Snapshots in a Distributed Persistent Object Storage System

by

Chuang-Hue Moh

Bachelor of Applied Science (Computer Engineering)
Nanyang Technological University, Singapore (2000)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2003

Author . . . .
Department of Electrical Engineering and Computer Science
May 27, 2003

Certified by . . . .
Barbara H. Liskov
Ford Professor of Engineering
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Eternity is a mere moment, just enough for a joke.

Hermann Hesse (1877-1962)

Abstract

The ability to take transaction-consistent snapshots of a distributed persistent object store is useful for online data archive, backup, and recovery. However, most modern snapshot utilities such as those in file systems and database systems are not designed for distributed environments and directly applying these methods to a distributed environment results in poor performance. This thesis describes an efficient snapshot scheme for a distributed persistent object store.

Our snapshot scheme provides two basic operations to the users: (1) take a snapshot of the persistent object store and (2) execute a read-only transaction on a previously taken snapshot. Snapshots in our system are based on time. Users take a snapshot by indicating that the snapshot should happen "now" (at the current time). To run a read-only transaction on a snapshot of the system, the user specifies the time in the past at which the transaction should be executed. Our snapshot scheme guarantees serializability with respect to concurrent transactions. It also guarantees external consistency based on the assumption of loosely synchronized clocks among the object stores.

A centralized snapshot coordinator is responsible for keeping track of all snapshots in the system. Snapshots are committed only at this snapshot coordinator and propagated throughout the entire system via a gossip mechanism. Concurrent transactions are allowed to proceed without having to wait for all the object stores to learn that a snapshot is being taken. However, this leads to the problem that update transactions may modify objects before they can be used for a snapshot that has occurred earlier We use an in-memory buffer called the Anti-MOB to store the pre-images of transactions. The Anti-MOB allows us to revert modified pages to the proper state when the object store learns of a snapshot.

Since snapshots are long-lived data, the amount of storage required for them becomes large as more and more snapshots are taken over time. In the long run, the storage required for archiving snapshots dwarfs the storage capacity of the persistent object store. We present and evaluate three different implementations of the archive store for our snapshot subsystem: (1) local archive store, (2) network archive store, and (3) p2p archive store.

Our scheme achieves low space overhead using copy-on-write and does not degrade the performance of user transactions on the current database. The performance of transactions that run on snapshots depends on the archive store implementation; these transactions perform well when the archive store is nearby on the network but much more slowly in a p2p implementation in which the archived data is likely to be far away from the users.

Thesis Supervisor: Barbara H. Liskov
Title: Ford Professor of Engineering

Acknowledgments

First and foremost, I would like to thank my research adviser, Professor Barbara Liskov[1]. Her patience, guidance, support, and encouragement made the completion of this work possible. I am extremely fortunate and honored to have her as my adviser, and like all her students, I remain in awe of her technical thinking and writing.

I would also like to thank Sameer Ajmani, Dorothy Curtis, Rodrigo Rodrigues, Steven Richman, and Liuba Shrira for creating a very supportive work environment. Sameer and Rodrigo offered me great advice on the design of the system and the writing of the thesis; Dorothy and Liuba gave me a lot of help in understanding THOR and are instrumental in the implementation the system. Steven always provided insights to the problems that I encountered. My appreciation also goes to Ben Leong for being a wonderful roommate and for providing valuable feedback on the writing of this thesis. Lim Dah-Yoh and Li Ji also provided helpful comments throughout the course of this work.

Next, I would like to express my gratitude to my undergraduate thesis supervisor, Professor Lim Ee-Peng[2], as well as, Professor Yeo Chai-Kiat[3] and Professor Heiko Schroder[4]. Without their encouragement and support, I probably would not have considered graduate education.

Last but not least, I would like to thank my parents for their constant unconditional support. My family had been, and would continue to be my greatest inspiration and motivation.

[1]First woman in the United States to receive a PhD from a Computer Science department.
[2]Nanyang Technological University, Singapore.
[3]Nanyang Technological University, Singapore.
[4]RMIT, Australia.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Persistent object stores provide persistent storage for a large number of objects of varying sizes and user applications execute transactions to modify the states of objects stored in the system. Transactions are serialized: a later transaction will observe the modification of an earlier transaction once it has committed. Atomic transactions [20] allow the system to properly handle concurrency and failures. Consequently, transactional persistent object stores enable safe sharing of objects across space and time and provide a powerful software engineering paradigm where the object model is decoupled from individual programs.

The ability to take snapshots of the persistent object store is useful for archiving data, as well as, for data backup and recovery. A snapshot contains the state of all the objects in the object store at the time the snapshot was taken. Subsequently, users are able to specify when snapshots are taken and execute read-only transactions on these snapshots.

In this thesis, we propose an efficient snapshot scheme for taking transaction-consistent snapshots of the object store in THOR. We describe the semantics of snapshots in the persistent object store and discuss a scalable implementation of our snapshot scheme. Our performance goal is that the snapshot scheme should not compromise the performance of the system, i.e., it should have minimum impact on the performance of concurrent transactions on the current database. We present our experiment results to show that our implementation meets this goal.

1.1 THOR: A Distributed OO Database System

THOR [20, 21] is a distributed object-oriented database system that is built based on the client-server model. Servers run the Object-Repository (OR) part of THOR and form a distributed persistent object store. Every client machine runs the Front-End (FE) process, which is responsible for communicating with the servers and provides efficient cache management of the client-side caches. Figure 1-1 depicts the system architecture of THOR. User applications run at the client machines and interact with THOR via the FE running on the same machine. The FE maintains a cache in which it keeps copies of persistent objects and fetches objects into this cache as needed for applications. When a user application attempts to commit a transaction, the FE communicates with the ORs storing the objects affected by the transaction and the ORs will decide whether the commit is possible. If the transaction commits, all modifications made to the persistent objects in that transaction

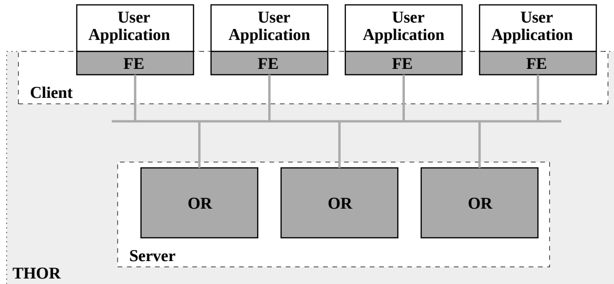


Figure 1-1: Architecture of THOR

becomes permanent; otherwise, if the transaction aborts, none of these modifications is installed in the persistent object store.

1.2 Semantics of Snapshots

Our snapshot scheme provides two basic operations to the users: (1) take a snapshot of the persistent object store and (2) execute a read-only transaction on a previously taken snapshot.

Snapshots in our system are based on time. Users take a snapshot by indicating that the snapshot should happen "now" (at the current time). To run a read-only transaction on a snapshot of the system, the user specifies the time in the past at which the transaction should be executed. The snapshot satisfying this request is the most recent snapshot that is older than the specified time.

Our scheme guarantees the following correctness criteria:

- *Serializability*: Snapshots are serialized with respect to normal user transactions in the system, i.e., a snapshot either reflects all the modifications of a transaction (which may span multiple ORs) or none of its modifications.
- *External Consistency*: Snapshots provide external consistency within the limitations of clock synchronization: a snapshot taken at time T reflects all transactions committed at least ϵ seconds before T and does not reflect any transactions that occurred later than ϵ seconds after T, where ϵ is the maximum clock skew among clocks at the ORs.

External consistency depends on the synchrony among the clocks at the ORs. If the clocks are synchronized, the users will not observe any inconsistent state. For example, if Alice commits a transaction at T, and Bob takes a snapshot at time T+δ (δ is the granularity of time observable by a user) and tells Alice about it, Alice's transaction at time T will be included into the snapshot. Conversely, if the clock skews among the clocks on the servers grow beyond δ, then Alice would notice that her transaction is not included in Bob's snapshot although it occurred before the snapshot was taken.

Current time synchronization technologies like the Network Time Protocol (NTP) [25, 26] allow computers on a network to be synchronized within 50 ms of each other. It is therefore reasonable to assume that the clocks in the ORs are loosely synchronized within ϵ seconds of each other for some small ϵ. The external consistency criterion is a soft constraint: if the clocks in the ORs get out of sync by more that ϵ seconds, snapshots may not be externally consistent. However, serializability of the snapshots is still maintained.

Although THOR provides atomicity guarantee for application transactions, we only provide serializability guarantee for snapshots. This is because providing atomicity to snapshots will require the participation of all the servers for every snapshot and the failure of a single server can result in the abortion of the snapshot.

1.3 Snapshot Archive

Since snapshots are long-lived data, the amount of storage required for archiving them becomes large as more and more snapshots are taken over time. In the long run, the storage required for archiving snapshots dwarfs the storage capacity of the persistent object store. Consequently, snapshots cannot be archived in the same storage medium as the persistent object store itself. In our system, snapshots are archived into a separate snapshot archive storage system (archive store).

Our first archive store design uses a peer-to-peer (p2p) approach to take advantage of the inherent qualities of fault-tolerance, self-organization and configuration, and automatic load-balancing in p2p systems. Furthermore, p2p storage systems [8,34] enable the participating nodes to pool unused storage to provide a large storage area for archiving snapshots. We developed a prototype snapshot archive system on the Chord [37] p2p location and routing infrastructure. Our design is inspired by the Cooperative File System (CFS) [8], which is a file system built on top of Chord.

Using a p2p approach is not without its disadvantages. One disadvantage of the p2p approach is the long delay incurred when locating data. This is mainly due to the multiple hops (approximately $O(\log_n)$ hops, where n is the number of nodes in the p2p system) required to locate the node storing the data. Another drawback of using a p2p approach is that data is not located close to the users.

We also explore two other alternatives in designing the archive store: (1) storing snapshots in a remote storage node on the LAN and (2) storing snapshots locally, on a separate disk sub-system. These approaches provide faster storing and retrieval of data but requires explicit mirroring or replication for fault-tolerance.

1.4 Contributions

Our scheme provides transaction-consistent snapshots of the data store in an environment that supports distributed atomic transactions. Furthermore, our implementation is designed to scale to thousands of servers and does not interfere with concurrent transactions in the system. To our knowledge, the ability to provide transaction-consistent snapshots in a distributed environment, without compromising system performance or interfering with concurrent transactions is unique to our snapshot scheme.

Snapshots or point-in-time copies have been used in many domains such as file systems, storage systems, and databases for several applications such as backup, checkpointing, and testing. EMC's TimeFinder [41] provides a point-in-time copy of the storage subsystem by creating a mirror image of the data before creating the point-in-time copy. A point-in-time copy is subsequently created by "splitting the mirror" at the time of copy. On the other hand, EMC's SnapView [36] and IBM's FlashCopy [23] use copy-on-write to provide incremental backup of data. In these systems, data is stored in one location e.g., one large centralized disk array; these systems are not designed to provide consistent point-in-time copies across distributed data stores.

File system snapshots guarantee close-to-open consistency and have been widely used to provide backup features. A file system snapshot provides a read-only view of the entire file system at a particular instance in time. File system snapshots are often created using block-level copy-on-write. AFS [13], Plan 9 [31], WAFL [12], and Frangipani [40] are examples of file systems that provide a snapshot feature. On the other hand, versioning file systems like Elephant [35] provide a snapshot feature implicitly because they keep all versions of files in the file system. Most snapshot schemes in file systems are designed to work in an environment where data is stored centrally. Applying these snapshot designs directly to a distributed system, such as THOR, requires concurrent update operations to be blocked when taking the snapshot in order to maintain consistency. These operations are only allowed to proceed after all nodes have been informed about the snapshot; this does not meet our requirement of non-interference with concurrent transactions.

Database management systems (DBMS), such as DB2 [27] and Informix Online [14], provide an archive copy utility for backup and recovery. A full archive copy copies all the pages in the database while an incremental archive copy only copies pages that have been modified since the last archive copy operation. When concurrent transactions are allowed to proceed while the archive copy is created, it is called a fuzzy archive copy. Fuzzy archive copies may contain uncommitted data. Like file system snapshots, database archive copy utilities are not directly applicable to THOR's distributed environment without compromising system performance since this will require all concurrent transactions to be blocked during the archive copy operation.

1.5 Thesis Organization

The rest of this thesis is organized as follows. In Chapter 2, we present our definition of snapshots and discuss how snapshot information is propagated. In Chapter 3, we present the architecture of our snapshot subsystem in THOR. We further discuss the design of the archive storage system for archiving snapshots in Chapter 4. After snapshots have been created, users can run read-only transactions on these snapshots. We describe the execution of these read-only transactions, as well as the extension of the client (FE) to support them in Chapter 5. In Chapter 6, we present the results of our experiments to show that our snapshot scheme meets our performance goals. We discuss related work in Chapter 7 and conclude in Chapter 8.

Chapter 2

Defining Snapshots

In this chapter, we describe the semantics of snapshots and how information about snapshots is propagated throughout the system.

2.1 Ordering Snapshots

A snapshot is a transaction-consistent copy of the entire database. Snapshots are transactions and they are serialized relative to other (application) transactions: a snapshot reflects modifications of all application transactions that are serialized before it and does not reflect any modifications of transactions that are serialized after it.

In our system, snapshots are committed at a centralized snapshot coordinator called the OR_{ss}. Information about snapshots is then propagated, in the background, from the OR_{ss} to other nodes (ORs and FEs) in the system. Using a centralized coordinator ensures that snapshots do not abort and are committed immediately.

The OR_{ss} is responsible for keeping track of all snapshots in the system and ensuring that information pertaining to the snapshots is persistent. When a user wishes to take a snapshot, he or she sends a request to the OR_{ss}. Once the OR_{ss} writes the snapshot (and its timestamp) to persistent storage, the snapshot is considered to be committed.

The OR_{ss} is a light-weight process: it is only responsible for recording and time-stamping snapshots and does not store large amount of data. Consequently, we run the OR_{ss} as part of an OR. To improve reliability, we can replicate the OR_{ss}, or even use a Byzantine group [5] for it. Replicating the OR_{ss}, however, is orthogonal to this thesis.

For the approach of using the OR_{ss} to work, we need a way to ensure the serializability of the snapshot transaction relative to application transactions. Our approach takes advantage of the way transactions are serialized in THOR. THOR serializes transactions using timestamps. When a transaction is about to commit, it is given a timestamp by reading the local clock (the identifier of the node assigning the timestamp is attached as the lower-order bits to ensure that the timestamp is unique). The timestamp of a transaction determines it serialization order: the transaction is serialized after all other transactions that have lower timestamps than itself.

