated by the Theta compiler for the traversals, the THOR system code, and the C++ implementation of OO7 were all compiled using GNU's gcc with optimization level 2.

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.

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 [TN91] found it was possible to achieve good average use only 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 [CS89] 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 [TN91].

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 only present results for traversals T1, T2a and T2b. Since the performance of THOR relative to a page-based system improves when the clustering does not match the traversal, this underestimates the performance gain afforded by our techniques.

7.2 Read-Only Traversals

This section evaluates the performance of THOR running a hot read-only traversal, T1. It starts by presenting a detailed analysis of the overhead when all the pages accessed by T1 fit in the client cache and there are no cold cache misses. Then, it analyzes the performance for the common case when not all the pages fit in the client cache.

Traversals without cache management Our design includes choices (such as indirection) that penalize performance when all the pages accessed fit in the client cache to improve performance in the common case when they do not; this section shows that the price we pay for these choices is reasonable.

We compare THOR with C++/OS running a T1 traversal of the database. Both systems run with a 55 MB client cache to ensure that there are no client cache misses.

Table 1 shows where the time is spent in THOR. 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 in the direct-mapped processor caches, we used *cord* and *ftoc*, two Digital Unix utilities that reorder procedures in an executable to reduce conflict misses. We used cord on all THOR executables and on the C++/OS executable.

	T1 (sec)
Exception code	0.86
Concurrency control checks	0.64
Usage statistics	0.53
Residency checks	0.54
Swizzling checks	0.33
Indirection	0.75
C++/OS traversal	4.12
Total (THOR traversal)	7.77

Table 1. Breakdown, Hot T1 Traversal, Without Cache management

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. As we discussed before these features are very important for safe sharing of persistent objects. Since the C++/OS system does not offer these features, it incurs no overhead for them.

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 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++/OS traversals from the elapsed times obtained with a THOR front-end executable from which the code corresponding to all the other lines in the table had been removed.

Note that 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.

Figure 6 presents elapsed time results for the hot T1 traversals without any cache

misses. The results show that the overheads introduced by THOR on hit time are quite reasonable; THOR adds an overhead relative to C++/OS of 89% on T1. Furthermore, our results show that this overhead is quickly offset by the benefits of our implementation techniques in the presence of client cache misses or modifications .

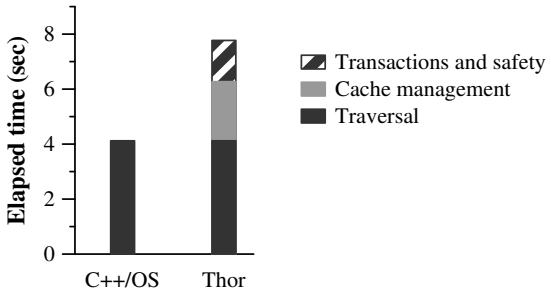

Fig. 6. Elapsed time, Hot T1 traversal, Without Cache Management

Traversals with cache management The previous experiments compared the performance of THOR and C++/OS when the client cache can hold all the pages touched by the traversal. Now we analyze the performance for smaller cache sizes and show that THOR performs better than C++/OS in this region.

Figure 7 shows elapsed times for the hot T1 traversals with cache management. We measured running a hot T1 traversal on both systems while varying the amount of memory devoted to caching at the client. For THOR, this includes the memory used by the indirection table. The amount of memory available to the client in the C++/OS system was restricted by using a separate process that locked pages in physical memory. The elapsed times in THOR do not include the time to commit the transaction. This time was approximately 1.8 seconds.

The performance gains of THOR relative to C++/OS are substantial because it has a much lower miss rate. For example, for a cache size of 18 MB, THOR has 4132 client cache misses whereas C++/OS has 16945. The maximum performance difference between THOR and C++/OS occurs for the minimum cache size at which all the objects used by the traversal fit in the client cache; THOR performs approximately 15 times faster than C++/OS in traversal T1. From another perspective, THOR requires less than half the memory as C++/OS to run traversal T1 without cache management activity.

The miss rate is lower because of HAC and because our objects are smaller. HAC allows the client to cache only the objects accessed by the traversal rather than their pages. Since on average T1 uses only 49% of each page, HAC improves performance significantly. The impact is even bigger in transactions that match the database clustering poorly, e.g., HAC improves performance by up to three orders of magnitude in T6 [CALM97].

