TimeLine: A High Performance Archive for a Distributed Object Store

Chuang-Hue Moh and Barbara Liskov
MIT Computer Science and Artificial Intelligence Laboratory

Abstract

This paper describes TimeLine, an efficient archive service for a distributed storage system. TimeLine allows users to take snapshots on demand. The archive is stored online so that it is easily accessible to users. It enables "time travel" in which a user runs a computation on an earlier system state.

Archiving is challenging when storage is distributed. In particular, a key issue is how to provide consistent snapshots, yet avoid stopping user access to stored state while a snapshot is being taken. The paper defines the properties that an archive service ought to provide and describes an implementation approach that provides the desired properties yet is also efficient. TimeLine is designed to provide snapshots for a distributed persistent object store. However the properties and the implementation approach apply to file systems and databases as well.

TimeLine has been implemented and we present the results of experiments that evaluate its performance. The experiments show that computations in the past run well when the archive store is nearby, e.g., on the same LAN, or connected by a high speed link. The results also show that taking snapshots has negligible impact on the cost of concurrently running computations, regardless of where the archived data is stored.

1 INTRODUCTION

This paper describes TimeLine, an efficient archive service for a storage system. TimeLine allows users to take snapshots of system state on demand, either by issuing a command, and/or by running a program that requests snapshots periodically, e.g., once every six hours. The archive containing the snapshots is stored online so that it is easily accessible to users.

Access to the archive is based on time. Each snapshot is assigned a timestamp, reflecting the time at which it was requested. The archive enables "time travel" in which users can run computations "in the past." The user indicates the time at which the computation should run. The snapshot used to run the computation is the latest one whose timestamp is no greater than the specified time. We chose to use a time-based interface because we believe it matches user needs. Note however that users can easily build directory structures that allow them to associate names with snapshots.

TimeLine is designed to provide access to old states of objects in the Thor persistent object store (Thor [2, 13, 14]). Thor is a distributed system in which storage resides at many servers; it is designed to scale to very large size, with a large number of servers that are geographically distributed. Thor allows users to run computations that access the current states of objects; TimeLine extends this interface so that users can also run computations that access earlier states of objects.

The key issue in providing an archive is providing consistent snapshots without disrupting user access to stored state while a snapshot is being taken. This issue arises in any kind of storage system, not just a persistent object store. Consistency without disruption is easy to provide when all storage resides at a single server but challenging when storage is distributed.

To our knowledge, TimeLine is the first archive for a distributed storage system that provides consistent snapshots without disruption. Most other archive systems (e.g., [8, 24, 31]) disrupt user access to a greater or lesser extent while a snapshot is being taken. Elephant [26] does not cause such a disruption, but it is not a distributed system. In addition, Elephant snapshots all data; by taking snapshots on command we reduce archive storage and CPU costs associated with taking snapshots.

TimeLine has been implemented and we present the results of experiments that evaluate its performance. Our experiments show that taking snapshots has negligible impact on the cost of running computations that are using the "current" system state. We also present results showing the cost of using snapshots, i.e., running computations in the past. Our experiments show that computations in the past run as well as those in the present when the archive state is co-located with the current system state. However, it is not necessarily desirable to co-locate storage like this, since the archived state can become very large. Therefore we looked at a range of options for using shared storage for snapshots. Our results show that good performance for computations in the past is possible using shared archive storage provided it is reasonably close, e.g., on the same LAN, or connected by a high speed link. The results also show that the cost of running computations in the present while archiving is occurring is insensitive to where the archived data is stored.

The remainder of the paper is organized as follows. Section 2 discusses requirements for an archive system and describes our approach for providing consistency without disruption. Sections 3 to 6 describe TimeLine and its implementation. Section 7 presents our performance results and Section 8 discusses related work. We conclude in Section 9.

2 SNAPSHOT REQUIREMENTS

This section defines requirements for an archive service and describes our approach to satisfying them.

An archive must provide global consistency [18]:

- *Consistency*. A snapshot must record a state that could have existed at some moment in time.

Typically, an application controls the order of writes by completing one modification before starting another, i.e., by ensuring that one modification *happens before* another [11]. Then, a snapshot must not record a state in which the second modification has occurred but the first has not.

However, we cannot afford to disrupt activities of other users:

- *Non-Disruption.* Snapshot should have minimal impact on system access: users should be able to access the system in parallel with the taking of the snapshot and the performance of user interactions should be close to what can be achieved in a system without snapshots.

Note that non-disruption is typically not a requirement for a backup system, since backups tend to be scheduled for times when there is little user activity. In contrast, our system allows snapshots to be taken whenever users want, and if snapshotting isn't implemented properly, the impact on concurrent users can be unacceptable.

One way to get a consistent snapshot is to freeze the system. In a distributed system, freezing requires a two-phase protocol (such a protocol is used in [31]). In phase 1 each node is notified that a snapshot is underway. When a node receives this message, it suspends processing of user requests (and also notes that modifications that have happened up to that point will be in the snapshot but later ones will not). The coordinator of the protocol waits until it receives acks from all nodes. Then in phase 2 it notifies all nodes that the freeze can be lifted. Freezing provides consistency because it ensures that if a modification to x is not reflected in a snapshot, this modification will be delayed until phase 2, and therefore any modification that happened after it will also not be included in the snapshot. Freezing is necessary since otherwise x's node would not delay the modification to x and therefore the snapshot could observe some modification that happened after the modification of x, yet miss the modification of x.

Doing snapshots by freezing the system obviously does not satisfy our non-disruption goal. The approach is particularly problematical in a large-scale distributed system: when there are many nodes that may be widely distributed, the time required to run the protocol can be large. Also, if any storage node is down or not communicating, the freeze can last a very long time.

An alternative implementation is to record the information about the snapshot at one server, and have every server check this information before carrying out a modification. This approach is even less desirable than freezing, since it delays every modification.

We can avoid these problems if the system records information about happens before. This can be accomplished as follows: Each node includes its local time in every message it sends, and a node can perform a modification when the time of its own clock is later than any time it already heard of. (This protocol is a variation on one proposed in [17].) For example, if a user reads information from server X and then writes to server Y, information about server X's timestamp will flow to server Y, and server Y will delay performing the modification until its clock is larger than server X's clock was when it performed the modification to x. Note that the approach is cheap to implement and in practice modifications are unlikely to be delayed, provided server clocks are loosely synchronized.

Assuming timestamps as just described, we can implement snapshots as follows. A single server acts as the *snapshot coordinator*. To take a snapshot, a user communicates with this server. The server assigns a timestamp to the snapshot by reading its local clock. Information about the snapshot then flows to all servers, but this can be done in the background. The timestamp determines what modifications are contained in the snapshot: the snapshot includes modifications that occurred earlier than this time but not those that occurred after this time. To make this work, nodes need to track when modifications happen. This could be done by associating a "time-last-modified" timestamp with each object but in fact only information about recent modifications is needed, as discussed in Section 4.

The approach provides consistency without disruption. Taking a snapshot requires communication with the snapshot coordinator, which must be up and communicating. But such communication is also needed when all storage is at a single server and in either case, we can increase availability by using standard replication (e.g., [21]). There could be more than one snapshot coordinator; then snapshots would be ordered relative to modifications by using vector clocks [10, 23] with an entry for each coordinator.

With this approach it is possible that users might notice anomalies [6, 11]: a snapshot might fail to include a modification that a person knows happened before it, or it might include a modification that a person knows happened after it. Such anomalies require communication outside the system. E.g., if Bob takes a snapshot after being told by Alice that a certain modification has been made, the snapshot might nevertheless miss this modification. Anomalies are highly unlikely because current time synchronization technologies, such as NTP [20], synchronize clocks tightly enough to prevent them. Thus, the timestamp assigned to Bob's snapshot is highly likely to be greater than that of Alice's modification. (A similar use of loosely synchronized clocks to avoid such anomalies was proposed in [15].)