After the timestamp is assigned to the transaction, THOR's transaction commit protocol checks whether the transaction can be committed given its timestamp [1]: the transaction must not modify any objects read by a transaction with a larger timestamp and must not read new versions of objects modified by a transaction with a smaller timestamp. If a transaction satisfies these requirements, it is allowed to commit; otherwise, it is either aborted or the commit is attempted with a larger timestamp.

We take advantage of THOR's transaction commit protocol by also assigning timestamps to snapshot transactions. The OR_{ss} assigns the timestamp when it receives a request to take a snapshot; this ensures that snapshot transactions are serialized relative to one another and reflects the order in which they occur in real time.

Information about committed snapshots is then propagated, in the background, to all the other ORs and FEs in the system. This approach is efficient but does not guarantee external consistency: it is possible that a transaction T committed after a snapshot S yet its effects are included in S; it is also possible that T is committed before S yet its effects are excluded from S. Our system guarantees the following:

- A transaction T will be included in snapshot S if it occurs no earlier than ϵ before S was taken.
- A transaction T will be excluded from snapshot S if it occurs later than ϵ after S was taken.

Here ϵ is the bound on the clock synchronization across the system (at all the ORs). Thus there is a window in which external consistency is not guaranteed. With current time synchronization technologies such as NTP, which allows computers on a network to be synchronized within 50 ms of each other, this window is expected to be small.

Since timestamps are internal to the system and not known to the users, the users will only notice external consistency violations if there is some form of communication among them. Communication can either be in-system or external to THOR, e.g., two users communicating via email. In the former case, we can avoid the obvious problem of the same user running a transaction and then taking a snapshot by sending clock values on in-system messages and synchronizing clocks at each step.

Violations of external consistency are thus possible only when communication is external to THOR. For example, a user, say Alice, may take a snapshot at FE_1 and send a message to inform another user, say Bob, who executes a transaction at FE_2, and then uses the snapshot taken by Alice. It is possible that Bob might observe that the snapshot taken by Alice includes the effects of his transaction, although the transaction committed after Bob learns about the snapshot. This situation is highly unlikely if Bob is a person but may be more of a problem if Bob is a program. In the either case, the smaller the ϵ, the less likely that external consistency violations will be observed: violations of external consistency are highly unlikely to be noticed if ϵ is small.

In THOR, transaction timestamps are assigned by FEs. We have changed this in the new system so that transaction snapshots are assigned by ORs instead. This change means that we only need to rely on clock synchronization at the ORs and therefore we can be confident that ϵ is small.

2.2 Propagating Snapshot Information

Snapshots are committed only at the OR_{ss}. After a snapshot committed at the OR_{ss}, other ORs and FEs in the system have to be informed about it. ORs use this snapshot information to create snapshots. FEs, on the other hand, use this information to determine the correct snapshot satisfying a user's request to execute a read-only transaction in the past.

ORs and FEs can obtain updated snapshot information by periodically by querying the OR_{ss} to determine if a new snapshot has been taken. However, if we want the FEs and ORs to have up-to-date information about snapshots, this approach makes the OR_{ss} a hot spot in the system. As an optimization, we use a gossip-based approach to propagate information about snapshots. When a snapshot commits at the OR_{ss}, information about this snapshot, including its timestamp, is propagated from the OR_{ss} to other ORs and FEs. This is done by piggybacking the snapshot information on other messages between the nodes.

In our system, snapshot information about recent snapshots is sent out periodically from the OR_{ss}. If the OR_{ss} does not communicate with any other node within this time period, it will randomly select a few ORs and send them the updated snapshot information. Snapshot information is then disseminated throughout the entire system via gossip, i.e., it is piggybacked on other messages between nodes in the system.

To quickly and efficiently propagate snapshots, we can superimpose a distribution tree on the system. For example, the OR_{ss} can disseminate snapshot information to K other ORs, who will, in turn, pass this information to K other nodes in the system. The time taken to disseminate the information throughout the entire system is hence $O(\log_K N)$, where N is the number of nodes in the system. For example, if we choose K to be 10 and if there are 10,000 nodes in the system, it will take 4 rounds to disseminate the message to all the nodes in the system; this typically occurs within a matter of seconds. Furthermore, since snapshot messages are small and the snapshot information that each node has is fairly up-to-date, snapshot information can be propagated with little overhead.

Nodes can always fall back on querying the OR_{ss} if snapshot information is not propagated fast enough. Even if the OR_{ss} is unreachable for a short period of time, our design ensures that the nodes can still function correctly. The design of the system is presented in Chapter 3.

2.3 Snapshot Message

The snapshot history of the system is the information about all the snapshots in the system and grows with time. One way of propagating snapshot information is to send the entire snapshot history on each snapshot message. However, this is inefficient because the snapshot messages will be large and the recipient of a snapshot message will already have information about most of the snapshots in the snapshot history. Therefore, the snapshot message only needs to contain information about recent snapshots. Since snapshots can be represented by their timestamps, the snapshot message will thus contain a list of snapshot timestamps of the most recent snapshots, e.g., snapshots that were taken in the past 10 minutes.

The OR_{ss} sends its current timestamp, TS_{curr}, in each snapshot message. This information enables ORs to know how up-to-date they are with respect to snapshots and this in turn allows them to manage snapshots efficiently. Without this information, ORs would only know that snapshots did not occur between two snapshots S_1 and S_2 when S_2 happens, and in the meantime they would have to retain information about pre-images of modified objects just in case a snapshot had happened that they had not yet learned about.

Since the gossip protocol neither guarantees that all snapshot messages will reach a node nor that these messages will be delivered in order, it is possible that a node may miss a snapshot message. To allow the recipient nodes to detect a missing snapshot message, each message also contains the timestamp, TS_{prev}, which acts as a lower bound on information in the message. Thus each snapshot message contains a portion of snapshot history: it contains the timestamps of all snapshots that happened after TS_{prev} and before TS_{curr}.

A node will accept a snapshot message if its TS_{prev} is less than or equal to the TS_{curr} of the previous snapshot message that it accepted. TS_{prev} is defined such that each node will have a high probability of accepting a snapshot message (i.e., it is very likely that the nodes' snapshot information is more recent that TS_{prev}).

For example, when a node communicates frequently with another node, it can set TS_{prev} based on what it heard from that other node in its last communication. When a node communicates with a node that it hasn't heard from in a long time, it sets TS_{prev} to a time that is highly likely to be old enough, e.g., to the current time minus one hour.

In the unlikely event that TS_{prev} of the message is greater than the TS_{curr} of the previous message, the node knows that snapshot information between these two timestamps is missing. To update its snapshot information, it sends an update request, containing TS_{prev} of the new snapshot message and TS_{curr} of the previous snapshot message, to the sender node. On receiving an update request, the sender node replies with its own snapshot information between the two timestamps if its snapshot information is at least as recent as TS_{prev}. In the event that the sender nodes fails after sending out the first snapshot message, updated snapshot information can still be obtained from the OR_{ss}.

For instance, a node X receives a snapshot message M_1 with TS_{prev} of 100 from node Y. In addition, the previous snapshot message received by X has a TS_{curr} 87. In this case, X detects that it is missing snapshot information between 87 and 100 and sends an update request for this information to Y. Y, which must have updated snapshot information at least up to 100, will send its snapshot information from 87 to 100 back to X in message M_2. This enables X to update its snapshot information using M_2 and then M_1.

The format of a snapshot message is shown in the Figure 2-1. As shown in the figure, the list of snapshot timestamps in a snapshot message can be empty; a snapshot message with no snapshot timestamps and containing only TS_{curr} and TS_{prev} is used to inform nodes in the system that there are no snapshots between TS_{prev} and TS_{curr}. Even if there are no new snapshots in the system, the OR_{ss} will periodically send out snapshot messages with

$$\mathrm{TS_{prev}}, \overbrace{\mathrm{TS_{S1}}, \mathrm{TS_{S2}}, \ldots, \mathrm{TS_{Sn}}}^{\text{optional}}, \mathrm{TS_{curr}}$$

Figure 2-1: A Snapshot Message

an empty snapshot list.

We expect that in most cases, the snapshot messages will only contain $\mathrm{TS_{prev}}$ and $\mathrm{TS_{curr}}$ because taking a snapshot is a relatively rare event. If snapshots are taken after the last snapshot message was sent out, the next snapshot message will contain the timestamps of these snapshots. Therefore, unless there is a sudden flood of snapshots within a very short time period (i.e., between two snapshot message), snapshot messages will be small.

Snapshot messages are also designed to be general enough to allow nodes to tailor the information they send to other nodes when propagating snapshot information. This ability enables each node to minimize the size of snapshot messages. Suppose that nodes A and B have complete snapshot information up to times $\mathrm{T_A}$ and $\mathrm{T_B}$ respectively, where $\mathrm{T_A} > \mathrm{T_B}$. When node A wants to send snapshot information to node B, it only needs to send the information between $\mathrm{T_A}$ and $\mathrm{T_B}$ instead of its entire snapshot history.

2.3.1 Updating Snapshot Information Using Snapshot Messages

An OR uses snapshot messages to update its snapshot history. It maintains a timestamp T_{gmax} that represents the latest time that the OR knows of in the snapshot history. Our protocol guarantees that a node's knowledge of snapshot history is always complete up to $\mathrm{T_{gmax}}$. A node can therefore assume complete knowledge of snapshots up to this time; it must also assume that there might be further snapshots after this time. Since the $\mathrm{OR_{ss}}$ sends out snapshot messages periodically, the $\mathrm{T_{gmax}}$ of each node will be fairly close to the current time (e.g., with a small delta of approximately one minute). When a node receives a new snapshot message, it appends the timestamps of the list of snapshots in the message into its snapshot history and sets its $\mathrm{T_{gmax}}$ to $\mathrm{TS_{curr}}$.

If $\mathrm{TS_{prev}}$ is small enough, each node will have a high probability of accepting snapshot messages that they receive. This will enable the ORs to keep their $\mathrm{T_{gmax}}$ close to the present, i.e., keep their snapshot history up-to-date. When a node's $\mathrm{T_{gmax}}$ falls behind the $\mathrm{TS_{prev}}$, our protocol enables the node to update its $\mathrm{T_{gmax}}$ from other nodes. If a redundant snapshot message is received i.e., $\mathrm{TS_{curr}}$ is smaller than a node's $\mathrm{T_{gmax}}$, this message is discarded since the node's knowledge of snapshot information is more recent than that contained in the message.

Chapter 3

System Architecture

In this chapter, we describe the system architecture of our snapshot subsystem. We provide details about the system design, data structures, and algorithms used in the snapshot subsystem.

3.1 Overview

As described in Chapter 2, information about snapshots is propagated lazily throughout the system. Therefore, when a transaction commits at an OR, it is almost certain that the OR does not know whether the old states of modified objects are required for a snapshot. To ensure correctness of snapshots, the OR must preserve the old states of these modified objects before the modifications are applied to their respective pages on disk.

A simple approach is for ORs to create *opportunistic snapshot pages* using copy-on-write, i.e., duplicate every page (and store the copy on disk) before applying modifications to it. However, this approach is inefficient in both space and time because a transaction typically only modifies a few object on each affected page. Our approach to solving this problem is to preserve the old states of the modified objects instead of their entire containing pages. We call these objects that contain the old states the *pre-images* of the transaction that modified the objects. Pre-images of a transaction must be forced to disk before its modifications can be applied to the pages, so that no information is lost in case the OR crashes immediately after applying the modification.

Since snapshots occur infrequently as compared to application transactions, most pre-images are not required for creating snapshots. Furthermore, storing pre-images on disk is expensive. Consequently, we would like to avoid creating pre-images unless we are certain that they are required for creating snapshots. One way of achieving this is to apply modifications of committed transactions to pages lazily, so that by the time the modifications are applied to the pages, the OR would have obtained updated snapshot information, which would enable it to determine whether the pre-images are required for a snapshot.

Implementing the snapshot subsystem does not require substantial change in the way that THOR behaves because THOR already tracks modifications at the object-level and writes them back to disk lazily. We take advantage of the existing architecture of THOR to provide an efficient implementation of our snapshot scheme. In the next section, we provide essential background information about THOR, i.e., details about the existing architecture that supports our snapshot subsystem. We then present the design of our snapshot subsystem, which is built on top of this architecture.

3.2 Background: The THOR Object Repository

THOR is distributed object-oriented database system that is developed based on the client-server model. The FE fetches data from the ORs in fixed-size pages and ships only modified objects back to the OR during a transaction commit. Object shipping reduces the size of commit requests substantially.

3.2.1 Data Organization

In THOR, objects are uniquely identified in an OR with 32-bit *object references* (*orefs*). The combination of the 32-bit *OR number* (*ORnum*) of the object's OR and its oref is called the *xref* of the object. The xref uniquely identifies the object in the entire system.

Each oref is 32 bits long and is composed of a 22-bit *PageID* and a 9-bit *oid*. The PageID identifies the object's page and allows the location of the page (either on the OR's disk or in its page cache) to be determined quickly. The oid is an index into a offset table that is stored in the page; the offset table contains the 16-bit offsets of objects within that page. This extra level of indirection allows an OR to compact its pages independently from other ORs and FEs.

3.2.2 Modified Object Buffer (MOB)

To reduce the write performance degradation due to object shipping, the OR uses an in-memory *Modified Object Buffer* (MOB) [10] to store objects modified by a transaction when it commits, rather than writing these objects to disk immediately. The MOB 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. This reduces disk activity and improves system performance. Furthermore, disk writes can now be done in the background and deferred to a convenient time. 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. In addition, since THOR uses asynchronous disk I/O, inserting modifications into the MOB is done in parallel with logging them in the transaction log. Therefore, using the MOB does not incur any additional overhead.