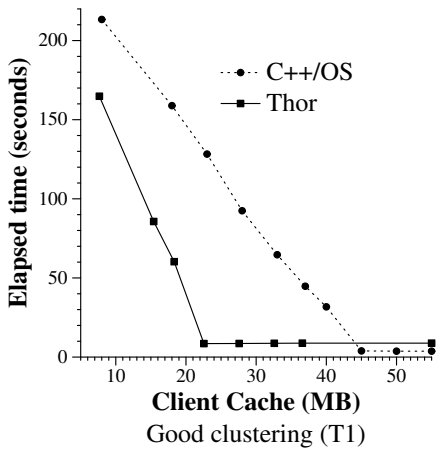

Fig. 7. Elapsed Time, Hot T1 Traversal, With Cache Management

THOR's swizzling technique and the use of surrogates allow it to use 32-bit pointers, but at the cost of extra computation and extra space to store the indirection table. As discussed previously, the C++/OS system avoids these costs but needs to use 64-bit pointers instead. The result is a space overhead for the database that is two times larger than the space used up by our indirection table during traversal T1.

C++/OS does not incur any disk reads in the experiments reported in this section whereas THOR does; the time to read from disk accounts for approximately 20% of THOR's miss penalty. This happens because we conservatively allowed the NFS server in C++/OS to use a 137 MB cache while the THOR server used a 36 MB cache (from which 6 MB were dedicated to the MOB). THOR's performance relative to C++/OS would have been even better if we had not been conservative. More generally, as discussed in [Bla93], we expect the miss rate in the server cache to be high in real applications. This will lead to high miss penalties and will further increase our performance gains.

7.3 Read-Write Traversals

All experiments presented so far ran traversal T1, which is read-only. This section shows that THOR outperforms C++/OS for traversals with updates even when all the pages accessed fit in the cache. Figure 9 presents elapsed times for hot traversals T2a and T2b running with a 55 MB client cache. For this cache size, there is no cache replacement in either of the two systems.

The results show that THOR outperforms C++/OS. The time to run the traversal is lower in the C++/OS system mainly for the causes discussed in Section 7.2. But the time to commit the modifications to the server is much lower in THOR for two reasons. First, THOR ships only the modified objects back to the server whereas C++/OS ships the modified pages. For example, in T2b, THOR ships back 4.5 MB whereas C++/OS ships back 25.7 MB. Second, C++/OS installs all the modified pages on disk at the server;

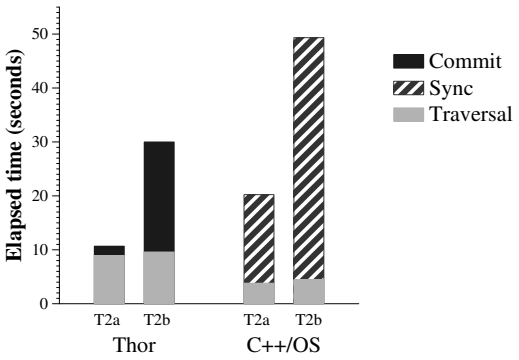

Fig. 8. Elapsed time, Hot Read-Write traversals, Without Cache Management

the THOR server only inserts the modified objects in the MOB and in the stable log (i.e., forces the modified objects to the backup replica). This performance gain can be attributed to the MOB, which enables efficient object shipping by allowing installation reads to be performed in the background as discussed in Section 6.

The small 6 MB MOB used in these experiments is more than sufficient to absorb all the modifications in the OO7 traversals. Therefore, there are no installation reads in these experiments. The detailed study of the MOB architecture in [Ghe95] analyzes workloads with installation reads and shows that this architecture outperforms a page shipping system for almost all workloads. The only exception are the rare workloads that modify almost all the objects they access.

We also ran traversals T2a and T2b in a configuration with cache management. Figure 9 presents elapsed times for a configuration with a 18 MB client cache. The results show that with cache management and modifications THOR significantly outperforms C++/OS. This happens because, in the C++/OS system, the cache replacement mechanism is forced to write modified pages back to the server to make room for missing pages while the traversal is running. This increases the number of page writes relative to what was shown in Figure 8 and these writes are to random locations. For example, C++/OS performs 12845 page writes in T2b with cache management and only 3285 writes without cache management. On the other hand, THOR keeps modified objects in the cache until the transaction commits.