3 CREATING SNAPSHOTS

TimeLine provides snapshots for objects in Thor [13, 14]. This section provides a brief overview of Thor and then describes what happens when a user requests a snapshot.

3.1 Overview of Thor

Thor is a persistent object store based on the client-server model. Servers provide persistent storage for objects; we call these *object repositories (ORs)*. User code runs at client machines and interacts with Thor through client-side code called the *front end (FE)*. The FE contains cached copies of recently used objects.

A user interacts with Thor by running atomic transactions. An individual transaction can access (read and/or modify) many objects. A transaction terminates by committing or aborting. If it commits all modifications become persistent; if it aborts none of its changes are reflected in the persistent store.

Thor employs optimistic concurrency control to provide atomicity. The FE keeps track of the objects used by the current transaction. When the user requests to commit the transaction, the FE sends this information, along with the new states of modified objects, to one of the ORs. This OR decides whether a commit is possible; it runs a two-phase commit protocol if the transaction made use of objects at multiple ORs.

Thor uses timestamps as part of the commit protocol. Timestamps are globally unique: a timestamp is a pair, consisting of the time of the clock of the node that produced it, and the node's unique ID. Each transaction is assigned a timestamp, and can commit only if it can be serialized after all transactions with earlier timestamps and before all transactions with later timestamps.

Thor provides highly available and reliable storage for its objects, either by replicating their storage at multiple nodes using primary/backup replication, or by careful use of disk storage.

3.2 Taking a Snapshot

To create a snapshot, a user communicates with one of the ORs, which acts as the snapshot coordinator. This OR (which we will refer to as the SSC) assigns a timestamp to the snapshot by reading its local clock. The SSC serializes snapshot requests so that every snapshot has a unique timestamp, and it assigns later timestamps to later snapshots. It also maintains a complete *snapshot history*, which is a list of the assigned timestamps, and records this information persistently.

Once a snapshot request has been processed by the SSC, information about the snapshot must flow to all the ORs, and we would like this to happen quickly so that usually ORs have up-to-date information about snapshots. We could propagate snapshot information by having the SSC send the information to each OR, or we could have ORs request the information by frequent communication with the SSC, but these approaches impose a large load on the SSC if there are lots of ORs. Therefore, instead we propagate history information using gossip [3]: history information is piggybacked on every message in the system.

A problem with using gossip is that if the timestamp of the most recent snapshot in the piggybacked history is old, this might mean that no more recent snapshot has been taken, or it might mean that the piggybacked information is out of date. We distinguish these cases by having the SSC include its current time along with the snapshot history, thus indicating how recent the information is.

Gossip alone might not cause snapshot history information to propagate fast enough. To speed things up, we can superimpose a distribution tree on the ORs. To start the information flowing the SSC notifies a subset of ORs (those at the second level of the tree) each time it creates a new snapshot, and also periodically, e.g., every few seconds.

ORs can always fall back on querying the SSC (or one another) when snapshot information is not propagated fast enough. Even if the SSC is unreachable for a period of time, e.g., during a network partition, our design (see Section 4) ensures that the ORs still function correctly.

By using timestamps for snapshots, we are able to serialize snapshots with respect to user transactions: a snapshot reflects modifications of all user transactions with smaller timestamps. Serializing snapshots gives us atomicity, which is stronger than consistency. For example, we can guarantee that snapshots observe either all modifications of a user transaction, or none of them. However the approach of using timestamps also works in a system that provides atomicity for individual modifications but no way to group modifications into multi-operation transactions.

3.3 Snapshot Messages

Over time the snapshot history can become very large. However, usually only very recent history needs to be sent, because the recipient will have an almost up to date history already.

Therefore, our piggybacked snapshot messages contain only a portion of the snapshot history. In addition, a snapshot message contains two timestamps, TS_{curr} and TS_{prev}, that bound the information in the message: the message contains the timestamps of all snapshots that happened after TS_{prev} and before TS_{curr}.

An OR can accept a snapshot message m if $m.TS_{prev}$ is less than or equal to the TS_{curr} of the previous snapshot message that the OR accepted. Typically TS_{prev} is chosen by the sender to be small enough that the receiver is highly likely to be able to accept a message. Even with a TS_{prev} quite far in the past, the number of snapshot timestamps in the message is likely to be small. In fact, we expect a common case is that the message will contain no snapshot

timestamps, since the rate of taking snapshots is low relative to the frequency at which history information is propagated.

If a node is unable to accept a snapshot message, it can request the missing information from another node, e.g., the sender.

4 STORING SNAPSHOTS

This section and the two that follow describe the main functions of TimeLine. This section decribes how the OR saves snapshot information in the archive. Section 5 describes the archive service, which stores snapshot information when requested to do so by ORs and also provides access to previously stored snapshots. Section 6 describes how TimeLine carries out user commands to run computations in the past.

4.1 Design Overview

TimeLine provides snapshots by using a combination of current pages at ORs and pages stored in the archive. In particular, if a page has not been modified since a snapshot occurred, then the copy on the OR disk contains the proper information. On the other hand, if the page has been modified, its previous state needs to be written to the archive before its disk copy is overwritten. Thus we use a copy-on-write scheme, specialized to work for snapshots.

To create a snapshot page the OR needs to know all snapshots whose timestamps are less than t, the maximum timestamp of any transaction whose modification is being recorded on disk by overwriting the page. A fundamental problem with a gossip scheme, however, is that an OR's history information is never completely up to date. Therefore it might not know about all snapshots whose timestamps are less than t. In particular, since gossip takes time to arrive, an OR won't know about timestamps of snapshots that were taken very recently.

We could solve this problem by creating snapshot pages just in case they are needed, but that is expensive, especially since most of the time they won't be needed. Instead, we would like to avoid unnecessary creation of snapshot pages. We accomplish this by delaying overwriting of pages on disk until we know whether snapshot pages are needed. Our approach is based on details of the OR implementation, which is discussed in the next section.

4.2 Thor ORs

This section describes relevant details of how Thor ORs are implemented.

Thor stores objects in 8KB pages. Typically objects are small and there are many of them in a page. Each page belongs to a particular OR. An object is identified uniquely by the 32-bit ORnum of its OR and a 32-bit OREF that is unique within its OR. The OREF contains the PageID of the object's page and the object's offset within the page.

The FE responds to a cache miss by fetching an entire page from the OR, but only modified objects are shipped back from a FE to an OR when the transaction commits. This means that in order to write the modifications to the page on disk, the OR will usually need to do an *installation read* to obtain the object's page from disk. If we had to do this read as part of committing a transaction, it would degrade system performance.

Therefore instead the OR uses a volatile *Modified Object Buffer* (MOB) [5] to store modifications when a transaction commits, rather than writing them to disk immediately. This approach allows disk reads and writes to be deferred to a convenient time and also enables *write absorption*, i.e., the accumulation of multiple modifications to a page so that these modifications can be written to the page in a single disk write. Use of the MOB thus reduces disk activity and improves system performance.

Using the MOB does not compromise the correctness of the system because correctness is guaranteed by the transaction log: modifications are written to the transaction log and made persistent before a transaction is committed.

Entries are removed from the MOB when the respective modifications are installed in their pages and the pages are written back to disk. Removal of MOB entries (*cleaning* the MOB) is done by a *flusher thread* that runs in the background. This thread starts to run when the MOB fills beyond a pre-determined *high watermark* and removes entries until the MOB becomes small enough. The flusher processes the MOB in log order to facilitate the truncation of the transaction log. For each modification it encounters, it reads the modified object's page from disk, installs all modifications in the MOB that are for objects in that page, and then writes the updated page back to disk. When the flusher finishes a pass of cleaning the MOB it also removes entries for all transactions that have been completely processed from the transaction log.