The structure of the MOB is shown in Figure 3-1. The MOB contains a directory called the *iTable* and a storage area in the OR's heap. The iTable is organized into an array of hash tables, one for each page with modifications in the MOB. Each hash table maps orefs of objects that are contained in the page to object wrappers for these objects in the MOB. An object wrapper, in turn, contains bookkeeping information about a modification and a pointer to an object in the storage area that represents the new state of the object due to the modification. A modification is applied to the page by overwriting the old object on

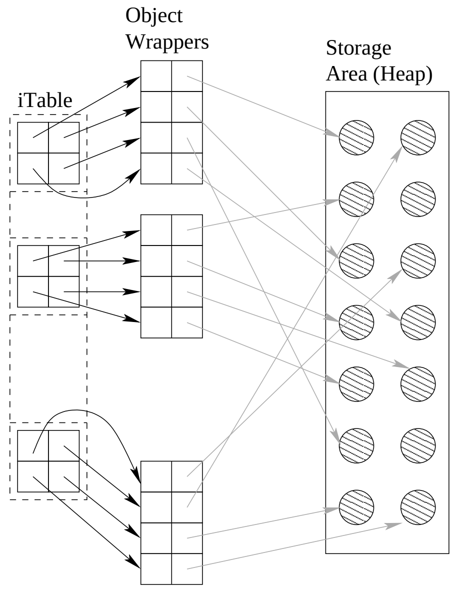


Figure 3-1: Structure of the MOB

the page with the corresponding new object in the MOB.

Entries are inserted into the MOB when a transaction commits; they are removed when the corresponding modifications are installed in their respective pages and the pages are written back to disk. In addition, when an entry is inserted into the MOB and there is already an earlier entry that corresponds to a modification to the same object, the earlier entry is discarded because the later entry will contain the most up-to-date state for that object. The earlier modification is said to have been overwritten by the later one.

Modifications are installed in a page by (1) reading the page from disk to the page cache in memory, (2) installing the modifications by overwriting the old objects on the page with objects in the MOB, and (3) writing the updated page back to disk. The process of reading a page into the page cache prior to installing its modifications is called an *installation read*. The OR uses on a background *flusher* thread to clean the MOB, i.e., to install modifications from the MOB to the page so that their corresponding MOB entries can be discarded. The MOB is cleaned when it fills beyond a pre-determined *high watermark*. The cleaning process is done in log order to facilitate the truncation of the transaction log.

3.2.3 Handling Page Fetch Requests from the FE

When a FE fetches a page from the OR, the page on disk will need to be updated with the latest modifications (that have accumulated in the MOB) before returning it to the FE. This is ensures that pages fetched from the ORs always reflect all committed transactions.

To update the page with its corresponding modifications in the MOB, the OR first performs an installation read to bring the page from disk to the page cache. All modifications in the MOB for that page are then installed in the page before returning it to the FE. If a page in

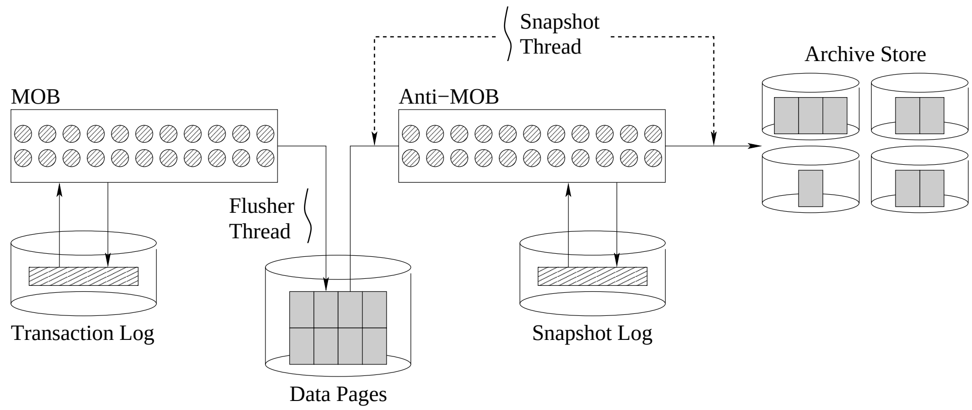


Figure 3-2: Architecture of the Snapshot Subsystem in an OR

the page cache is requested again, the OR checks whether all modifications in the MOB have been installed in the page, and reinstalls all the modifications in the MOB for that page if the page is not up-to-date. An up-to-date page in the page cache becomes "outdated" if a new transaction that modifies this page commits at the OR. The page cache is managed in an LRU fashion and pages evicted from the cache are not written back to disk. Instead, the OR relies only on the flusher thread to install modifications from the MOB to the pages on disk.

3.3 Design of the Snapshot Subsystem

Figure 3-2 shows the overall architecture of the snapshot subsystem in an OR; this snapshot subsystem works in tandem with the OR's MOB and transaction log. The OR's flusher thread is a background thread that is used to apply modifications in the MOB to the disk and the transaction log ensures correctness after the OR recovers from a failure. The MOB, transaction log, and flusher thread are components of the original THOR system.

In the snapshot subsystem, we use an additional *snapshot thread* to create and archive snapshot pages in the background. Snapshot pages are created by reverting modifications made to the current pages since the time the snapshot was taken. The in-memory *Anti-MOB* buffer stores the pre-images of transactions that are necessary for reverting the modifications. We will discuss the Anti-MOB in more detail in Section 3.3.2. Snapshot pages are archived into an archive store after they have been created. We present the archive store design in Chapter 4. The snapshot subsystem uses a persistent *snapshot log* to ensure that the OR is able to recover its Anti-MOB after a failure. Failure and recovery are discussed in Section 3.4.

3.3.1 Creating Snapshot Pages

Snapshot pages are created copy-on-write: a snapshot copy of a page is created when the page is modified for the first time after the snapshot. In particular, we create snapshot copies only when pages on the OR's disk are modified for the first time since the snapshot and just before the modification takes place.

In our system, snapshot pages are created as part of the MOB cleaning process to take advantage of the installation read: we avoid having to read the page from disk to memory because the installation read performed during the MOB cleaning process would have already brought the page into memory. Furthermore, pages on disk are only modified during the MOB cleaning process; this fits into our copy-on-write scheme of creating snapshot copies of pages only when the page on disk is modified.

To know whether a snapshot copy of a page is required for a snapshot, we need to know whether the page is being modified for the first time after the snapshot. To keep track of this, every OR maintains a *snapshot page map*, which has an entry for every page in the system. Each snapshot page map entry is a timestamp. Each time we create a snapshot copy of a page, we update its corresponding snapshot page map entry with the timestamp of the snapshot. All snapshot page map entries are initially zero.

Snapshot copies of a page are always created in timestamp order, i.e., the snapshot copy of a page P is for snapshot S_1 is always created before the snapshot copy of P for snapshot S_2 if S_1 occurred before S_2. Therefore, when the snapshot page map entry for a page P has a timestamp greater than a snapshot S's timestamp, the OR must have already created a snapshot copy of P for S.

3.3.2 The Anti-MOB

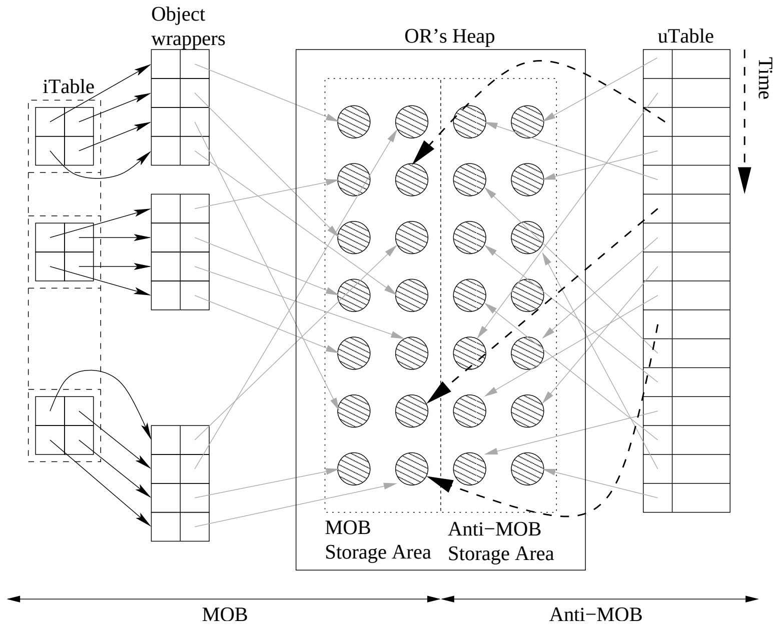
The approach of creating snapshot copies of entire pages does not take into account the following:

- A page in the page cache may already reflect modifications in the MOB, including those for transactions later than the snapshot. However, we do not want to read the page from the disk again when creating snapshot pages. Therefore, we need a way to revert modifications.

- If a transaction modifies objects that already have entries in the MOB (due to older transactions), the old entries will be overwritten by new entries corresponding to the new transaction. However, we may need these overwritten entries for creating snapshot copies of pages. Hence, we need a place to store them.

- If the MOB is processed to the end, we will encounter modifications that belong to transactions committed after T_{gmax}; this is inevitable since ORs are always lagging in their snapshot information, which is propagated via gossip. As a result, we need to preserve the pre-images of transactions committed after T_{gmax} because they may be needed for creating snapshot pages: the OR cannot ascertain, at the time the MOB is processed, whether these pre-images are required since its snapshot information is older than the transactions.

We use an in-memory buffer called the Anti-MOB for storing overwritten MOB entries and transaction pre-images, and for creating snapshot copies of pages. The structure of the Anti-MOB is shown in Figure 3-3. As shown in the figure, each Anti-MOB entry contains a pointer to the pre-image in the OR's heap and the timestamp of the pre-image's transaction.



The Anti-MOB is similar to the MOB. It is designed to reuse the memory space of overwritten MOB entries to avoid creating copies of these entries.

Figure 3-3: Structure of the Anti-MOB

The Anti-MOB and the MOB are similar; the main difference between them is that the Anti-MOB contains the pre-images of update transactions while the MOB contains their post-images.

Creating Anti-MOB Entries

When installing a modification from the MOB to its page in the page cache, the OR first reads the corresponding "old version" of the object being modified from the page and stores it into the Anti-MOB as the pre-image of the transaction before installing the modification to the page.

However, pre-images are written to the Anti-MOB only if they are needed for a snapshot or might be needed. The latter case occurs when the OR processes the MOB entries for transactions with timestamp greater than T_{gmax}. Normally, the OR only cleans entries from the MOB if they were written by transactions with timestamps less than or equal to T_{gmax}. The OR will process transactions beyond T_{gmax} only when T_{gmax} is older than its current time by some delta, e.g., two minutes. An old T_{gmax} rarely occurs; it usually occurs only when there is a problem in getting recent information from the OR_{ss}.

In addition, entries are made in the Anti-MOB due to overwriting and to page fetches.

Anti-MOB entries are needed when there are overwrites to preserve intermediate values that may be needed for snapshots. For example, suppose transactions A and B committed at times T_1 and T_2 ($T_1 < T_2$), and both transactions modify object X. If a snapshot S is taken between T_1 and T_2, the state of X required for S is the state of X after applying the modification by transaction A and before applying that by transaction B. If the MOB entry corresponding to the modification to X by transaction A is overwritten by that of transaction B, the state of X after transaction A will be lost unless it is preserved. We use the Anti-MOB for this purpose.

An object corresponding to the overwritten MOB entry is stored into the Anti-MOB as the pre-image of the transaction that caused this entry to be overwritten. Since the object is already in the MOB, we swing the Anti-MOB entry to point to the object in the MOB's storage area instead of creating a new object in the Anti-MOB's own storage area. This is illustrated in Figure 3-3.

In addition, the first time an object is overwritten, we also create an 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. Then the system stores $< T_2,\ X,\ X_1 >$ to indicate that X_1 (the value of X created by T_1) is the pre-image of X for T_2. In addition, if T_1 was the first modification to X 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 system also adds $< T_1,\ X,\ \text{null} >$ to the Anti-MOB, indicating that for s, the proper value of X is the one currently stored on disk.

MOB cleaning takes these special entries into account. When the system encounters a MOB entry $< T,\ X >$ as part of cleaning the MOB and assuming the pre-image of X is needed for a snapshot, it writes X's current value in the heap. If there is no entry in the Anti-MOB for T and X, one is created; this entry points to X's value in the heap. Otherwise the system finds the Anti-MOB entry for X containing null and modifies that entry so that it now points to the copy of X's current value in the heap.

Anti-MOB entries are also created when pages are fetched into the page cache. As we have discussed, these pages are updated by applying all modifications in the MOB to them before returning them to the FE that requested them. If the page is in the page cache when the MOB is cleaned, the cache copy is used for the cleaning. But we may need to revert that copy if some snapshot requires an earlier state. Therefore, as modifications are applied to the page, pre-images of those modifications are stored in the Anti-MOB; this is similar to what happens when the MOB is cleaned. Pre-images are created only if needed, i.e., if some snapshot may require the pre-image.

Creating Snapshot Pages Using the Anti-MOB

Snapshot pages are created by first processing all MOB entries to bring the page up-to-date; this causes entries to be made in the Anti-MOB (to capture pre-images) if needed. Then the OR makes a copy of the page and uses the Anti-MOB to revert the page to the proper state for the snapshot. Processing to create snapshot pages is done only when needed. For example, if the OR is processing a modification for transaction T_1 and the page already has a snapshot page for the most recent snapshot before T_1, no entry is added to the Anti-MOB and no reverting is done.

Algorithm: `create_snapshot(i, t)`

Let $\mathrm{SCAN}_{(\mathrm{i,t})}$ = set of objects scanned for page `i` for snapshot `t` and initialized to $\emptyset$
for each entry $\mathtt{U_i}$ in the Anti-MOB for a transaction with timestamp $\geq$ `t`
    if $\mathtt{U_i.oref} \notin \mathrm{SCAN}_{(\mathrm{i,t})}$ **then**
        $\mathrm{SCAN}_{(\mathrm{i,t})} = \mathrm{SCAN}_{(\mathrm{i,t})} \cup \{\mathtt{U_i.oref}\}$
        install $\mathtt{U_i}$ into page `i`
    endif
end

Figure 3-4: Algorithm for Creating Snapshot Pages from the Anti-MOB

The reason for installing all modifications to a page and then reverting some of these modifications to create a snapshot copy is that this approach is required for pages that are already in the buffer cache. Therefore, we chose to use a uniform approach that works in this case, where the state of the page in the buffer cache may be different from that in the disk, as well as, in the case when both copies have the same state, i.e., when the page is read from disk for cleaning the MOB.