As discussed previously, THOR uses a *no-steal* [GR93] cache management policy: modified objects cannot be evicted from the client cache until the current transaction commits. We claim there is no significant loss in functionality or performance in our system due to the lack of a steal approach; 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 9: HAC allows THOR to run 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). Evicting modified pages is needed in a page-caching system, since the cache is used much less efficiently.

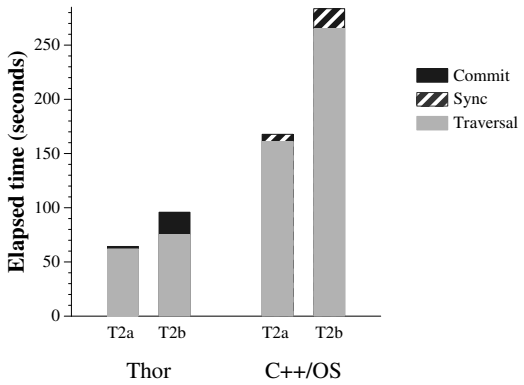

Fig. 9. Elapsed time, Hot Read-Write traversals, With Cache Management (18 MB)

8 Conclusions

THOR provides a powerful programming model that allows applications to safely share objects across both space and time. It ensures that sharing is safe by allowing objects to be accessed only by calling their methods. In addition, it provides atomic transactions to guarantee that concurrency and failures are handled properly.

The paper describes three implementation techniques that enable THOR to support this powerful model while providing good performance: the CLOCC concurrency control scheme; the HAC client caching scheme; and the MOB disk management architecture at servers. It also presents the results of experiments that compare the performance of THOR to that of C++/OS, which provides persistency using memory mapped files but does not support safe sharing or transactions. The performance comparison between THOR and C++/OS is interesting because the latter approach is believed to deliver very high performance. However, our results show that THOR substantially outperforms C++/OS in the common cases: when there are misses in the client cache, or when objects are modified.

The high performance of THOR is due to all three of its implementation techniques. The MOB provides good performance in modification workloads because it handles small writes efficiently and reduces communication overhead by allowing object shipping at transaction commits. HAC has low overhead and takes advantage of object shipping to reduce capacity misses by managing the client cache at an object granularity. And, although CLOCC does not show up to full advantage in these experiments (since there is no concurrency), our performance benefits from its low overhead and the fact that it avoids introducing extra communication between clients and servers.

We believe these results have important implications for future architectures for supporting persistence. As object systems become the accepted base for building modern distributed applications, the semantic gap between file systems and applications will keep widening. This gap translates into loss of safety and, as indicated by our results, a loss of performance. The paper shows that there is a very attractive alternative: a persistent object storage system that provides both a powerful semantic model and high

performance.

Acknowledgements

THOR was the result of the work of many researchers in the Programming Methodology Group at MIT. The following people were instrumental in designing and implementing various parts of THOR: Phillip Bogle, Chandrasekhar Boyapati, Dorothy Curtis, Mark Day, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Umesh Maheshwari, Andrew Myers, Tony Ng, Arvind Parthasarathi, Quinton Zondervan.

References

[Ady94] A. Adya. Transaction Management for Mobile Objects Using Optimistic Concurrency Control. Master's thesis, Massachusetts Institute of Technology, Jan. 1994. Also available as MIT Laboratory for Computer Science Technical Report MIT/LCS/TR-626.

[Ady99] A. Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, Mar. 1999.

[AGLM95] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari. Efficient Optimistic Concurrency Control using Loosely Synchronized Clocks. In *Proc. of ACM SIGMOD International Conference 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, Laboratory for Computer Science, MIT, Cambridge, MA, May 1977.

[BL94] P. Bogle and B. Liskov. Reducing Cross-Domain Call Overhead Using Batched Futures. In *Proc. OOPSLA '94*, pages 341–359. ACM Press, 1994.

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

[Boy98] C. Boyapati. JPS: A Distributed Persistent Java System. Master's thesis, Massachusetts Institute of Technology, Sept. 1998.