The OR also has a volatile page buffer that it uses to satisfy FE fetch requests. Before returning a page to the requesting FE, the OR updates the buffer copy to contain all modifications in the MOB for that page. This ensures that pages fetched from the OR always reflect committed transactions. Pages are flushed from the page buffer as needed (LRU). Dirty pages are simply discarded rather than being written back to disk; the flusher thread is therefore the only part of the system that modifies pages on disk.

4.3 The Anti-MOB

Our implementation takes advantage of the MOB to delay the writing of snapshot pages until the snapshot history is sufficiently up to date.

To know whether to create snapshot pages, an OR needs to know the current snapshot status for each of its pages. This information is kept in its *snapshot page map*, which stores a timestamp for each page. When a snapshot page

for page p is created due to snapshot s, the corresponding snapshot page map entry is updated with the timestamp of s. All snapshot page map entries are initially zero. Snapshots of a page are created in timestamp order; therefore if the snapshot page map entry for a page has a timestamp greater than that of a snapshot, the OR must have already created a snapshot copy of the page for that snapshot.

An OR also stores what it knows of the snapshot history together with T_{gmax}, the highest SSC timestamp it has received.

Recall that the only time pages are overwritten on disk is when the MOB is cleaned. Therefore we can also create snapshot pages as part of cleaning the MOB. Doing so has two important advantages. First, it allows us to benefit from the installation read already being done by the flusher thread. Second, because the flusher runs when the MOB is almost full, it works on transactions that committed in the past. Thus, by the time a transaction's modifications are processed by the flusher, the OR is highly likely to have snapshot information that is recent enough to know whether snapshot pages are required.

A simple strategy is to start with the page read by the flusher thread and then use the MOB to obtain the right state for the snapshot page. For example, suppose the MOB contains modifications for two transactions T_1 and T_2, both of which modified the page, and suppose also that the snapshot for which the snapshot page is being created has a timestamp that is greater than T_1 but less than T_2. Then we need to apply T_1's modifications to the page before writing it to the archive.

However, this simple strategy doesn't handle all the situations that can arise, for three reasons. First, if the page being processed by the flusher thread is already in the page buffer, the flusher does not read it from disk. But in this case, the page already reflects modifications in the MOB, including those for transactions later than the snapshot. We could get the pre-states for these modifications by re-reading the page from disk, but that is undesirable. Hence, we need a way to revert modifications.

A second problem is that if a transaction modifies an object that already has an entry in the MOB (due to an older transaction), the old MOB entry is overwritten by the new one. However, we may need the overwritten entry to create the snapshot page.

Finally, if the MOB is processed to the end, we will encounter modifications that belong to transactions committed after T_{gmax}. Normally we avoid doing this: the flusher does not process MOB entries recording modifications of transactions with timestamps greater than T_{gmax}. But sometime the flusher must process such entries – when propagation of the snapshot history is stalled. When this happens we must be able to continue processing the MOB since otherwise the OR will be unable to continue committing transactions. But to do so, we need a place to store information about the pre-image of the page before it is overwritten on disk.

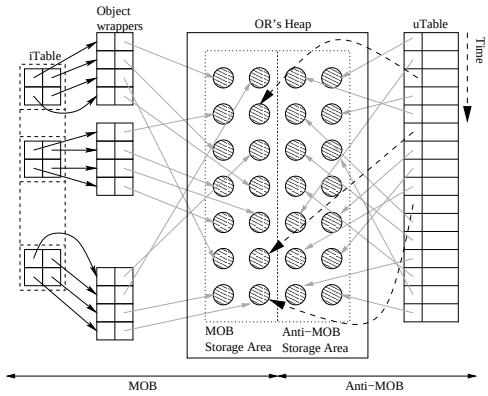

Figure 1. Structure of the Anti-MOB

The OR uses an in-memory buffer called the *Anti-MOB* to store overwritten MOB entries and transaction pre-images. The Anti-MOB (like the MOB) records information about objects. As shown in Figure 1, each Anti-MOB entry contains a pointer to the object's pre-image in the OR's heap, which it shares with the MOB. The entry also contains the timestamp of the pre-image's transaction, i.e., the transaction that caused that pre-image to be stored in the Anti-MOB. For example, if T_2 overwrites an object already written by T_1, a pointer to the older version of the object is stored in the Anti-MOB along with T_2's timestamp. Note that we don't make a copy of the old version; instead we just move the pointer to it from the MOB to the Anti-MOB.

Information is written to the Anti-MOB only when it is needed or might be needed. The Anti-MOB is usually needed when an object is overwritten, since the committing transaction is almost certain to have a timestamp larger than T_{gmax}. But when a page is read from disk in response to a fetch request from an FE, a modification installed in it might be due to a transaction with an old enough timestamp that the OR knows the pre-state will not be needed.

In addition, the first time an object is overwritten, we may create an additional Anti-MOB entry for the transaction that did the original write. This entry, which has its pointer to the pre-image set to null, indicates that the pre-image for that transaction is in the page on disk. For example, suppose transaction T_1 modified object x and later transaction T_2 modifies x. When the OR adds T_2's modified version of x to the MOB, it also stores $< T_2, x, x_1 >$ in the Anti-MOB to indicate that x_1 (the value of x created by T_1) is the pre-image of x for T_2. Furthermore, if T_1's modification to x was the first modification to that object by any transaction currently recorded in the MOB and if there is or might be a snapshot S before T_1 that requires a snapshot page for x's page, the OR also adds $< T_1, x, null >$ to the Anti-MOB, indicating that for S, the proper value of x is the one currently stored on disk.

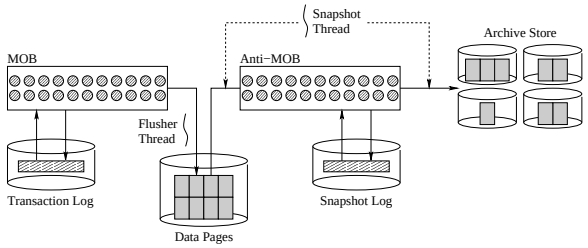

Figure 2. The Snapshot Subsystem in an OR

Let $SCAN_{(P,S)}$ = set of objects scanned for page P
for snapshot S and initialized to ∅

for each entry U_i in the Anti-MOB for a transaction
with timestamp ≥ S's timestamp, S.t
  if U_i.Oref ∉ $SCAN_{(P,S)}$ **then**
    $SCAN_{(P,S)}$ = $SCAN_{(P,S)}$ ∪ {U_i.Oref}
    install U_i into P
  endif
end

Figure 3. Algorithm for Creating Snapshot Pages

4.3.1 Creating Snapshot Pages

Now we describe how snapshot pages are created. Figure 2 shows the design of the snapshot subsystem, which works in tandem with the OR's MOB and transaction log. The work of taking snapshots is done partly by the flusher thread and partly by the *snapshot thread*, which also works in the background.

Snapshot pages are produced using both the MOB (to apply modifications) and the Anti-MOB (to revert modifications). As mentioned, to install a modification in the MOB, the flusher thread reads the page from disk if necessary. Then, it goes through the MOB and applies each modification for that page to the in-memory copy provided the transaction that made the modification has a timestamp less than some bound. Typically this bound will be T_{gmax}, but occasionally (when the history information is old), it will be larger than this.

Before applying a modification to the page, the flusher decides whether a pre-image is or might be needed. The pre-image might be needed if the transaction that did the modification has a timestamp greater than T_{gmax}; it will be needed if the OR knows of a snapshot with timestamp less than that of the transaction, and where the entry for the page in the snapshot page map is less than that of the snapshot.

To make a pre-image the flusher copies the current value of the modified object, x, from the page into the heap before overwriting the object. If there is no entry in the Anti-MOB for x and T, the transaction that caused that modification, it creates one; this entry points to x's value in the heap. Otherwise, if there is an Anti-MOB entry for x containing null, the flusher modifies the entry so that it now points to x's value in the heap.