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 Anti-MOB can contain several pre-images corresponding to a single object. When creating a snapshot page for a snapshot, the OR only installs the pre-image of the oldest transaction that occurred after the snapshot because only this pre-image will contain the correct state of the object for the snapshot. For example, assume that the Anti-MOB contains three pre-images I_1, I_2, and I_3 that corresponds to object X. Further suppose that I_1, I_2, and I_3 are pre-images of transactions T_1, T_2, and T_3 respectively, where $T_1.t < T_2.t < T_3.t$. When creating a snapshot page S for a snapshot with timestamp S.t, where $S.t < T_1.t$, only I_1 will contain the correct state of X for that snapshot, since I_1 contains the state of X when it was first modified (by T_1) after the snapshot was taken.

The algorithm used for creating snapshot pages using the Anti-MOB is shown in Figure 3-4. In the figure, `t` is the timestamp of the snapshot and `i` is the page being processed. In this algorithm, the portion of the Anti-MOB later than `t` is scanned as described above. The SCAN set keeps track of pre-images scanned and ensures that only the first pre-image corresponding to an object is used in creating the snapshot page. Note that the algorithm assumes that the Anti-MOB is scanned in timestamp order.

3.3.3 Removing Entries From the Anti-MOB

Entries can be discarded from the Anti-MOB as soon as they are no longer needed for making snapshot pages. This happens when (1) the MOB is cleaned, (2) new snapshot information is received by the OR (i.e., T_{gmax} is updated), or (3) the Anti-MOB fills up beyond a pre-defined threshold.

When cleaning the MOB for a particular page P, snapshot pages will be made for P if necessary. After processing of P is complete, entries for P can be removed from the Anti-MOB. Such entries are removed if they will not be needed for any other snapshot pages for P. This condition is true if the timestamp in the entry 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's T_{gmax} is updated. This will allow the OR to 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).

When the Anti-MOB fills beyond a pre-defined threshold, e.g., 90% filled, the OR can spontaneously clean the Anti-MOB. This process is similar to the process of creating snapshot pages as part of the cleaning process. However, in this case, additional installation reads may have to be performed to create the snapshot pages.

3.4 Persistence of OR Snapshot Information

An OR can fail, e.g., due to a power failure, but the information that it stores must be recoverable after it recovers. THOR's persistent transaction log ensures that committed transactions are recoverable. In this section, we present the portion of the design of the snapshot subsystem that ensures snapshots are also recoverable after a failure.

3.4.1 The Snapshot Log

We use a persistent (on disk) snapshot log to allow the OR to recover snapshot information after a failure. The snapshot log is similar to the transaction log. In fact, we could have used the transaction log to make snapshot information persistent. However, in the current implementation of THOR, the MOB is cleaned in timestamp order to facilitate the truncation of the transaction log. Putting the snapshot log into the transaction log can prevent the OR from truncating the transaction log if T_{gmax} stops getting updated; when this happens, the "old" log entries corresponding to Anti-MOB entries cannot be discarded, preventing the OR from truncating the transaction log.

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 written back to disk as part of the MOB cleaning process, the corresponding snapshot log entries containing the pre-images of modifications to the page are flushed to disk. The snapshot page can then be written to the archive asynchronously. Forcing the snapshot log ensures that the necessary pre-images are available so that the snapshot page can be recreated should a failure occur after the page is written to disk but before the snapshot page is archived.

It is possible that more than one snapshot page will be created for page P as part of cleaning the MOB. In this case, the way to recover the snapshot pages for P from the snapshot log is to work backward. 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}.

Entries can be removed from the snapshot log once the associated snapshot pages have been saved to the archive since they are not needed for later snapshots: an entry $< T,\ X,\ X_1 >$ (where X_1 is the pre-image of X) can only be removed when a snapshot page for X's page P exists for the latest snapshot with timestamp less than T. This condition is the same as the one used for discarding entries from the Anti-MOB.

3.4.2 Recovery of Snapshot Information

When the OR recovers from a failure, it proceeds as follows:

1. It recovers its snapshot history by communicating with another OR.
2. It initializes the MOB and Anti-MOB to empty and it initializes the snapshot page map to have all entries "undefined".
3. 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).
4. It reads the snapshot log and adds its entries to the Anti-MOB.
5. At this point it can resume normal processing.

When the OR cleans the MOB, it needs to find out the proper values for the snapshot page map. This information is stored in the archive as explained in Chapter 4. The OR can obtain this information from the archive in the background, in advance of when it is needed. For example, as it adds an entry to the MOB for a page whose entry in the snapshot page map is undefined, it can retrieve the snapshot information for the page from the archive. This can be done asynchronously and by the time the information is needed for cleaning the MOB, it is highly likely that the OR will know it.

The OR also needs the snapshot page map information to process fetches of snapshot pages; this issue is discussed in Chapter 5.

3.4.3 Anti-MOB Overflow

We mentioned that the OR can spontaneously clean the Anti-MOB when it becomes filled beyond a threshold. However, this is not always possible. The process of cleaning the Anti-MOB is dependent on T_{gmax}, which is updated via gossip and by communicating directly with the OR_{ss}. In the event that a failure renders the OR_{ss} unreachable, the Anti-MOB can overflow because the OR will no longer be able to clean its Anti-MOB. We push the current contents of the Anti-MOB into secondary storage in the event of an overflow. This frees up the Anti-MOB's memory and allows the OR to continue inserting new entries into the Anti-MOB.

We take advantage of the snapshot log to provide persistence for the Anti-MOB. However, in the case of Anti-MOB overflow, we must write the entire Anti-MOB to the snapshot log. This writing causes entries that might be discarded without ever becoming persistent (e.g., entries due to overwriting objects in the MOB where the pre-states are not needed for a snapshot) to be placed in the snapshot log. Once the contents of the Anti-MOB have been stored in the snapshot log, the OR discards the current contents of the Anti-MOB.

When the OR finally gets updated snapshot information and advances its T_{gmax} (i.e., it can further process the Anti-MOB), it recovers the discarded portion of the Anti-MOB from the snapshot log. To further process the Anti-MOB when its T_{gmax} is advanced, the OR swaps the current contents of the Anti-MOB for the earliest section of the Anti-MOB (recovered from the snapshot log). The OR then creates snapshot pages by progressively processing sections of the Anti-MOB by reading them from the disk.

It is possible that the portion of the Anti-MOB read into memory is insufficient to create a snapshot page, e.g., the pre-images required for creating a snapshot page may span more than one section of the Anti-MOB read into memory. We process each page by installing as many pre-images as possible, using the section of the Anti-MOB that is currently in memory. This creates *partial snapshot pages*. The OR may write these pages to disk if there is insufficient space to accommodate them in primary memory. When processing the next section of the Anti-MOB, the OR will use the partial snapshot page of a page being processed if it exists; otherwise, the OR will use the current page.

In addition, the OR keeps track of pre-images that have already been installed in a partial snapshot page by remembering the SCAN table (Figure 3-4) used for creating it. Suppose that the OR writes a partial snapshot page, $P_{S'}$, to disk, and it had already installed a pre-image of X in $P_{S'}$. When the OR processes $P_{S'}$ again, it may encounter another pre-image of X in the Anti-MOB. The SCAN table for $P_{S'}$ enables to the OR to know that it had already installed a previous pre-image of X and allows it to ignore this new pre-image.

Snapshot pages that are created are written to the archive store and partial snapshot pages are discarded once the Anti-MOB processing is completed.

Chapter 4

Archiving Snapshots

This chapter describes the archiving of snapshot pages. Our main goal of the archival storage is fault-tolerance: since snapshot pages are stored only in the archive store, we cannot tolerate their loss due to failures. Consequently, we require high reliability and availability for the archive store: the archive store should be as reliable and available as the ORs themselves. In other words, if we can access the current page at an OR, we should be able to access any of its snapshot pages.

The performance goal of our system is to minimize the cost of snapshots when executing transactions on the current database. Transactions on the current database cause snapshot pages to be archived and we want the archiving of snapshot pages to not degrade the overall performance of the ORs. In our system, snapshot pages are archived in the background as part of the MOB cleaning process. Therefore, the speed of writing snapshot pages to the archive store is unlikely to have a direct effect on the performance of the ORs. On the other hand, there will be a performance penalty when running transactions on snapshots, i.e., using pages from the archive store. Since these transactions in the "past" are expected to occur relatively infrequently as compared to transactions in the "present", optimizing the performance of transactions on snapshots is not a primary performance goal of our system.

4.1 The Archive Subsystem

We provide for archival storage by means of an archive store abstraction. This abstraction is implemented by code that runs at each OR. Its interface is shown in Figure 4.1.

The OR uses the `put` method to store a snapshot page. Page_{ss} is the value of the snapshot page. The ORnum and PageID identify the page for which Page_{ss} is the snapshot value. Each snapshot page is created to retain information needed for a particular snapshot; TS_{ss}, the timestamp of that snapshot, is used to identify that snapshot page from among all the snapshot pages of that page. The `put` method completes only after the new snapshot page has been written to disk.

The OR uses the `get` method to fetch a snapshot page. The ORnum and PageID arguments identify the page, and TS_{ss} identifies the snapshot for which a value is required. If there is no snapshot copy of that page with timestamp greater than or equal to TS_{ss} (because no snapshot page had been created for that page at TS_{ss}), null is returned.

Operation	Description
`put`(ORnum, PageID, TS_{ss}, $Page_{ss}$)	Stores the snapshot page $Page_{ss}$ into the archive store using the ORnum and PageID to identify the page, and TS_{ss} to identify the snapshot.
`get`(ORnum, PageID, TS_{ss})	Retrieves the snapshot page satisfying the given ORnum, PageID and TS_{ss} from the archive store. Returns null if there is no snapshot page satisfying this request.
`getTS`(ORnum, PageID)	Retrieves the latest timestamp for the page identified by the ORnum and PageID. Returns 0 if no snapshot copy for this page exists.

Table 4.1: The Archive Store Interface

Snapshot pages are created copy-on-write. This means that if a page P has not been modified since a snapshot, say S_1, was taken, then no snapshot page of P for S_1 will be created. However, the user can still request the snapshot page of P for S_1. In this case, the snapshot page of P for the oldest snapshot that is more recent than S_1 will contain the state of P at the time S_1 was taken and hence will satisfy this fetch request.

More generally, a fetch request for the snapshot page of page P at time TS_{ss} will be satisfied by the oldest snapshot page of P whose snapshot's timestamp is greater than or equal to TS_{ss}. Thus the value returned by the `get` method is that of the oldest snapshot for that page whose timestamp is greater than or equal to TS_{ss}.

When the OR recovers from a failure, it needs to learn the timestamp of the most recent snapshot page of each page in order to initialize the snapshot page map. It can obtain this information by calling the `getTS` method.

4.1.1 Archive Store Implementations

There are several approaches to implementing the archive store.

In a system where ORs are replicated, it is possible to archive the snapshot pages locally, on a separate disk while still maintaining the degree of availability and reliability that we require. Advancements in magnetic storage technologies suggest that it may be possible to store a terabyte of data on a single disk in the near future. Therefore, even though the storage requirement of archiving snapshots dwarfs the storage requirement for data storage in an OR, it is still conceivable that each OR would archive its snapshot pages on separate magnetic disks on the same machine. Storing snapshot pages locally has the advantage of fast storage and retrieval times for snapshot pages. However, this approach is not flexible: ORs cannot share their excess storage for snapshot pages.

In a network storage environment, storage for the archive store can provided by a remote storage node on the local area network. If ORs are spread across a wide area, there can be several of these storage nodes and ORs that are close to each other (e.g., ORs that are on the same LAN) can share the storage node that is closest to them. Venti [32] is an example of such an archival system. This approach also provides fast storage and retrieval

Storage System	Description	Examples
p2p storage	A p2p storage system is shared by all ORs and can be used either in the local area or the wide area. The storage system provides automatic load-balancing, self-configuration, and resilience to failures. Storing and retrieving data can be slow since it requires multiple hops to find the storage nodes and the required data can be far away. Recent work suggests that the problem of slow lookups in p2p storage systems can be solved by storing complete information about all p2p nodes in the system [11] and that the problem of data being far away can be mitigated by taking advantage of replication [33].	CFS [8] and PAST [34]
Network storage	A network storage system can either have a single storage node or multiple storage nodes. Such a system will have at least one storage node per LAN and ORs in the same LAN (or nearby network) will share the storage node in the local area. Replicating the storage nodes provides fault tolerance. Storing and retrieving data requires only communication in the local area.	Venti [32]
Magnetic storage	Each OR has a separate disk for archiving snapshots. Fault tolerance is provided if ORs are replicated. Retrieving and storing snapshot pages can be as fast as storing and retrieving normal data pages.	Elephant [35]

Table 4.2: Classes of Storage Systems for the Archive Store

of snapshot pages. However, the storage nodes need to be replicated to meet our reliability and availability requirements. In fact, we require that the ORs and storage nodes have the same amount of replication.

The archive store can also be implemented using a p2p approach: snapshot pages are stored on a separate p2p storage network. Using a p2p approach allows us to take advantage of the self-organizing, automatic load-balancing, and fault resilient qualities of p2p systems. In addition, p2p storage systems enable the pooling of unused storage in the nodes to form a large storage area. This large storage area is required for the potentially large storage requirement of archiving snapshots. One drawback of p2p systems is the long delay incurred when locating data. This is mainly due to the multiple hops required to find the node storing the data. A second disadvantage of using a p2p storage system is that the data may be located far away from the users. Recent work [11, 33] shows that these disadvantages can be mitigated: we can avoid lookups by remembering where the data is stored and we can obtain fast access by using replication.

What we have just described are three different storage designs for the snapshot archive store. This discussion is summarized in Table 4.2.

4.2 p2p Archive Store

In this section, we present the design of our p2p archive store for archiving snapshots. The p2p archive store was developed based on the Chord [37] p2p location and routing infrastructure and the design of the archive store is inspired by the CFS's DHash layer [8].

We assume that the storage nodes in the archive store are specialized storage nodes, which are relatively stable and do not leave and join the system frequently. Even so, the self-organizing quality of the p2p approach enables new storage nodes to be added and faulty ones to be removed without the need for administrator intervention.

4.2.1 Background: Chord And DHash

Chord is a p2p routing and location infrastructure that provides one main service to the user: given a key, it maps the key to a node in the p2p system. The node responsible for the key is called the *successor node* of the key. Chord uses consistent hashing [16, 18] for load-balancing and for reducing the movement of keys when nodes join and leave the system. Each node is identified by a 160-bit *ChordID*, which is the SHA-1 hash[1] of the node's IP address. Physical nodes with large capacities act as several virtual nodes in DHash; the ChordID of each virtual node is derived from the SHA-1 hash of the physical node's IP address and its virtual node index.