[C$^+$] M. J. Carey et al. Shoring Up Persistent Applications. In *Proc. of ACM SIGMOD International Conference on Management of Data*, pages 383–394, Minneapolis, MN, May 1994. ACM Press.

[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.

[CALM97] M. Castro, A. Adya, B. Liskov, and A. Myers. HAC: Hybrid Adaptive Caching for Distributed Storage Systems. In *Proc. 17th ACM Symp. on Operating System Principles (SOSP)*, pages 102–115, St. Malo, France, Oct. 1997.

[CDN93] M. J. Carey, D. J. DeWitt, and J. F. Naughton. The OO7 Benchmark. In *Proc. of ACM SIGMOD International Conference on Management of Data*, pages 12–21, Washington D.C., May 1993.

[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. At `ftp://ftp.cs.wisc.edu/OO7`.

[CFZ94] M. Carey, M. Franklin, and M. Zaharioudakis. Fine-Grained Sharing in a Page Server OODBMS. In *Proc. of ACM SIGMOD International Conference on Management of Data*, pages 359–370, Minneapolis, MN, June 1994.

[CLFL94] J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska. Sharing and Protection in a Single-Address-Space Operating System. In *ACM Transactions on Computer Systems*, volume 12, Feb. 1994.

[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.

[CS89] W. W. Chang and H. J. Schek. A Signature Access Method for the Starburst Database System. In *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, pages 145–153, Amsterdam, Netherlands, August 1989.

[D+90] O. Deux et al. The Story of O_2. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):91–108, March 1990.

[Day95] M. Day. *Client Cache Management in a Distributed Object Database*. PhD thesis, Massachusetts Institute of Technology, 1995. Also available as MIT Laboratory for Computer Science Technical Report MIT/LCS/TR-652.

[DGLM95] M. Day, R. Gruber, B. Liskov, and A. C. Myers. Subtypes vs. Where Clauses: Constraining Parametric Polymorphism. In *Proc. OOPSLA '95*, pages 156–168, Austin TX, Oct. 1995. ACM SIGPLAN Notices 30(10).

[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*, Mar. 1994.

[Ghe95] S. Ghemawat. *The Modified Object Buffer: a Storage Management Technique for Object-Oriented Databases*. PhD thesis, Massachusetts Institute of Technology, 1995. Also available as MIT Laboratory for Computer Science Technical Report MIT/LCS/TR-656.

[Gif83] D. Gifford. Information Storage in a Decentralized Computer System. Technical Report CSL-81-8, Xerox Corporation, March 1983.

[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. N. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., 1993.

[Gru97] R. Gruber. *Optimism vs. Locking: A Study of Concurrency Control for Client-Server Object-Oriented Databases*. PhD thesis, M.I.T., Cambridge, MA, 1997.

[Hae84] T. Haerder. Observations on Optimistic Concurrency Control Schemes. *Information Systems*, 9(2):111–120, June 1984.

[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.

[K+89] W. Kim et al. Architecture of the ORION Next-Generation Database System. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):109–124, June 1989.

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

[LAC+96] B. Liskov, A. Adya, M. Castro, M. Day, S. Ghemawat, R. Gruber, U. Maheshwari, A. Myers, and L. Shrira. Safe and Efficient Sharing of Persistent Objects in Thor. In *Proc. of ACM SIGMOD International Conference on Management of Data*, pages 318–329, Montreal, Canada, June 1996.

[LACZ96] B. Liskov, A. Adya, M. Castro, and Q. Zondervan. Type-safe Heterogenous Sharing Can Be Fast. In *Proceedings of the 7th International Workshop on Persistent Object Systems*, Cape May, NJ, May 1996.

[LCD+94] B. Liskov, D. Curtis, M. Day, S. Ghemawat, R. Gruber, P. Johnson, and A. C. Myers. *Theta Reference Manual*. Programming Methodology Group Memo 88, MIT Laboratory for Computer Science, Cambridge, MA, Feb. 1994. Available at http://www.pmg.lcs.mit.edu/papers/thetaref/.

[LGG+91] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. Replication in the Harp File System. In *Proc. 13th ACM Symp. on Operating System Principles (SOSP)*, pages 226–238. ACM Press, 1991.