If the page being modified is already in the page cache, the flusher applies the modifications to this copy. If this copy already contains modifications, the appropriate pre-images will exist in the Anti-MOB because they were put there as part of carrying out the fetch request for that page.

Once the flusher has applied the modifications to the page, and assuming a snapshot is required for the page, it makes a copy of the page and then uses the Anti-MOB to revert the page to the proper state. To undo modifications to page P as needed for a snapshot S with timestamp S.t, the OR scans the portion of the Anti-MOB with timestamps greater than S.t for pre-images corresponding to modifications to P and installs them in P. The algorithm used for creating snapshot pages is shown in Figure 3. The Anti-MOB can contain several pre-images corresponding to a single object. However, only the pre-image of the oldest transaction more recent than a snapshot will contain the correct version of the object for that snapshot. The algorithm uses the SCAN set to keep track of pre-images scanned and ensure that only the correct pre-image corresponding to an object (i.e., the first one encountered) is used for creating the snapshot page.

A snapshot page belongs to the most recent snapshot, S, with timestamp less than that of the earliest transaction, T, whose pre-image is placed in the page. The OR can only know what snapshot this is, however, if T has a timestamp less than T_{gmax}. Therefore snapshot pages are produced only when this condition holds. Section 4.5 discusses what happens when the OR cleans the MOB beyond T_{gmax}.

The snapshot page must be made persistent before the disk copy of the page can be overwritten. We discuss this point further in Section 4.4.

4.3.2 Discarding Anti-MOB Entries

Entries can be discarded from the Anti-MOB as soon as they are no longer needed for making snapshot pages.

The flusher makes snapshot pages up to the most recent snapshot it knows about for all pages it modifies as part of a pass of cleaning the MOB. At the end of this pass all entries used to make these snapshot pages can be deleted from the Anti-MOB. Such entries will have a timestamp that is less than or equal to T_{gmax} (since all snapshot pages for P for snapshots before T_{gmax} will have been made by that point).

When new snapshot information arrives at an OR, the OR updates T_{gmax}. Then it can re-evaluate whether Anti-MOB entries can be discarded. More specifically, when the OR advances its T_{gmax} from t_1 to t_2, it scans its Anti-MOB for entries with timestamps between t_1 and t_2. An entry can be discarded if the corresponding snapshot page of the most recent snapshot with a timestamp smaller than itself has already been created. Usually, all such entries will be removed (because there is no snapshot between t_1 and t_2).

4.4 Persistence and Failure Recovery

Once a page has been modified on disk, the Anti-MOB is the only place where pre-images exist until the snapshot pages that require those pre-images have been written to the archive. Yet the Anti-MOB is volatile and would be lost if the OR failed.

Normally snapshot pages are created as part of cleaning the MOB, and we could avoid the need to make information in the Anti-MOB persistent by delaying the overwrite of a page on disk until all its snapshot pages have been written to the archive. However, we decided not to implement things this way because the write to the archive might be slow (if the snapshot page is stored at a node that is far away in network distance). Also, there are cases where we need to overwrite the page before making snapshot pages, e.g., when propagation of snapshot history stalls (see Section 4.5).

Therefore we decided to make use of a snapshot log as a way of ensuring persistence of pre-images. Pre-images are inserted into the snapshot log as they are used for reverting pages to get the appropriate snapshot page. Thus, the creation of a snapshot page causes a number of entries for that page to be added to the snapshot log. Together, these entries record all pre-images for objects in that page whose modifications need to be reverted to recover the snapshot page.

Before a page is overwritten on disk, snapshot log entries containing the pre-images of modifications to the page are flushed to disk (or to the OR backups). The snapshot page can then be written to the archive asynchronously. Note that we can write snapshot log entries to disk in a large group (covering many snapshot pages) and thus amortize the cost of the flush.

Entries can be removed from the snapshot log once the associated snapshot pages have been saved to the archive.

4.4.1 Recovery of Snapshot Information

When the OR recovers from a failure, it recovers its snapshot history by communicating with another OR, initializes the MOB and Anti-MOB to empty, and initializes the snapshot page map to have all entries "undefined". Then it reads the transaction log and places information about modified objects in the MOB. As it does so, it may create some Anti-MOB entries if objects are overwritten. Then it reads the snapshot log and adds its entries to the Anti-MOB. The snapshot log is processed by working backwards. For example, suppose P required two snapshot pages P_{S_1} and P_{S_2} for snapshots S_1 and S_2 respectively ($S_1.t < S_2.t$). To recover P_{S_1}, the system reads P from disk, applies the entries in the snapshot log for P_{S_2}, and then applies the entries for P_{S_1}.

At this point, the OR can resume normal processing, even though it doesn't yet have correct information in the snapshot page map. The OR obtains this information from the archive. For example, when it adds an entry to the MOB, if the entry for that page in the snapshot page map is undefined, the OR requests the snapshot information from the archive. This is done asynchronously, but by the time the information is needed for cleaning the MOB, the OR is highly likely to know it.

4.5 Anti-MOB Overflow

When propagation of snapshot history information stalls, the OR must clean the MOB beyond T_{gmax} and this can cause the Anti-MOB to overflow the in-memory space allocated to it. Therefore at the end of such a cleaning, we push the current contents of the Anti-MOB into the snapshot log and force the log to disk. Then we can clear the Anti-MOB, so that the OR can continue inserting new entries into it.

When the OR gets updated snapshot information and advances its T_{gmax}, it processes the Anti-MOB blocks previous written to the snapshot log. It does this processing backward, from the most recently written block to the earliest block. Each block is also processed backward. Of course, if the OR learns that no new snapshots have been taken, it can avoid this processing and just discard all the blocks. It can also stop its backward processing as soon as it reaches a block all of whose entries are for transactions that are earlier than the earliest new snapshot it just heard about.

To process a block, the OR must use the appropriate page image. The first time it encounters an entry for a particular page, this will be the current page on disk. But later, it needs to use the page that it already modified. Therefore the OR uses a table that records information about all pages processed so far. These pages will be stored in memory if possible, and otherwise on disk.

Once snapshot pages have been written to the archive, the OR removes the Anti-MOB blocks from the snapshot log and discards all the temporary pages used for processing the blocks.

5 ARCHIVING SNAPSHOTS

This section describes the design of the archive store. We require that the archive provide storage at least as reliable and available as what Thor provides for the current states of objects. For example, if Thor ORs are replicated, snapshot pages should have the same degree of replication.

5.1 Archive Store Abstraction

Table 1 shows the interface to the archive. It provides three operations: put, get, and getTS.

The put operation is used to store a page in the archive. Its arguments provide the value of the snapshot page and its identity (by giving the ORnum of its OR, its PageID within the OR, and the timestamp of the snapshot for which it was created). The operation returns when the page has been

Operation	Description
put(ORnum, PageID, TS_{ss}, $Page_{ss}$)	Stores the snapshot page $Page_{ss}$ into the archive
get(ORnum, PageID, TS_{ss})	Retrieves the snapshot page from the archive.
getTS(ORnum, PageID)	Retrieves the latest timestamp for the page.

Table 1. Interface to the Archive

stored reliably, e.g., written to disk, or written to a sufficient number of replicas.

The get operation is used to retrieve a page from the archive. Its arguments identify the page by giving its ORnum and PageID, and the timestamp of the snapshot of interest. If the archive contains a page with that identity and timestamp, it returns it. Otherwise, if there is a later snapshot page for that page, it returns the oldest one of these. Otherwise it returns null.

It is correct to return a later snapshot page because of the way we produce snapshot pages: we only produce them as pre-images before later modifications. If there is no snapshot page with the required timestamp but there is a later one, this means there were no changes to that page between the requested snapshot and the earliest later snapshot, and therefore the information stored for the later snapshot is also correct for this snapshot.