DHash provides a distributed hash table abstraction for storing fixed-size data blocks among the p2p nodes in the system. There are two types of blocks in DHash, namely root blocks and data blocks. A root block's ChordID is its owner's public key while a data block's ChordID is the SHA-1 hash of its contents. The p2p node responsible for storing this block is the node in the Chord ring whose ChordID immediately succeeds the block's ChordID.

4.2.2 Archive Store Design

As in DHash, each snapshot page is stored in the node that succeeds the snapshot page's ID, i.e., the node with the smallest ChordID that is larger than the snapshot page's ID. We compute the ID of a snapshot page using its ORnum and PageID, as follows:

$$\text{ID(Page)} = \text{SHA-1(ORnum} \circ \text{PageID)}$$

where $\circ$ denotes the concatenation operator. Using the ORnum and PageID to key a snapshot page causes all the snapshot pages corresponding to a single page to be mapped to a single ID. This facilitates the retrieval of snapshot pages in the p2p implementation of the archive store.

As mentioned, a `get` request returns the oldest snapshot page whose timestamp is greater than or equal to the TS_{ss} argument. Therefore, the implementation of `get` must have an

[1] SHA-1 is the Secure Hash Algorithm designed for use with the Digital Signature Standard to ensure the security of the Digital Signature Algorithm (DSA). In summary, when a message of length $\leq 2^{64}$ bits is used as the input to the SHA-1, it outputs a 160-bit message called the message digest. SHA-1 is collision resistant because given the digest, it is computationally infeasible to recover the message, or to find two different messages that produce the same message digest. The SHA algorithm is inspired and modeled after the MD4 message digest algorithm designed by Ron Rivest, Laboratory for Computer Science, MIT.

efficient way of determining what snapshot pages exist for the requested page. Computing the ID based on just the ORnum and PageID causes all snapshot pages for a page to be stored at the same p2p node and therefore `get` can be implemented efficiently.

DHash uses a data block's content hash as its ChordID. This makes the block self-certifying. Using the ORnum and PageID of a snapshot page to derive its ChordID does not provide this self-certifying property and the OR will have to sign each snapshot page using its private keys before archiving it. This is similar to DHash's root blocks, which are signed using the owner's private key. (Recall that the public key of the public-private key pair keys the root block.)

Storing and retrieving data from Chord can be expensive due to multiple hop lookups. To avoid the need to perform a Chord lookup every time a snapshot page is stored or retrieved, the archive layer in the OR caches the IP address of the successor node where snapshots of the page are stored. This IP address is obtained when a snapshot page is stored or retrieved for the first time and is stored in a *successor node map*, which maps each page (identified by the ORnum of the OR and PageID of the page) to the IP address of its successor node. The successor node map enables snapshot pages for that page to be stored and retrieved directly from the successor node without having to perform a Chord lookup.

The drawback of maintaining such a table is that as storage nodes leave and join the system, the table becomes outdated. In this case, the archive layer can still rely on Chord lookups to archive or retrieve snapshot pages. Although this does not compromise the correctness of the system, the performance gain of using this successor node table is diminished when nodes leave and join frequently. However, since we do not expect nodes in the archive store to leave and join frequently, using the successor node table is expected to improve the overall system performance of the archive store.

4.2.3 p2p Storage Organization

In this section, we outline the basic design of the storage organization in each p2p archive node; this is depicted in Figure 4-1. The snapshot pages are written to disk as a log. The node maintains an index that maps the ORnum, PageID and timestamp of a snapshot page to its location on disk; the index is implemented as a two-level hash table. Snapshot pages are of a fixed size and contain the timestamp of the snapshot that they belong to.

To make the index recoverable after a failure without explicitly making it persistent, we append the ORnum and PageID to the end of each snapshot page on disk. A special *end-of-archive* marker is also appended after the last snapshot page. After recovering from a failure, the node reconstructs the index by reading each snapshot page from the disk until the end-of-archive marker is reached.

4.3 Alternative Implementations

Our p2p archive store design provides us with the flexibility to use the network storage or local disk storage as the archive store. In either case, the OR still makes use of the same interface in its archive layer and the only difference that is observed by the OR is the delay experienced when snapshot pages are stored and retrieved.

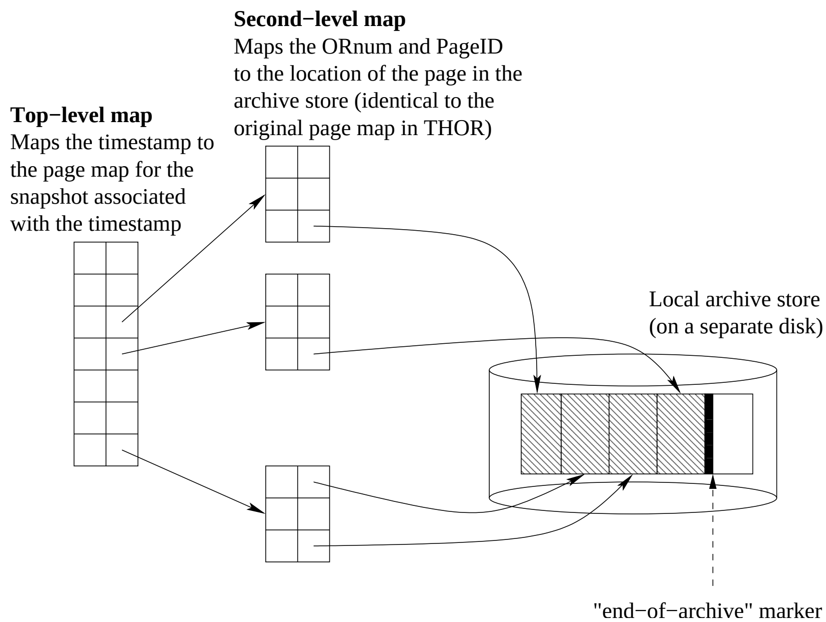


Figure 4-1: Storage System Design of an Archive Storage Node

4.3.1 Network Storage Implementation

In a network storage model, we use one or more well-known storage nodes on the same local area network as the OR as the archive store. The storage organization on this node is the same as that of a p2p node.

4.3.2 Local Disk Storage Implementation

In the local disk storage model, we use the storage organization of p2p nodes for storing the archive pages on a separate disk. A minor difference is that the first level hash table of the index will map only the PageID (instead of ORnum and PageID) to the second level hash table.

Chapter 5

Using Snapshots

Transactions are executed on cached objects in a FE. If a transaction uses an object that is not in the FE's cache, the page that contains the object is fetched from the OR into the FE's cache before the object is used. Objects modified by the transaction are shipped back to the ORs when the transaction commits. ORs keep track of cached objects at the FEs. Whenever an object is modified, an OR sends invalidation messages to all the FEs that have a copy of that object in their cache.

In this chapter, we extend the FE architecture to support the execution of read-only transactions on snapshots. After a snapshot has been taken, the user can execute read-only transactions on it. To execute such a transaction, the user specifies a time in the "past" at which the transaction should execute. The user is allowed to specify an arbitrary timestamp and the most recent snapshot, S, with timestamp less than or equal to the specified timestamp will be the snapshot used for this transaction. The FE also uses the timestamp of S to fetch snapshot pages from the ORs.

A simple approach for handling transactions in the FE is to clear the FE's cache every time a user runs a transaction on a different snapshot of the database. This prevents objects belonging to the wrong snapshot from being used in the transaction. However, clearing the FE's cache every time the user executes a transaction on a new snapshot forces the FE to discard hot objects from its cache. Thus it will seriously impact the performance of the system if the user switches repeatedly between snapshots, e.g., from the current database to a snapshot and back to the current database again.

Our approach to this problem is to allow FEs' caches to contain pages from multiple snapshots. The FE manages its cache in such a way that a transaction only sees objects of the correct version. For normal transactions, these are current objects; for transactions on snapshots, these are objects that belong to that snapshot.

5.1 Background: The THOR Front-End

Objects refer to other objects using 32-bit pointers, which are not large enough to store an xref. Instead, pointers store orefs: thus object can refer directly only to objects within their own OR. Objects point to other objects in a different OR indirectly via *surrogates*. A surrogate is a small object that contains the xref of the object it refers to.

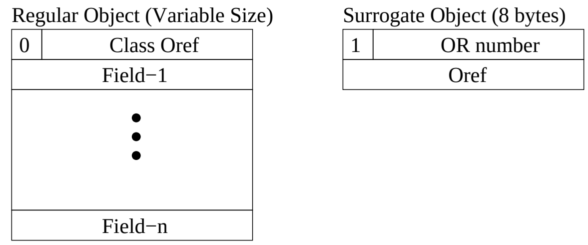


Figure 5-1: Object Formats in THOR

The object formats for regular and surrogate objects are shown in Figure 5-1. The header of an object contains the oref of the object's class object. The class object contains information such as the number and types of the object's instance variables and the code of its methods.

5.1.1 Persistent Cache And Hybrid Adaptive Caching

To speed up client operations, the FE caches copies of persistent objects fetched from the OR in its in-memory *Persistent Cache* (PC). A *page map* is used to efficiently locate pages in the PC using the page's PageID and the ORnum of the OR responsible for storing the page. A page is fetched from an OR on-demand, i.e., when an object in the page is accessed and the object is not in the FE's cache or has been invalidated. On the other hand, only modified objects are shipped back to the OR when the transaction commits.

FEs use a novel *Hybrid Adaptive Caching* (HAC) [4] scheme to manage its cache. HAC is a hybrid between page and object caching that combines the virtues of both while avoiding their disadvantages: it achieves the low miss penalties of a page-caching system but performs well when locality is poor because it can discard pages while retaining their hot objects during page compaction. HAC is also adaptive: when locality is good, it behaves like a page-caching system and when locality is poor, it behaves like an object-caching system.

The PC is organized into page-size *page frames* and a page fetched from the OR is stored into a page frame. As a FE only has a fixed number of page frames, it needs to compact its existing page frames to make room for a new incoming page if all its page frames are used up. This page compaction algorithm starts by selecting page frames with a low percentage of hot objects, referred to as victim frames. Cold objects from the victim frames are then discarded while the hot objects are compacted to free one of the frames. A page frame that holds a page fetched from the OR, in its entirety, is called an *intact page frame* and one that holds hot objects retained when their page frames were compacted is called a *compacted page frame*. Objects are classified as either hot or cold based on the recency and frequency of usage.

5.1.2 Indirect Pointer Swizzling and The Resident Object Table

To avoid having to translate the oref of an object to its memory location in the FE's cache every time a pointer is dereferenced, FEs perform pointer swizzling [15,28,42]: they replace the orefs in an object's instance variables by virtual memory pointers to speed up traversals.

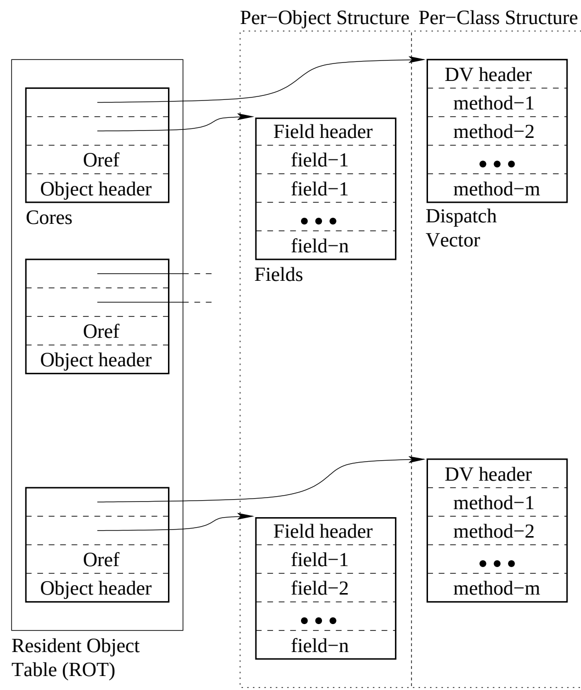


Figure 5-2: The Resident Object Table

In THOR, FEs perform indirect pointer swizzling [15] to reduce the overhead of moving and evicting objects from the cache. The *Resident Object Table* (ROT) [19, 20] is a data structure used for indirect pointer swizzling: object references are translated to pointers to ROT entries and ROT entires contain pointers to objects in the FE's cache.

Pointer swizzling and installation of objects in the ROT are done lazily: pointers are swizzled the first time they are loaded from an instance variable into a register; objects are installed in the ROT the first time a pointer to them is swizzled. Since an entire page is fetched from the OR whenever an object from that page is accessed, laziness allows the FE to avoid doing extra work of installing (in the ROT) objects that are in the cache but never used and of swizzling pointers to these objects.

Figure 5-2 shows the structure of the ROT in the FE. Each ROT entry is a called a *core* and contains the object's oref, a pointer to the object's fields, a pointer to the object's methods, and the object's header information.

The ROT makes it easy to move objects from one frame to another because we only need to change the ROT entry for the object that moves. Changing the ROT entry is sufficient to allow all objects that refer to the object to find it in its new location.

5.1.3 Using Objects in the FE

Table 5.1 shows the operations performed by a FE when an object is accessed. In summary, the object must be in the PC and the reference to the object must be swizzled to a valid ROT entry (pointing to that object in the PC) before the object is accessed.

No.	Reference Swizzled	Valid ROT Entry	Object in PC	Action
1	N	N	N	The object reference is not swizzled and the PC lookup fails. The FE sends a fetch request to the OR for the page that contains the object. When this page arrives, the FE installs this page in its PC. Next, the FE creates a ROT entry for the referenced object and swizzles the reference to point to this ROT entry.
2	N	N	Y	The object reference is not swizzled but the object is in the PC. Object's page was previously fetched when another object in the same page is accessed at the FE. The FE creates a ROT entry for the object and swizzles the reference to the object to point to the ROT entry.
3	N	Y	N	Impossible for the ROT entry to be valid while the object is not cached in the PC.
4	N	Y	Y	The FE swizzles the object reference to point to the valid ROT entry.
5	Y	N	N	Object reference was swizzled to an invalid ROT entry and PC lookup fails. This can be due to the object reference being swizzled before the object was invalidated. FE fetches the page from the OR and installs in the page into its PC. Then, it marks the invalid ROT entry as valid and makes it point to the object in the PC.
6	Y	N	Y	Object reference was swizzled but the object was invalidated since the last access. However, the page containing this object was fetched again from the OR due to an access of another object on the same page. The FE sets the ROT entry to point to the object in that page.
7	Y	Y	N	Impossible for the ROT entry to be valid while the object is not cached in the PC.
8	Y	Y	Y	Uses object pointed to by the ROT entry.