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

[M+99] G. Morrisett et al. TALx86: A Realistic Typed Assembly Language. Submitted for publication, 1999.

[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.

[MH92] D. Muntz and P. Honeyman. Multi-level Caching in Distributed File Systems or Your Cache ain't nothin' but trash. In *Winter Usenix Technical Conference*, 1992.

[Mil92] D. L. Mills. Network Time Protocol (Version 3) Specification, Implementation and Analysis. Network Working Report RFC 1305, March 1992.

[MK94] W. J. McIver and R. King. Self Adaptive, On-Line Reclustering of Complex Object Data. In *Proc. of ACM SIGMOD International Conference on Management of Data*, pages 407–418, Minneapolis, MN, May 1994.

[ML94] U. Maheshwari and B. Liskov. Fault-Tolerant Distributed Garbage Collection in a Client-Server Object-Oriented Database. In *Third International Conference on Parallel and Distributed Information Systems, Austin*, Sept. 1994.

[ML97a] U. Maheshwari and B. Liskov. Collecting Cyclic Distributed Garbage by Controlled Migration. *Distributed Computing*, 10(2):79–86, 1997.

[ML97b] U. Maheshwari and B. Liskov. Partitioned Collection of a Large Object Store. In *Proc. of SIGMOD International Conference on Management of Data*, pages 313–323, Tucson, Arizona, May 1997. ACM Press.

[ML97c] U. Maheswari and B. Liskov. Collecting Cyclic Distributed Garbage using Back Tracing. In *Proc. of the ACM Symposium on Principles of Distributed Computing*, Santa Barbara, California, Aug. 1997.

[Mos90] J. E. B. Moss. Design of the Mneme Persistent Object Store. *ACM Transactions on Office Information Systems*, 8(2):103–139, March 1990.

[Mos92] J. E. B. Moss. Working with Persistent Objects: To Swizzle or Not to Swizzle. *IEEE Transactions on Software Engineering*, 18(3), August 1992.

[MS95] M. McAuliffe and M. Solomon. A Trace-Based Simulation of Pointer Swizzling Techniques. In *Proc. International Conf. on Data Engineering*, Mar. 1995.

[MWCG98] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. In *Proc. 25th ACM Symp. on Principles of Programming Languages (POPL)*, San Diego, California, Jan. 1998.

[Ont92] Ontos. Inc. 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 *Proc. of ACM SIGMOD International Conference on Management of Data*, Washington, D.C., May 1993.

[OS94] J. O'Toole and L. Shrira. Opportunistic Log: Efficient Reads in a Reliable Storage Server. In *Proc. of First Usenix Symposium on Operating Systems Design and Implementation*, pages 119–128. ACM Press, 1994.

[OS95] J. O'Toole and L. Shrira. Shared Data Management Needs Adaptive Methods. In *In Proc. of IEEE Workshop on Hot Topics in Operating Systems*, May 1995.

[Par98] A. Parthasarathi. The NetLog: An Efficient, Highly Available, Stable Storage Abstraction. Master's thesis, Massachusetts Institute of Technology, June 1998.

[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'l Workshop on Persistent Object Systems*, San Miniato, Italy, Sept. 1992.

[TG90] K. T. and K. G. *LOOM—Large Object-Oriented Memory for Smalltalk-80 Systems*, pages 298–307. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1990.

[TN91] M. Tsangaris and J. Naughton. A stochastic approach for clustering in object bases. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 12–21, Denver, CO, 1991. ACM.

[WD92] S. J. White and D. 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.

[WD94] S. J. White and D. J. Dewitt. Quickstore: A high performance mapped object store. In *SIGMOD '94*, pages 187–198, 1994.

[WD95] S. J. White and D. J. Dewitt. Implementing crash recovery in QuickStore: A performance study. In *SIGMOD '95*, pages 187–198. ACM Press, 1995.

[ZCF97] M. Zaharioudakis, M. J. Carey, and M. J. Franklin. Adaptive, Fine-Grained Sharing in a Client-Server OODBMS: A Callback-Based Approach. *ACM Transactions on Database Systems*, 22(4):570–627, Dec. 1997.

This article was processed using the LaTeX macro package with LLNCS style