The getTS operation takes an ORnum and PageID as arguments. It returns the latest timestamp of a snapshot page for the identified page; it returns zero if there is no snapshot for that page. It is used by the OR to reinitialize its snapshot page map when it recovers from a failure.

5.2 Archive Store Implementations

We considered three alternatives for implementing the archive store: a local archive store at each OR, a network archive store on the local area network, and an archive store on servers distributed across a wide area network.

A local archive store archives the snapshot pages created at its OR. As we will discuss further in Section 6, all snapshot pages are fetched via the OR. Therefore this approach will provide the best performance (for transactions on snapshots) as compared to the other two implementations. On the other hand, the archive can potentially become very large. In addition, ORs may differ widely in how much archive space they require, so that it may be desirable to allow sharing of the archive storage.

In a network archive store, snapshots are archived in a specialized storage node located close to a group of ORs, e.g., on the same LAN. This design allows sharing of what might be a high-end storage node or nodes, e.g., a node with a disk array that provides high reliability. The impact of a network archive store on the performance of fetching snapshot pages depends on the speed of the underlying network connection. For example, on a gigabit LAN, we expect the performance of the network archive store to be close to that of a local archive store.

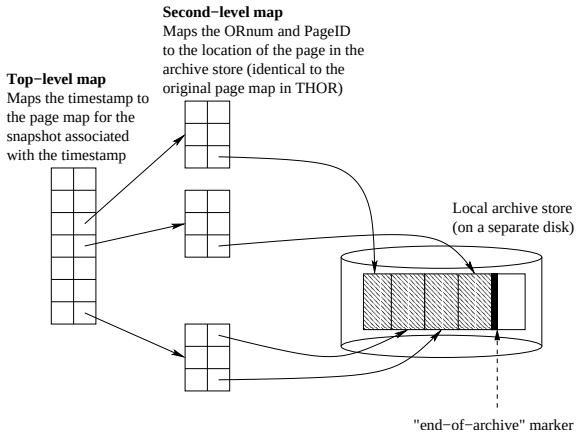

Figure 4. Storage System of an Archive Store Node

A distributed archive store is implemented as a set of storage nodes that are distributed across the wide area network. All ORs share the archive store and therefore this approach allows even more sharing than the network archive store. Also, with proper design, the system can provide excellent reliability, automatic load-balancing, and self-organization features, similar to what can be achieved, e.g., in peer-to-peer systems [25, 28]. The downside of this scheme is that the data may be located far away from the ORs, making it costly to run transactions on snapshots.

We expect the choice of where to place the archive store to mainly affect the cost of running computations on snapshots. All designs should perform similarly with respect to taking snapshots since writing snapshot pages to the archive is done asynchronously.

5.3 Implementation Details

In this section, we describe the archive store implementations.

In a local archive store, snapshots are stored on a separate disk (or disks) so that requests to read or write snapshot pages don't interfere with normal OR activity, e.g., fetching current pages.

The disk is organized as a log as shown in Figure 4. Snapshots pages are appended to the log, along with their identifier (ORnum, PageID, and timestamp of the snapshot for which the snapshot page was created). This organization makes writing of snapshot pages to disk fast (and we could speed it up even more by writing multiple pages to the log at once). A directory structure is maintained in primary memory to provide fast access to snapshot pages. The identifier information in the log allows the OR to recover the directory from the log after a failure.

A network archive store uses the same log-structured

organization for its disk. The OR runs a client proxy that receives calls of archive operations such as put, turns them into messages, and communicates with the storage server using UDP.

Each storage node in a distributed archive store also uses the same log-structured organization. These nodes may be ORs, separate storage nodes, or a combination of the two.

Again, each OR runs a client proxy that handles calls of archive operations. But now the proxy has something interesting to do.

The main issue in designing a distributed archive store is deciding how to allocate snapshot pages to storage nodes. We decided to base our design on consistent hashing [9]. With this approach each storage node in the (distributed) archive store is assigned an identifier, its *nodeID*, in a large (160 bit) space, organized in a ring. The nodeIDs are chosen so that the space is evenly populated. Each snapshot page also has an identifier, its *archiveID*. To map the snapshot page to a storage node, we use the *successor* function as in Chord [28]; the node with the nodeID that succeeds the snapshot page's archiveID will be responsible for archiving that page. However, note that any deterministic function will work, e.g., we could have chosen the closest node in the ID space, as in Pastry [25].

The archiveID of a snapshot page is the SHA1 hash of the page's ORnum and PageID. This way, all snapshot pages of the same page will be mapped to the same storage node. This choice makes the implementation of get and getTS efficient because they can be implemented at a single node.

To interact with the archive, the client proxy at the ORs needs to map the NodeID of the node responsible for the page of interest to its IP address. Systems based on consistent hashing typically use a multi-step routing algorithm [25, 28] to do this mapping although recent work shows that routing can be done in one step [7]. However, routing is not an important concern for us because the client proxy at an OR can cache the IP addresses of the nodes responsible for its OR's snapshot pages. Since the population of storage nodes is unlikely to change very frequently, the cache hit rate will be high. Therefore, the cost of using distributed storage will just be the cost of sending and receiving the data from its storage node.

We chose to use consistent hashing for a number of reasons. If the node IDs and snapshot IDs are distributed reasonably, it provides good load balancing. It also has the property of allowing nodes to join and leave the system with only local disruption: data needs to be redistributed when this happens but only nodes nearby in ID space are affected.

Of course, snapshot pages need to be replicated to allow storage nodes to leave the system without losing any data. Thus a snapshot page will need to be archived at several nodes. This does not slow down the system but does generate more network traffic. One benefit of replication is that snapshot pages can be retrieved in parallel from all the replicas and the first copy arriving at the OR can be used to satisfy the request. As a result, the cost of retrieving a snapshot page is the cost of fetching it from the nearest replica.

6 TRANSACTIONS ON SNAPSHOTS

Users can run transactions on a snapshot by specifying a time in the "past" at which the transaction should execute. We only allow read-only transactions, since we don't want transactions that run in the past to be able to rewrite history.

The most recent snapshot, S, with timestamp less than or equal to the specified timestamp will be the snapshot used for the transaction. The FE uses the timestamp of S to fetch snapshot pages. Since the FE needs to determine S, the timestamp specified by the user must be less than the T_{gmax} at the FE.

In this section, we first provide an overview of the FE. Then we describe how the FE and OR together support read-only transactions on snapshots.

6.1 Thor Front-End

This section contains a brief overview of the FE. More information can be found in [2, 13, 14].

To speed up client transactions, the FE maintains copies of persistent objects fetched from the OR in its in-memory *cache*. It uses a *page map* to locate pages in the cache, using the page's PageID and the ORnum. When a transaction uses an object that isn't already in the cache, the FE fetches that object's page from its OR. When a transaction commits, only modified objects are shipped back to the OR.

Pages fetched from the OR are stored in page-size page frames. However when a page is evicted from the cache (to make room for an incoming page), its hot objects are retained by moving them into another *compacted frame* that stores such objects.

Objects at ORs refer to one another using OREFs (recall that an OREF identifies an object within its OR). To avoid the FE having to translate an OREF to a memory location each time it follows a pointer, we do *pointer swizzling*: the first time an OREF is used, it is replaced by information that allows the object to be efficiently located in the FE cache. When an OREF is swizzled, it is changed to point to an entry in the *Resident Object Table* (ROT). This entry then points to the object if it is in the cache. This way finding an object in the cache is cheap, yet it is also cheap to discard pages from the cache and to move objects into compacted pages.