Table 5.1: FE Processing for Object Access

An object is invalidated when it is modified by some other FE in the system. When this happens, the OR sends out invalidation messages to all the FEs that have a copy of the object in their caches. The FE then discards the object and mark its ROT entry as invalid. In addition, an object is discarded and its ROT entry is marked as invalid when its page frame is evicted from the PC during page compaction.

When an object is discarded, its containing page is not evicted from the PC; the page compaction algorithm [4] determines which pages to evict. Subsequently, when the object is referenced again, the FE fetches another updated copy of its containing page from the OR. The page map is updated to point to the newly fetched page. However, there may be other objects in the original containing page that have been installed in the ROT; in this case, the FE uses the objects in the original containing page that are pointed to by ROT entries instead of the copies of these objects in the newly fetched page.

5.2 Executing Transactions on a Snapshot

The challenge in supporting transactions on snapshots is to ensure that this does not cause the system performance to degrade, especially in the common case of executing transactions on the current database. A cost will be incurred when the user switches between different snapshots of the database; our secondary goal is to minimize this cost.

5.2.1 Determining Snapshot from User Timestamp

The user can specify an arbitrary timestamp, which is smaller than the timestamp of the current time, for executing read-only transactions on a snapshot. The most recent snapshot with timestamp less than or equal to the specified timestamp is the correct version for this transaction. Like ORs, every FE also keeps a snapshot history list. A FE uses its snapshot history list to determine the correct snapshot for the user specified time. Subsequently, the timestamp of the correct snapshot, rather than the user specified timestamp, will be used for lookups and page fetches for this transaction. In addition, the FE maintains a timestamp, TS_{db}, which is the timestamp of the snapshot that it is currently running on. If the FE is running on the current database, TS_{db}'s value is set to null. Whenever a transaction starts running at a different "time", the FE updates TS_{db}'s value.

5.2.2 Clearing the ROT

When the user switches from the current database to a snapshot, or from one snapshot to another, we must ensure that the objects seen by transactions are objects belonging to the correct version of the database. A simple approach is to clear the PC before fetching the required snapshot pages from the OR. This prevents the FE from caching two versions of an object at the same time. However, clearing the PC is not desirable because it forces the FE to discard hot objects.

Our solution avoids clearing the PC. When the user switches between the current database to a snapshot, or from one snapshot to another, we mark all entries in the ROT as invalid. The effect of clearing the ROT is that subsequent accesses to objects in the snapshot transaction will cause a ROT "miss" and the FE will have to perform a lookup on the PC. Lookups on the PC are tagged with the timestamp of the snapshot to identify the version of the object that is required. Thus the lookup will succeed if the desired snapshot page for that object is in the PC. Otherwise, the lookup will fail and the FE will fetch the corresponding snapshot page from the OR. The fetching of the snapshot pages from the OR will be discussed in the next section.

After the OR returns the correct snapshot page, the FE installs this page in the FE's PC and sets the ROT entry of the object to point to the version of the object on this page.

The FE uses its $\mathrm{TS_{db}}$ to determine the timestamp of the snapshot it is running at and to ensure that all objects in the ROT belong to that snapshot. The ROT is cleared when the FE's $\mathrm{TS_{db}}$ changes. Subsequently, only objects belonging to the snapshot (represented by the FE's $\mathrm{TS_{db}}$ value) are installed in the ROT.

5.2.3 Persistent Cache Lookup

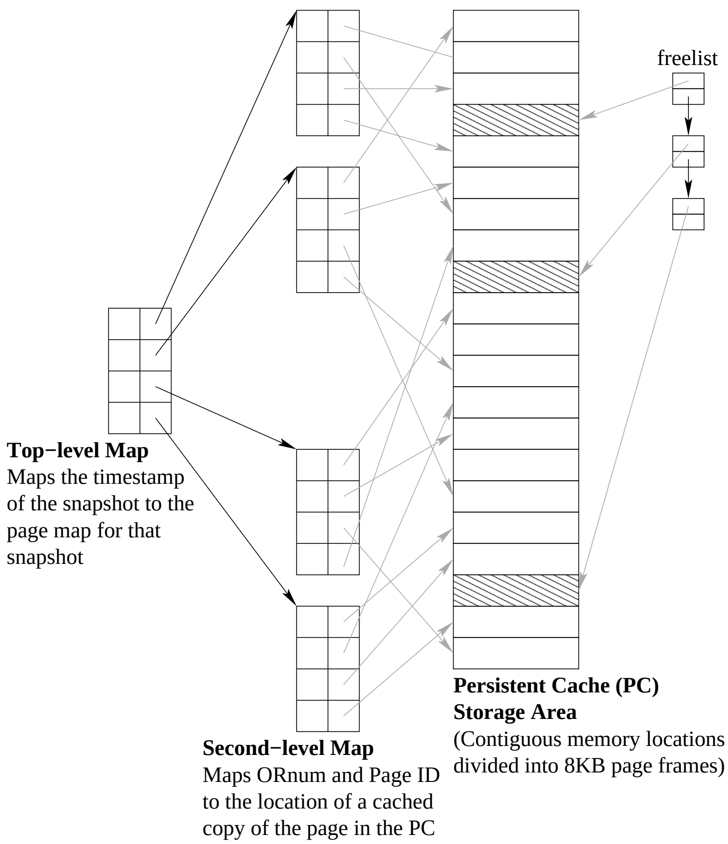
The FE uses a *page map* for mapping the ORnum and PageID of a page to the index of that page in the PC, i.e., it maps the system-wide identifier of the page to its location in the PC. The page map contains entries only for pages in its PC and not every possible page in the system. To support snapshots, the FE needs to map the system-wide identifier of a snapshot page to its location in the PC. This is done by tagging each page map lookup with the timestamp of the snapshot; lookups for current pages are tagged with a null timestamp.

We keep one page map for every snapshot with a page in the FE's PC; these page maps are implemented using a two-level hash table as shown in Figure 5-3. The top-level map, which contains only entries for snapshots with pages in the PC, maps the timestamp of a snapshot to the page map for that snapshot. The second level map is identical to the original FE page map.

Using a two-level map adds an extra level of indirection on page map lookups and hence degrades transaction performance. Since our goal is to avoid performance penalty on transactions on the current database, we should avoid using this two-level mapping when running on the current database. In our system, each FE maintains two separate page maps: a single-level page map (identical to the page map in the original system) for the current database and a two-level page map for snapshots. A FE uses its $\mathrm{TS_{db}}$ to quickly determine the appropriate page map to use. If the $\mathrm{TS_{db}}$ value is null, then the FE is running on the current database; in this case it uses the single-level mapping. On the other hand, a non-null $\mathrm{TS_{db}}$ value indicates that the FE is running on a snapshot and hence it uses the two-level mapping. In this way, we only incur performance penalty due to the additional level of indirection when running transactions on snapshots and not on the current database.

5.3 Handling Snapshot Page Fetch Requests

We extend the page fetch request (from a FE to an OR) to include an additional timestamp. This timestamp is set to the timestamp of the snapshot when fetching a snapshot page and to null when fetching a current page. The protocol for fetching current pages is unchanged. In this section, we discuss the handling of snapshot page fetch requests at the ORs.



The directory of the PC is extended to a two-level hash table that maps the identifier of the snapshot page, $<\mathrm{ORnum}, \mathrm{PageID}, \mathrm{TS}_{\mathrm{ss}}>$*, to its in the PC. Consequently, instead of having only one page map, the FE has a page map for every snapshot that has been recently used. Note that the freelist is used to keep track of free page frames in the PC.*

Figure 5-3: Structure of the Persistent Cache with Two-Level Page Map

When a snapshot page request arrives at an OR, the OR needs to determine whether the requested page has already been archived. It does this by consulting the snapshot page map. Recall that this 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. As discussed in Chapter 4, the archive will return the snapshot page with the largest timestamp that is less than or equal to the requested timestamp. If the archive store does not contain any snapshot copy of the requested page, a null is returned. In this case, the OR returns the current page to the FE.

On the other hand, if the timestamp of the request is more recent than what is stored in the snapshot page map, the OR will create the snapshot page using the MOB and the Anti-MOB, and return this newly created snapshot page to the FE. If the snapshot page is different from the current page, the OR also stores the page into the archive store and updates its snapshot page map[1]; this avoids creating the snapshot page again.

[1]Note that the OR's snapshot page map is different from the FE's page map.

Chapter 6

Experiments And Performance Evaluation

To evaluate the performance of our snapshot subsystem, we compared the performance of the original THOR [2] to that of *THOR-SS*, the version of THOR that supports snapshots. THOR-SS is a fully functional implementation of our design as described in Chapter 3 in the absence of failures. Our implementation includes making the Anti-MOB persistent but we have yet to implement the recovery mechanism. This, however, does not compromise the validity of our experiments because we have taken into consideration all the associated costs.

In this section, we present the results of our experiments. Our performance goal is to have only small or negligible overheads when executing transactions on the current database; this is the common case for THOR. Our experimental results show the performance of transactions on the current database on THOR-SS is comparable to that of THOR.

After a snapshot is taken, the OR creates snapshot pages when the MOB is cleaned and writes them to the archive. We measure the performance of running transactions when the OR is creating snapshot pages to measure the performance slowdown due to this additional activity.

We also present the results of executing a read-only transaction on a snapshot of the database and compare its performance against the performance of executing the same transaction on the current database. As the performance of executing transactions on snapshots is largely determined by the cost of retrieving snapshot pages, we conducted the experiment THOR-SS with different implementations of the archive store, namely (1) a local archive store on a separate disk, (2) a network archive store on the local area, and (3) a p2p archive store.

6.1 Experimental Setup

THOR was implemented with approximately 25,000 lines of C/C++ code and the snapshot subsystem adds another 4,000 lines of code. The system was complied with the GNU C/C++ 3.0.4 compiler. For this experiment, a simulator for the p2p archive was also developed with the SFS toolkit [22].

Parameter	Value
No. of Atomic Parts per Composite Part	200
No. of Connections per Atomic Part	3, 6, 9
No. of Composite Parts per Module	500
No. of Assembly per Assembly	3
No. of Assembly Levels	7
No. of Composite Parts per Assembly	3
No. of Modules	1
Size of Document (bytes)	2000
Size of Manual (bytes)	1M

Table 6.1: OO7 Medium Database Parameters

We used a single FE and a single OR for our experiments. The OR and FE ran on separate Dell Precision 410 workstations 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 simulating a p2p system with 60 nodes in the p2p archive store. Each workstation has a Pentium III 600 MHz CPU, 512 MB of RAM and a Quantum Atlas 10K 18WLS SCSI hard disk. The machines are connected by a 100 Mbps switched Ethernet and the round-trip time between any two machines on this network is approximately 130 μs. The FE was configured with 32 MB of persistent cache and the OR was configured with 16 MB of transaction log and 16 MB of page cache.

In our experiments, the space taken by the Anti-MOB was insignificant compared to the MOB because we assumed that the OR's T_{gmax} was very close to the current time and that the MOB was not processed beyond T_{gmax}. Consequently, the only entries that were created in the Anti-MOB were those caused by MOB overwrites; these entries pointed to the objects already in the MOB's heap and did not occupy extra memory space.

6.2 OO7 Benchmark

Our experiments are based on the OO7 Benchmark [3] for object-oriented databases. This benchmark is based on a single user workload and captures the characteristics of many different CAD applications without modeling any specific application. In this section, we provide a brief description of the benchmark.

The OO7 database contains a tree of *assembly* objects with 3 randomly chosen *composite parts* as its children. These composite parts are chosen from a population of 500 composite parts. Each composite part contains a graph of *atomic parts*, linked by bi-directional *connection* objects. Each atomic part can have either 3, 6, or 9 connections and our configuration used 3 connections. All atomic parts are reachable via a single root atomic part. We used the *medium* database configuration, which is approximately 44 MB in our implementation, for our experiments; this configuration is shown in Table 6.1.

We used the OO7 traversal operations, which are implemented as method calls on objects in the database, for our experiments. We considered both read-only traversals and read-write traversals. Read-only traversals invoke a null method on each atomic part object as it is visited while the read-write traversals update the state of the atomic parts when they are encountered.

We experimented with traversals *T1*, *T2A*, *T2B* and *T2C*:

- T1 (Read-Only): Measures the raw traversal speed by performing a depth-first search of the composite part graph, touching every atomic part. This experiments touches 437,400 atomic parts.
- T2 (Read-Write): Measures the traversal speed with updates. Traversal T2A updates one atomic part per composite part (21,870 updates), traversal T2B updates every atomic part per composite part (437,400 updates), and traversal T2C updates each atomic part four times (1,749,600 updates).

6.3 Baseline Experiments

Our first experiment compares the performance user transactions on the current database on THOR to that of user transactions on THOR-SS; running user transactions on the current database is the common case for our system. On the other hand, taking snapshots and running transactions on snapshots are relatively rare.

In this experiment, we measured the average execution times for T1, T2A, T2B, and T2C on a medium database with a hot FE cache, which contained all the objects used in the traversal. We use a hot FE cache to eliminate the cost of fetching pages from the OR since the number of pages fetched from the OR is the same for both systems and this cost dwarfs the cost of running and committing the traversals. By eliminating the cost of fetching pages from the OR, we are measuring the worst case performance penalty imposed by the snapshot subsystem because the cost of running a typical user transaction often includes the cost of fetching a few pages from the OR.

We assume that the OR's T_{gmax} is fairly up-to-date and that the OR does not clean its MOB beyond T_{gmax}. We further assume that all the snapshot pages that belong to earlier snapshots have been created. Consequently, the OR will not create any snapshot pages, i.e., we will not observe any cost due to the creation of snapshot pages. However, we will still be able to see the cost of saving overwritten MOB entires into the Anti-MOB. Even though we assume that T_{gmax} is fairly up-to-date and that the OR does not clean its MOB beyond T_{gmax}, a committing transaction's timestamp is also certain to be greater than T_{gmax}. Therefore, when an entry in the MOB is overwritten, it has to be saved into the Anti-MOB to prevent information loss.

As traversals are run repeatedly in our experiment, the slowdown experienced when running read-write transactions is the worst case because each object updated by the read-write transaction constitutes an overwrite in the OR's MOB. This forces the OR to store the overwritten object into the Anti-MOB. The result of this experiment is shown in Figure 6-1. The traversal time is the time from when the FE starts executing the traversal to the time the traversal ends; the commit time is the time from when the FE sends a commit request to the OR to the time the OR replies that the transaction has committed.