6.2 Using Snapshots

When a user requests to run a transaction in the past, the FE cache is likely to contain pages belonging to other time

lines, e.g., to the present. Similarly, when the user switches back from running in the past to running in the present, the FE cache will contain snapshot pages. In either case, the FE must ensure that the user transaction uses the appropriate objects: objects from current pages when running in the present; objects from snapshot pages for the requested snapshot when running in the past.

One approach to ensuring that the right objects are used is to clear the FE's cache each time the user switches to running at a different time. However, this approach forces the FE to discard hot objects from its cache and might degrade transaction performance. This could happen if the user switches repeatedly, e.g., from the current database to a snapshot and back to the current database again.

Therefore we chose a different approach. We extended the FE's cache management scheme so that the FE caches pages of multiple snapshots at the same time. Yet the FE manages its cache so that a transaction only sees objects of the correct version. Our implementation is optimized to have little impact on the running of transactions in the present, since we expect this to be the common case.

When the user switches from one time line to another, all entries in the ROT will refer to objects belonging to the time line it was using previously. We must ensure that the transaction that is about to run does not use these objects. We accomplish this by setting every entry in the ROT to null; this, in turn, causes ROT "misses" on subsequent accesses to objects.

When a ROT miss happens, the FE performs a lookup on the page map to locate the page on the cache. To ensure that we find the right page, one belonging to the time line being used by the currently running transaction, we extend the page map to store the timestamp for each page; current pages have a null timestamp. The page map lookup can then find the right page by using the timestamp associated with the currently running transaction. The lookup will succeed if appropriate page is in the cache. Otherwise, the lookup fails and the FE must fetch the page.

6.3 Fetching Snapshot Pages

The FE fetches a snapshot page by requesting it from the OR that stores that page. The reason for using the OR is that snapshots consist of both current pages and pages in the archive; the OR can provide the current page if that is appropriate, and otherwise it fetches the required page from the archive.

Page fetch requests from the FEs to the ORs are extended to contain a snapshot timestamp, or null if the FE transaction is running in the present. If the timestamp is not null, the OR uses it to determine the correct snapshot page by consulting its snapshot page map.

The snapshot page map stores the most recent snapshot timestamp for each page. If the timestamp of the request is less than or equal to the timestamp stored in the map for the requested page, the OR requests the snapshot page from the archive and returns the result of the fetch to the FE. Otherwise, the OR must create the snapshot page at that moment. It does this by using the page copy on disk or in the page buffer, plus the information in the MOB and Anti-MOB. It returns the resulting page to the FE. In addition if the snapshot page is different from the current page, the OR stores the page in the archive and updates its snapshot page map; this avoids creating the snapshot page again.

7 EXPERIMENTS

In this section, we present a performance evaluation of our snapshot service. We do this by comparing the performance of the original Thor to that of *Thor-SS*, the version of Thor that supports snapshots. Thor is a good basis for this study because it performs well. Earlier studies showed that it delivers comparable performance to a highly-optimized persistent object system implemented in C++ even though the C++ system did not support transactions [14]. Our experiments show that Thor-SS is only slightly more costly to run than Thor.

Thor-SS is a complete implementation of our design in the absence of failures; it includes making the Anti-MOB persistent but not the recovery mechanism. This implementation is sufficient to allow us to measure performance in the common case of no failures.

Our experimental methodology is based on the single-user OO7 benchmark [1]; this benchmark is intended to capture the characteristics of various CAD 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 bidirectional *connection* objects, reachable from a single *root* atomic part; each atomic part has three connections. We employed the medium OO7 database configuration, where each composite part contains 200 atomic parts. The entire database consumes approximately 44 MB.

We used OO7 traversals T1 and T2B for our experiments. T1 is read-only and measures the raw traversal speed by performing a depth-first search of the composite part graph, touching every atomic part. T2B is read-write; it updates every atomic part per composite part.

We used a single FE and a single OR for our experiments. The OR and FE ran on separate Dell Precision 410 workstations (Pentium III 600 Mhz, 512 MB RAM, Quantum Atlas 10K 18WLS SCSI hard disk) with Linux kernel 2.4.7-10. Another identical machine is used for the network storage node in the network archive store, as well as for running a simulator (developed using the SFS toolkit [19]) for simulating a 60 node wide-area archive store. The machines were connected by a 100 Mbps switched Ethernet.

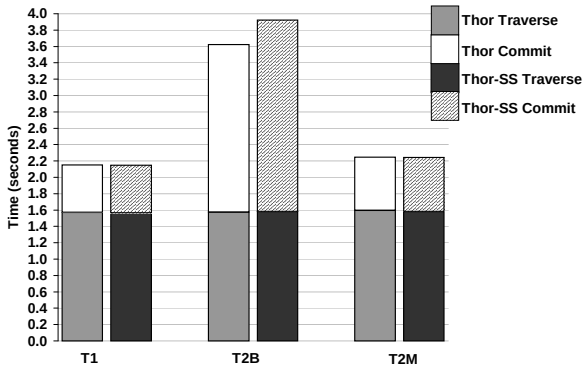

Figure 5. Baseline Comparison - Thor vs. Thor-SS

7.1 Foreground Costs

The first set of experiments compares the foreground cost of running Thor with Thor-SS. Particularly, we looked at the extra cost of committing transactions in Thor-SS when no MOB cleaning is occurring. The experiments used a 32 MB MOB to ensure MOB cleaning does not happen.

Thor-SS has extra work to do when the committing transaction overwrites previously modified objects. When overwriting occurs in Thor, the OR writes the modified objects to the MOB and modifies the MOB table to point to the new entry. This work also occurs in Thor-SS, but in addition Thor-SS must add the previous entry to the Anti-MOB. In either case this work is in the foreground: the OR doesn't respond to the FE's commit request until after work is done.

We measured the average execution times for T1 and T2B. In each case, the traversal was executed 20 times and each run was a separate transaction. The experiments were done with a hot FE cache that contained all the objects used in the traversal. This way we eliminated the cost of fetching pages from the OR, since otherwise the cost of fetching pages would dominate the execution time. The number of pages used in the traversals was identical in both Thor and Thor-SS.

The results of these experiments are shown in Figure 5. The figure shows that there is no significant difference between running the read-only traversal T1 in Thor and Thor-SS: in this case no writing to either the MOB or Anti-MOB occurs. In T2B, however, 98,600 objects were modified, which causes 98,600 overwrites in the OR's MOB and each overwrite leads to an object being moved to the Anti-MOB, leading to an 8% slowdown.

The CPU time for the OR to commit a T2B transaction is 1.65 seconds in Thor and 1.89 seconds in Thor-SS. The difference is the time taken by Thor-SS to do the extra work of storing overwritten objects into the Anti-MOB; the data show that it takes approximately 2.4 μsecs to update the Anti-MOB for each overwritten object.

The workload generated by T2B is not a very realistic representation of what we expect users to do: users are unlikely to repeatedly modify the same objects or modify almost all objects in the database. Therefore, we developed another workload that is more realistic. This workload, T2M, is a modified version of T2B: each T2M traversal randomly updates 10% (or 9,860 objects per traversal) of the objects that the original T2B modifies. In T2M there is much less overwriting than in T2B: each traversal overwrites an average of 3,852 objects. As shown in Figure 5, there is little difference in performance between Thor and Thor-SS for this traversal.

7.2 Impact of Snapshot Page Creation

In this section we look at the additional work that Thor-SS has to do when the MOB is cleaned. Thor cleans the MOB in the background using a low-priority process (the flusher) running in small time slices. Therefore, unless the OR is very heavily loaded, cleaning does not show up as a user-observable slowdown. We would like the same effect in Thor-SS, even when snapshot pages are being created.

Thor cleans several pages at once. To amortize disk seek time, it works on segments, which are 32 KB contiguous regions of disk, each containing four pages. Thor can read or write a segment about as cheaply as reading or writing a page, and if more than one page in the segment has been modified, cleaning costs will be lower using this approach.

The results in this section were obtained by running T2M (the modified T2B). In each experiment T2M ran 25 times, i.e., we committed 25 transactions. We ran experiments both with and without cleaning. To avoid cleaning we used an 8 MB MOB; to cause cleaning to occur we used a 4 MB MOB. With the 4 MB MOB, cleaning begins when the 12^{th} traversal commits. During the remaining traversals, 1,141 segments were cleaned and 3,097 pages were modified; an average of 56 modified objects (consuming 2.6 KB) were installed in each segment. We continued to use a 32 MB cache at the OR so that no disk activity occurred during the cleaning part of the experiment.

To get a sense of the OR load due to cleaning, we measured the OR CPU time required to do cleaning. The results of these experiments are shown in Table 2. In each case the measurement started at the point where the flusher thread decided that a segment needs to be updated. At that point the segment is already in memory. The measurement for Thor stops when all modifications have been installed in the segment and their entries removed from the MOB, but the segment has not yet been written to disk. The results show the average cost of cleaning a segment.

The table shows two measurements for Thor-SS. The first is for the case where no snapshots need to be made; however, Thor-SS still has extra work to do, since entries need to be removed from the Anti-MOB. The second measurement is for Thor-SS when snapshot pages are being made. In this case the measurement includes the cost of creating snapshot pages, reverting the changes, and writing the pre-images to the snapshot log; the measurement stops

System	Cleaning Time
Thor	6.74 ms
Thor-SS without snapshot	6.87 ms
Thor-SS with snapshot	11.09 ms

Table 2. Time Taken to Clean a Segment

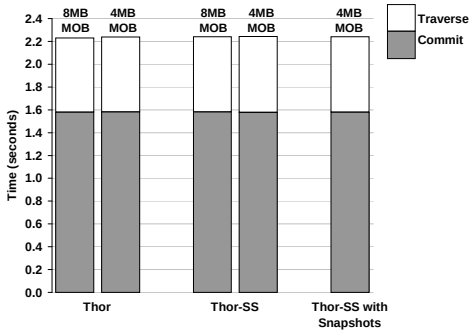

Figure 6. Cost of Taking Snapshots

before the snapshot log is forced to disk (since the flusher does not wait for this to complete). This is a worst case experiment in which every page requires a corresponding snapshot page, e.g., it represents a case where a snapshot has just been requested.

The impact on users of the additional cleaning activity in Thor-SS depends on the load at the OR. What we can hope is that when the OR has some spare cycles, we can run Thor-SS with cleaning and snapshot pages entirely in the background, just like Thor cleaning can be done in the background in this case. This is a reasonable expectation since, as we showed in Table 2, Thor-SS requires a modest amount of time to do cleaning even in the case where every page requires a snapshot page.

Figure 6 shows the results of these experiments. For both Thor and Thor-SS, the experiments compare the cost of running traversals with and without cleaning. The performance of the traversals is identical from the viewpoint of the user transaction: cleaning the MOB does not cause any slowdown. In particular, even in the case where every page that is cleaned requires a snapshot page, there is no slowdown.

7.3 Running Transactions on Snapshots

In this section, we examine the performance of running transactions in the past. We also compare this performance with that of running transactions in the present. These experiments use traversal T1, since we only allow read-only transactions to run in the past.

We ran T1 using a cold FE cache so that every page used by T1 must be fetched. Running T1 causes 5,746 snapshot pages to be fetched. We also used an empty page cache at the OR. Using a cold FE cache and an empty OR page cache provide us with a uniform setup to measure the slowdown that is caused by running T1 on a snapshot as compared to

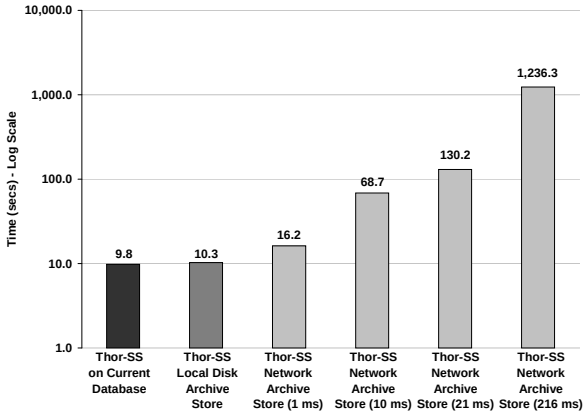

Figure 7. T1 Performance on Snapshot

running it on the current database. As discussed in Section 5, whenever there is a miss in the FE cache, the FE fetches the missing page from its OR. This is true whether the FE requires a snapshot page or a current page.

Figure 7 shows the results. The figure shows that running in the past is only slightly slower (approximately 5% slower) than running in the present when snapshots are stored on the OR's disk. This slowdown is due entirely to processing at the FE: in particular, it is due to an additional level of indirection in the FE's page map that affects transactions in the past but not in the present.

The figure also shows what happens when the archive store is not located at the OR. The slowdown is relatively small if the snapshot pages are archived at a nearby node that is connected to the OR by a 100 Mbps network, so that the cost to send 8 KB of data from the storage node to the OR is 1 ms: in this case the time taken to run T1 is 16.2 secs. Performance is more degraded when the the time required to send 8 KB of data increases to 10 ms, such as on a heavily loaded or slower (e.g., 10 Mbps) LAN; hence the performance of T1 is slower by an order of magnitude (68.7 secs). Performance is further degraded when the archive is even farther away.

These results imply that if the remote storage node were very close to the OR, e.g., connected to it by a gigabit network, performance of running on a snapshot would be essentially identical when storing the archive at the OR or at the remote node. Performance of running on snapshots degrades substantially as the storage moves farther away, but use of remote shared storage may still be the best choice because of the advantage of using such a shared facility.

8 RELATED WORK

A number of systems have provided snapshots, either for use in recovering from failure, or to provide the ability to look at past state. There is also a large body of work on systems that take checkpoints [4, 16]. Here we compare TimeLine to systems that provide snapshots, since they are most closely related.

The Plan 9 [24] file system is a one-server system that provides an atomic dump operation for backups. During a dump operation, the in-memory cache is flushed, the file system is frozen, and all blocks modified since the last dump operation are queued on disk for writing to the WORM storage. This system delays access to the file system while the dump is being taken.

The Write Anywhere File Layout (WAFL) [8] is a file system designed for an NFS server appliance that provides a snapshot feature. The first time a page is modified after the snapshot is taken, WAFL writes the page to a new location (thus preserving the old disk copy for the snapshot); it also modifies the disk copy of the directory block that points to this page. The only block that needs to be duplicated eagerly during the snapshot is the root inode block. A snapshot includes the current values of all dirty pages in the cache; if such a page is modified before being written to disk, the proper snapshot value for the page would not be known. To avoid this problem WAFL blocks incoming requests that would modify dirty data that was present in the cache at the moment the snapshot was requested. However, it does not block any other requests.

Frangipani [31] is a distributed file system built on the Petal [12] distributed storage system. Frangipani uses Petal's snapshot features to create a point-in-time copy of the entire distributed file system. This point-in-time copy can then be copied to a tape or WORM device for backup. To maintain consistency of the snapshot, the backup process requests an exclusive lock on the system. This requires stopping the system so that all servers can learn about the snapshot. When it hears about a snapshot, a server flushes its cache to disk and blocks all new file system operations that modify the data. These operations remain blocked until the lock is released. After that point, the backup service takes a Petal snapshot at each server to create the point-in-time copy. This is done copy-on-write without delaying user requests. (Petal doesn't block to take a snapshot but it supports only a single writer.)

Unlike Plan 9, WAFL, and Frangipani, we do not require the system to block users during the snapshot. Furthermore, TimeLine is a distributed system, unlike Plan 9 and WAFL. Frangipani is also distributed but requires a global lock to take a snapshot, which can lead to a substantial delay.