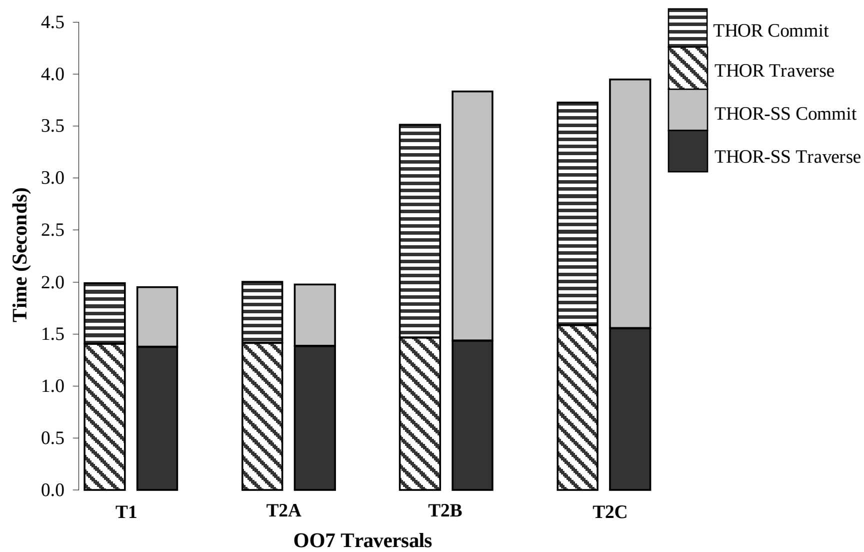


Figure 6-1: Baseline Comparison Between THOR and THOR-SS

As shown in Figure 6-1, there is no significant difference between the traversal times of the transactions in THOR and and that of transactions in THOR-SS. This is also the case for the commit times of T1 and T2A; only the commit times of T2B and T2C display slowdowns of approximately 15%.

In T2B and T2C, every transaction committed modifies 98,600 object, which causes 98,600 overwrites in the OR's MOB. The extra work done by THOR-SS in preserving these overwritten objects is the reason for this slowdown, which translates to less than 5μ seconds per object updated.

On the other hand, T1 and T2A do not show significant performance difference on THOR and on THOR-SS. In T1, no objects are modified and no MOB entries are overwritten; this means that the OR does not perform any extra processing. As a result, there is no slowdown. In T2A, only 494 objects are modified and the slowdown that is caused by the storing of the corresponding 494 overwritten MOB entires into the Anti-MOB is insignificant as compared to the commit time for the transaction.

6.4 Modified Workload Performance

As we have mentioned, the slowdown of T2B and T2C on THOR-SS is the worst case because every modification to an object causes an older entry in the MOB to be overwritten. This is not what we expect with a more realistic workload for our system. Therefore, we use another experiment to capture the expected system behavior.

In this experiment, we measure the performance of THOR-SS using a more realistic workload where only a small fraction objects modifications leads to an overwrites in the MOB. We modified T2B such that it randomly updates only 10% of all the atomic parts that it encounters. In the original T2B, 98,600 objects are modified per transaction whereas in

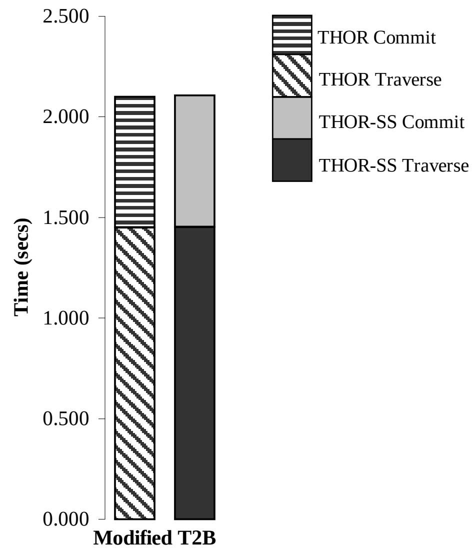


Figure 6-2: Comparison of Modified T2B on THOR and THOR-SS

the modified T2B, only 9,860 objects are updated per transaction. This reduces the degree of overlap of modified objects between each iteration of the traversal. The results of this experiment are given in Figure 6-2.

With this modified workload, the commit time on THOR-SS is only slower than that of THOR by a marginal 2%. The slowdown is due to the small number of overwriting that still occurs in the MOB. Our results show that with a more realistic workload, the performance of THOR and THOR-SS are comparable even for read-write transactions.

6.5 Transaction Performance With Snapshot Page Creation

In this section, we evaluate the performance of user transactions when the OR is creating snapshot pages. The creation of snapshot pages loads the server and, in turn, slows down the server-side performance of user transactions.

In the experiment, we first took a snapshot and then ran multiple read-write transactions until the MOB is cleaned. This forced the OR to clean its MOB and, in turn, triggered it to create snapshot pages. In addition, we assumed that T_{gmax} is always greater than the time of the snapshot and provided the OR with a large enough T_{gmax} so that snapshot pages could be created.

When creating a snapshot page as part of the MOB cleaning process, the OR needs to (1) write entries to the snapshot log and force it to disk, (2) write the snapshot page to the archive store, and then (3) write the current page to disk. However, since we are doing this work in the background, these tasks do not affect the end-to-end performance of the transactions. Rather, the transaction performance will be slowed down because of increased server CPU load due to the creation of snapshot pages.

In our experiment, the performance of T1 and T2A running on THOR-SS that is creating snapshot pages is comparable to that on THOR-SS that is not creating any snapshot page. This is because T1 is read-only and T2A only modifies a small number of objects. In other words, the performance impact on these transactions is very small because they require very little processing at the OR. On the other hand, the performance of T2B and T2C is slower by approximately 28% in both archive store implementations when the OR is creating snapshot pages; this is a result of the increased server load.

Figure 6-3 compares the performance of THOR-SS under normal operating conditions (when no snapshot pages are being created) and THOR-SS under loaded conditions (when snapshot pages are being created) with two implementations of the archive store, namely the local archive store and the network archive store.

From the figure, we see that there is no significant difference between running the traversals on THOR-SS with a local archive store and on THOR-SS with a network archive store. In a local archive store, the writing of snapshot pages to (a separate) disk is done asynchronously; in a network archive store, the pages are sent to the remote storage node using UDP. In both cases, the archiving of snapshot pages is done in the background and does not affect the end-to-end performance. We argue that this is also the case for a p2p archive store. In a p2p archive store, the archiving of snapshot pages takes a longer time because the storage nodes are further away. However, since we are using UDP to transport the snapshot pages, the archiving of snapshot pages does not slowdown transactions. Consequently, we expect that the implementation of the archive store to have little or no effect on the end-to-end performance of executing transactions on the current database, regardless of whether the OR is creating snapshot pages when the transaction takes place.

Finally, although the server-side performance is slowed by the creation of snapshot pages, we expect such slowdowns to be infrequent because snapshots are rarely taken. Furthermore, for a medium OO7 database (44 MB), the slowdown is only on the order of a few minutes. Therefore, we believe that this is an acceptable performance cost.

6.6 Read-Only Transaction Performance on a Snapshot

In this section, we compare the performance of a read-only transaction on a snapshot of the database with different storage designs for the archive store. When the user executes a read-only transaction on a snapshot of the database at the FE, the FE requests the required snapshot pages from the OR. The OR, in turn, fetches these pages from the archive store. Consequently, the design of the archive store will affect the performance of the transaction.

We evaluate THOR-SS with three different implementations of the archive store: (1) a separate magnetic disk at the OR, (2) a remote storage node over the network and (3) a p2p storage system. We compare the performance of the read-only T1 on these three different implementations.

Furthermore, we compare the performance of T1 on a snapshot with T1 on the current database to understand the performance penalty associated with running transactions in the “past”.

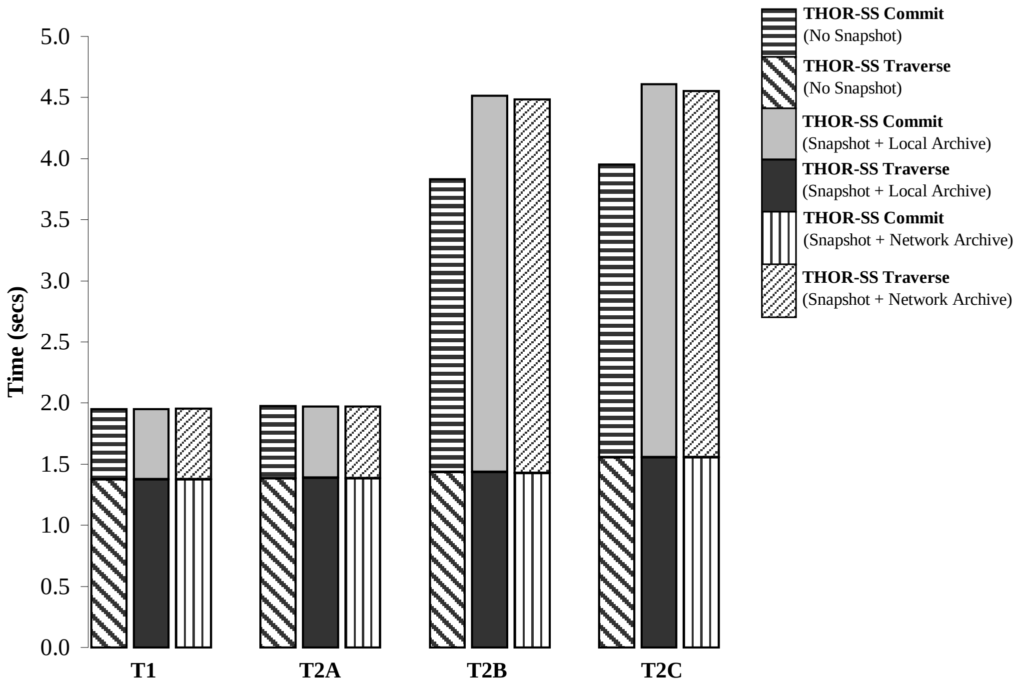


Figure 6-3: Performance of THOR-SS When Creating Snapshot Pages

The configuration of the OR and FE were the same as the previous experiments (FE: 32 MB PC, OR: 16 MB transaction log, 16 MB page cache). We run T1 on a cold FE cache, i.e., every page used by T1 must be fetched from the OR. A total of 5,746 snapshot pages are fetched for running T1. In addition, the OR's page cache was also empty prior to running T1.

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 running it on the current database. Every page that is used by T1 must be fetch from the OR, which, in turn, reads these pages from its disk. This is the case even when running on the current database. Therefore, when an object is accessed for the first time, its containing page will be fetched from the OR; the object will then be installed in the ROT.

When the FE executes a read-only transaction on a snapshot, it needs to clear its ROT. To measure the cost of clearing the ROT, we first executed T1 on the current database before we executed T1 on the snapshot. This caused 395,987 objects to be evicted from the ROT and took approximately 0.6 seconds, or 1.7μs per object evicted.

Figure 6-4 shows the results of our experiment (note that the y-axis is in logarithmic scale). The figure shows that there is only a slight slowdown of approximately 5% when snapshots are stored on the OR's disk. This slowdown is due entirely to processing at the FE. Because the ROT is cleared prior to running the transaction, the FE has to perform a ROT installation on every object that the transaction accesses (the first time it accesses

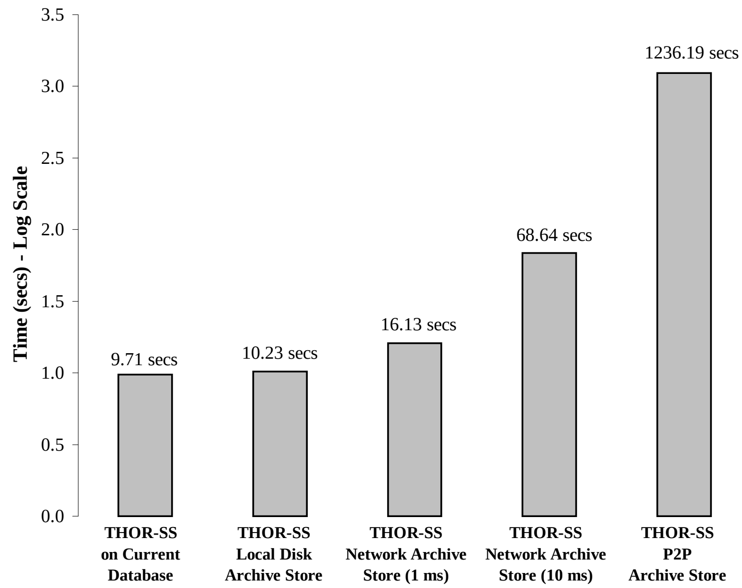


Figure 6-4: T1 Performance on Snapshot

Archive Store	Data Transfer Time (8 KB)	Total Time	Measured Time
Network	1 ms	1 ms/page × 5,746 pages + 10.23 secs = 15.98 secs	16.13 secs
Network	10 ms	10 ms/page × 5,746 pages + 10.23 secs = 67.69 secs	68.64 secs
p2p	210 ms	210 ms/page × 5,746 pages + 10.23 secs = 1,216.89 secs	1236.19 secs

Table 6.2: Calculated Versus Measured T1 Performance

the object). A ROT installation, in turn, causes a page map lookup. Since the FE uses a two-level page map for running on a snapshot of the database (as we have discussed in Chapter 5), the extra level of indirection causes transaction to experience a small slowdown when running on snapshots stored in a local archive store.

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: the time taken to run T1 is 15.6 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 (68.1 secs).

Performance is further degraded when THOR-SS uses a p2p archive store. The figure shows that when the data transfer time for 8 KB of data is 210 ms, the time taken to run T1 is two orders of magnitude slower (235.6 secs) than on THOR-SS with a local archive store.

Table 6.2 shows the comparison between the calculated and measured time for T1 for the various archive stores. If the snapshot is stored on a remote node, the performance of T1 is proportional to the time taken to fetch a snapshot page from the archive store. This is expected because the baseline cost of fetching snapshot page from disk and sending it from the OR to the FE is incurred regardless of whether the OR has a local archive store or remote archive store; the only additional cost is the cost of sending the snapshot pages across the network. In this experiment, 5,746 pages were fetched from the OR to the FE when executing T1. In our calculations, we estimate the baseline cost for executing transactions on snapshots by the time taken to execute T1 on THOR-SS with a local archive store (10.23 secs). From the table, we see that our experimental results coincide with our calculations.