The Elephant file system [26] maintains all versions and uses timestamps to provide a consistent snapshot. A version of a file is defined by the updates between an open and close operation on the file. The file's inode is duplicated when the file is opened and copy-on-write is used to create a pre-modification version of the file. Concurrent sharing of a file is supported by copying the inode on the first open and appending to the inode-log on the last close.

TimeLine is similar to Elephant in its use of timestamps but it is a distributed system while Elephant is a one-server system. In addition TimeLine makes snapshots on command while Elephant takes them automatically. Taking snapshots on command reduces the size of archive storage. It also arguably provides more useful information, since users can decide what snapshots they want, rather than having to contend with huge amounts of data that is hard to relate to activities that interest them.

Some database systems [22, 27] also provide access to historical information. These systems allow queries based on time information that is either provided explicitly by the user or implicitly by the system. When the time field is overwritten, the old value of the record is retained.

An alternative design for snapshots is to store either an undo or redo log and then materialize the snapshot on demand: with an undo log, the current state would be rolled backward to the desired time line, while with a redo log the initial state would roll forward. Use of such logs was pioneered in database system and some early work on the Postgres Storage Manager [29] even proposed keeping the entire state as a redo log. This approach was subsequently rejected because of the cost of materializing the current system state [30]. These results can also be taken to show that materializing snapshot state is not practical: snapshots must be available for use without a huge delay

9 CONCLUSION

This paper describes TimeLine, an efficient archive service for a distributed storage system. TimeLine allows users to take snapshots on demand. The archive is stored online so that it is easily accessible to users. It enables "time travel" in which a user runs a computation on an earlier system state.

TimeLine allows snapshots to be taken without disruption: user access to the store is not delayed when a snapshot is requested. Yet our scheme provides consistent snapshots; in fact it provides atomicity, which is stronger than consistency. The techniques used in TimeLine will also work in a system that only provides atomicity for individual modifications. Furthermore timestamps can be used to order snapshots in any system that implements logical clocks, and logical clocks are easy and cheap to implement.

TimeLine provides consistent snapshots with minimum impact on system performance: storing snapshot pages occurs in the background. Our scheme is also scalable: snapshots are requested by communicating with a single node and information is propagated to the nodes in the system via gossip.

The main performance goal of our system is that snapshots should not interfere with transactions on the current database and degrade their performance. An additional goal is to provide reasonable accesses to snapshots, i.e., to computations that run in the past. Our experiments show that computations in the past run as fast as in the present when

the archive state is co-located with the current system state, and run reasonably well using shared archive storage that is close, e.g., on the same LAN, or connected by a high speed link. The results also show that taking snapshots has negligible impact on the cost of concurrently running computations, regardless of where the archived data is stored.

10 Acknowledgements

The authors gratefully acknowledge the help received from Liuba Shrira, Rodrigo Rodrigues, Sameer Ajmani, our shepherd Jeff Chase, and the referees. This research is supported by NTT and NSF grant ANI-0225660.

References

[1] M. J. Carey, D. J. DeWitt, and J. F. Naughton. The OO7 Benchmark. *SIGMOD Record*, 22(2):12–21, Janaury 1994.

[2] M. Castro, A. Adya, B. Liskov, and A. C. Myers. HAC: Hybrid Adaptive Caching for Distributed Storage Systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, Saint-Malo, France, October 1997.

[3] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinchart, and D. Terry. Epidemic Algorithms for Replicated Database Maintenance. In *Proceedings of the 6th ACM Symposium on Principles of Distributed Computing*, Vancouver, Canada, August 1987.

[4] E. Elnozahy, L. Alvisi, Y-M Wang, and D. Johnson. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. Technical Report CMU-CS-99-148, Carnegie Mellon University, June 1999.

[5] S. Ghemawat. *The Modified Object Buffer: A Storage Management Technique for Object-Oriented Databases*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, September 1995.

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

[7] A. Gupta, B. Liskov, and R. Rodrigues. One Hop Lookups for Peer-to-Peer Overlays. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS)*, Lihue, Hawaii, May 2003.

[8] D. Hitz, J. Lau, and M.A. Malcom. File System Design for an NFS File Server Appliance. In *Proceedings of the USENIX Winter Technical Conference*, San Francisco, CA, USA, January 1994.

[9] D. Karger, E. Lehman, F. Leighton, M. Levine, D. Lewin, and R. Panigraphy. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots in the World Wide Web. In *Proceedings of the 29th ACM Symposium on Theory of Computing*, El Paso, TX, May 1997.

[10] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing High Availability Using Lazy Replication. *ACM Transactions on Computer Systems*, 10(4):360–391, November 1992.

[11] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[12] E. Lee and C. Thekkath. Petal: Distributed Virtual Disks. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Boston, MA, October 1996.

[13] 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 *Proceedings of the ACM SIGMOD Conference*, Montreal, Canada, June 1996.

[14] B. Liskov, M. Castro, L. Shrira, and A. Adya. Providing Persistent Object in Distributed Systems. In *Proceedings of the 13th European Conference on Object-Oriented Programming*, Lisbon, Portugal, June 1999.

[15] D. Lomet and B. Salzberg. *Temporal Databases: Theory, Design, and Implementation*, pages 388–417. Addison-Wesley, March 1993.

[16] D. Lowell, S. Chandra, and P. Chen. Exploring Failure Transparency and the Limits of Generic Recovery. In *Proceedings of the 4th Symposium on Operating System Design and Implementation*, San Diego, CA, October 2000.

[17] J. Lundelius. *Topics in Distributed Computing: The Impact of Partial Synchrony, and Modular Decomposition of Algorithms*. PhD thesis, Massachusetts Institute of Technology, 1988.

[18] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann, March 1996.

[19] D. Mazières. A toolkit for user-level fi le systems. In *Proceedings of the USENIX Annual Technical Conference*, Boston, MA, June 2001.

[20] D. Mills. Network Time Protocol (Version 3) specifi cation, implementation and analysis. Technical Report Network Working Group Report RFC-1305, University of Delaware, Newark, DE, USA, March 1992.

[21] B. Oki and B. Liskov. Viewstamped Replication: A General Primary Copy. In *Proceedings of the 7th Symposium on Principles of Distributed Computing (PODC)*, Toronto, Ontario, Canada, August 1988.

[22] G. Ozsoyoglu and R. Snodgrass. Temporal and Real-Time Databases: A Survey. *IEEE Transactions on Knoweldge and Data Engineering*, 7(4):513–532, August 1995.

[23] D. S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. Walker, E. Walton, J. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering*, SE-9(3):240–247, May 1983.

[24] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, Summer 1995.

[25] A. Rowston and P. Drushel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM Internationl Conference on Distributed Systems Platrofms (Middleware)*, Heidelberg, Germany, November 2001.

[26] D. Santry, M. Feeley, N. Hutchinson, A. Veitch, R. Carton, and J. Ofi r. Deciding When to Forget in the Elephant File System. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, Charleston, SC, USA, December 1999.

[27] A. Steiner and M. C. Norrie. Implementing Temporal Databases in Object-Oriented Systems. In *Proceedings of the 9th Int'l Conference on Advanced Information Systems Engineering*, Barcelona, Spain, June 1997.

[28] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balariishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proceedings of the ACM SIGCOMM Conference*, San Deigo, CA, USA, August 2001.

[29] M. Stonebraker. The Design of the POSTGRES Storage System. In *Proceedings of the 13th International Conference on Very-Large Data Bases*, Brighton, England, UK, September 1987.

[30] M. Stonebraker and J. Hellerstein, editors. *Readings in Database Systems*, chapter 3. Morgan Kaufmann Publishers, Inc., 3rd edition, 1998.

[31] C. Thekkath, T. Mann, and E. Lee. Frangipani: A Scalable Distributed File System. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, Sanit-Malo, France, October 1997.