In summary, the performance of read-only transactions on a snapshot increases as the network latency (network distance of the storage node) and bandwidth improve. It is not difficult to foresee that with increasing network bandwidth, such as in a gigabit network, the performance of running transactions on snapshots stored at a nearby archive store will be almost the same as running them on the current database.

6.6.1 Effect of Varying Network Latency

The message round-trip on a LAN is typically less than 1 ms while that on a wide area network is typically between 100 ms to 400 ms. For example, the round trip time of an ICMP ping between a computer at MIT and and a computer at Stanford is approximately 70 ms, and that between a computer in MIT and a computer at the University of Tokyo is approximately 200 ms. (Note that these numbers are intended to provide us with a rough idea of the typical wide-area latencies across different geographical locations and are not accurate representations of the network distances between these locations.)

Figure 6-5 shows the effect of varying data transfer time (for a 8 KB page) between the archive store and the OR on the performance of a read-only transaction on a snapshot. A zero data transfer time implies that the OR and the archive store is at the same machine, i.e., a local archive store. A larger data transfer time implies that the archive store is located further away from the OR.

We run T1 on a snapshot of the database as a benchmark for read-only transactions. We show only the traversal times since the commit times are the same across the different readings.

From the graph, we see that T1 takes 10 secs on THOR-SS with a local archive store. With a remote archive store, the performance of T1 decreases from 16 secs to 2,105 secs as the data transfer time for a 8 KB page increases from 1 ms to 316 ms. The relationship between the performance of T1 and the network distance is linear because the ORs always fetch one snapshot page at a time from the archive store and each snapshot page fetch incurs a delay that equals the fetch time for a 8 KB block.

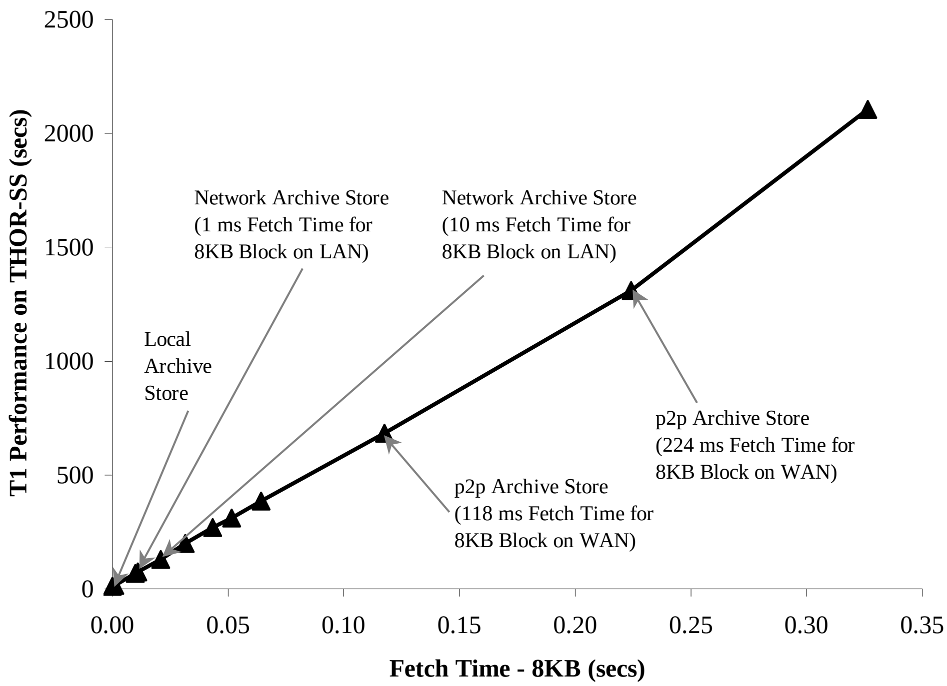


Figure 6-5: Performance of T1 on Snapshot with Different Data Transfer Rates

Chapter 7

Related Work

The classical algorithm for taking snapshots of a distributed system that captures a consistent global state of the system, was presented by Chandy and Lamport [6] in 1985. The algorithm is non-intrusive and does not make any time synchrony assumptions but assumes error-free FIFO communication channels. The global state of the system consists of the states of all the processes and the contents of all the communication channels. To record the global state, a special marker token is passed around the system.

In [9], Fischer, Griffeth and Lynch presented a non-intrusive global checkpointing algorithm for transaction-based systems. Their algorithm uses parallel processing: computation is split into two branches when a checkpoint operation is executed, with one branch simulating the execution of a normal transaction and the second branch handling the execution of checkpoints. In fact, one copy of the entire system is required for every checkpoint transaction executed.

Although these work provide firm theoretical foundations on algorithms for non-intrusive snapshots in distributed systems, most existing implementations of snapshot schemes either consider only taking snapshots of centrally located data or state, or require the system to block concurrent operations when the snapshot is taken.

7.1 File Systems

File system snapshots have close-to-open consistency and provide a read-only view of the entire file system at a particular point in time. They are usually created incrementally using block-level copy-on-write. In this section, we examine the snapshot utilities in some state-of-the-art file systems.

The Plan 9 [31] file system from Bell Labs has three levels of storage, namely the main memory cache (100 MB), the disk cache (27 GB), and the write-once-read-many (WORM) jukebox (350 GB). As the names imply, the main memory and disk storage are used as caches; the WORM jukebox provides immutable storage for all data in the system. A snapshot is created using an atomic dump operation that uses block-level copy-on-write to create the snapshot incrementally. To maintain consistency, the cache is flushed and the file system is frozen after all pending file operations are completed. Subsequently, blocks modified since the last dump operation are queued for writing, in the background, to the

WORM jukebox. A read-only copy of the dumped file system appears in a hierarchy of all dumps in the file system and is named by the time the dump operation is executed. In addition, the dump operation maintains the permissions of the files at the instance when the dump is executed. The dump operation is designed to capture snapshots of a centrally located file system and operates in an environment where the system can be frozen for the duration of the dump operation.

One of the main features of the Write Anywhere File Layout (WAFL) [12] file system, which is a file system designed for network access, is its snapshot utility. Like Plan 9, WAFL uses block-level copy-on-write to implement its snapshot. WAFL is organized as a tree of blocks rooted at the root inode block and data is stored at the leaf blocks. When the snapshot operation is executed, the root inode block is first duplicated. Subsequently, when a file is modified, the affected data blocks are duplicated; this causes their parent blocks to be duplicated as well. The process of duplication is propagated up the tree until the new root inode block is reached. During the snapshot, cached data is forced to disk and new file operations are blocked. This can be expensive if the cache is large and can impact system performance. WAFL keeps at most 20 snapshots and each snapshot is named using a system-generated number. The system keeps track of snapshots via their root inode; each snapshot has it own unique root inode. Snapshots can be mounted as a separate directory in the file system. SnapMirror [30] is an asynchronous remote mirroring system that uses the WAFL snapshot utility for incremental mirror update.

Petal [17] is a distributed storage system that provides a single virtual disk view of the underlying distributed storage system to applications such as file systems and database systems. Virtual disk locations are translated to physical disk locations using a global data structure at the top most level of the translation hierarchy. This data structure, called the virtual directory, is replicated and consistently updated on all the servers. A virtual directory entry is added for every snapshot in the system to enable the translation from virtual disk location to the snapshot's physical location. Each snapshot is distinguished using a monotonically increasing version number. Applications need to be frozen to update the virtual directory on every server when a snapshot is taken. Frangipani [40] is a file system built on Petal. It makes use of the Petal snapshot to create consistent dumps of the entire file system. To ensure consistency at the file system level when taking snapshots, the cache needs to be flushed and a system-wide lock needs to be acquired before the snapshot can be taken.

While Plan 9, WAFL, AFS [13] and Frangipani provide snapshots at the file system level, the Elephant versioning file system [35] provides a checkpoint operation at the file level to cater to the different usage patterns of the individual files. A version of a file is defined by the updates between an open and close operation on the file. Elephant provides several policies for file version retention. Like WAFL, Elephant maintains the versions of a file using the file's inode, which is duplicated when the file is opened. Subsequently, when the file is modified, the affected data blocks are duplicated using copy-on-write and the inode is updated. Finally, when the file is closed, the new inode is appended to an inode log for that file. Each version of the file is named by its inode number and the last modification time for that 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.

Ivy [29] is a p2p log-based file system based on Chord [37]. Each participant in Ivy creates a private snapshot of the file system by traversing the log in temporal order for records that are more recent than the last snapshot. The modifications in the log records encountered during the traversal are applied to the data blocks. The updated data blocks are subsequently stored back into the system. Private snapshots are optimizations that prevent the need to traverse the entire log. Since the log and data are immutable, and snapshots are created "after the fact", consistency of the private snapshots is guaranteed.

A direct application of file system snapshots to THOR does not meet our performance requirements. THOR is designed to handle a large number of transactions and cannot be frozen during a snapshot operation. Furthermore, THOR requires the consistency of snapshots to be maintained across distributed operations (transactions). This requirement is not considered in the snapshot schemes that we have described in this section.

7.2 Database Systems

The POSTGRES [38] storage system provides similar semantics to the Elephant file system with its no-overwrite scheme. Stemming from the argument that the popular Write-Ahead Logging (WAL) scheme complicates recovery, the POSTGRES storage system replaces WAL with a scheme where data is immutable. The first version of a record is called the anchor point and each anchor point has a chain of associated deltas. Walking the chain of deltas and applying the deltas to the record until the appropriate version is reached creates a snapshot of the record. Each record and its deltas contain the transaction ID of their respective transactions; these transaction IDs can be used to determine the version of the snapshot constructed. Although users are given the flexibility of constructing snapshots of the entire database at any point in time (in the past), this operation is expensive because it involves walking the chains of all the records in the system. In practice, no-overwrite has worse run-time performance than WAL schemes even for normal database operations [39].

IBM's DB2 [7] database management system provides an archive copy (AC) utility that support full and incremental fuzzy AC operations. A full AC operation copies all the data pages in the database when the AC operation is executed while an incremental AC operation only copies pages that have been modified since the last AC operation. Each AC is named using a log sequence number, which is a monotonically increasing system generated number. The correctness criteria for an AC is that it must capture the effects of all updates prior to the AC. In a multi-processor environment with multiple shared disks, all the processors must be informed of the AC operation before concurrent update transactions can continue. On the other hand, the Informix Online [14] database management system uses checkpoints to provide archive copies. Pages are copied on first write after the checkpoint. Logical timestamps are used to name the archive copies, as well as to track modifications. All update operations to the system must be blocked during the checkpoint operation.

Database AC utilities in IBM's DB2 and Informix suffer from the need to block concurrent transactions during the AC operation. DB2 offers a solution to this problem by allowing concurrent transactions to proceed in a single processor environment by using page latching (at the physical level) rather than page locking (at the transaction layer). The drawback of this scheme is that pages may contain uncommitted data. However, this is allowed in a fuzzy AC. In a multi-processor environment with multiple shared disks, all processors need to be blocked and be informed of the AC operation before concurrent operations can continue.

Like file system snapshots, AC utilities cannot be directly applied to THOR without any substantial performance penalty because applying these approaches directly to THOR requires concurrent operations to be blocked during the AC operation. This does not meet our performance goal of minimizing the performance penalties of concurrent transactions when taking snapshots.

7.3 Archive Storage Systems

In this section, we examine the Venti [32] archival storage system and its applicability to our archive store design.

Venti is a write-once archive storage system that replaces traditional tape-based backup solutions. Venti provides storage at the block level and organizes its storage using a hierarchy. Data blocks in Venti are addressed by their SHA-1 content hash and these addresses are packed into additional blocks called pointer blocks. The content hashes of pointer blocks are, in turn, packed into higher-level pointer blocks. This process is repeated until a single root pointer block is obtained. The resultant data structure is a hash tree that bears close resemblance to a Merkle tree [24]. The addressing of blocks by a collision-resistant content hash allows coalescing of duplicate blocks because if two blocks have exactly the same data, then their identifiers will be the same. Venti maintains an index for mapping the 160-bit block address to the physical location of the block; this index is constructed by scanning the data blocks. Maintaining such an index, which resides on the disk, is expensive; this is the main drawback of Venti.

We could have used Venti for our network archive store: each cluster of ORs in the system (that are close to each other on the network) would share a single Venti server. In this case, an additional directory structure that maps the ORnum, PageID and timestamp of the snapshot to the 160-bit block address would have to be maintained for each snapshot page.

Chapter 8

Conclusion And Future Work

8.1 Conclusion

In this thesis, we presented an efficient snapshot scheme for a distributed persistent object store. This scheme is able to capture transaction-consistent snapshots of the object store without interfering with concurrent transactions or compromising system performance. Our snapshot scheme guarantees serializability with concurrent transactions and provides external consistency within the limits of clock synchronization, which is on the order of tens of milliseconds.

The ability to support transaction-consistent snapshots with minimum impact on system performance is unique to our system. Our snapshot scheme is also scalable: snapshots are only committed at a single snapshot coordinator and information about the snapshot is propagated to the nodes in the system via a gossip mechanism. Our experimental results showed that our claim of performance and scalability is a reasonable one.

The main performance goal of our system is that snapshots should not interfere with transactions on the current database and degrade their performance. In our system, we trade-off granularity of snapshots for performance. We do not keep every possible version of the database; rather, we give the users flexibility to take snapshots as frequently as they require. There are two advantages to this approach. Firstly, keeping only the required snapshots significantly reduces the amount of storage for archiving the snapshot data (most of which will never be used if we were to keep all versions). Secondly, our scheme does not degrade the performance of the system when running on the current database.

We assume that snapshots occur relatively infrequently as compared to transactions and under these assumptions, our system meets our performance goal. In particular, our experiments showed that the performance of the version of THOR with snapshot support, THOR-SS, is comparable to that of the original THOR when executing transactions on the current database. Read-only transactions and update transactions that modify only a small number of objects have almost identical performance on both system. The worst case performance penalty where every object modified constitutes an overwrite in the MOB is 15%. On a more realistic workload, the performance penalty is only 1%, which is a reasonable price to pay for the added snapshot functionality.

We also evaluated the performance of running read-only transactions on a snapshot of the database. The performance of running transactions on snapshots depends on the storage design of the archive store. We evaluated three different implementations of the archive store, namely, the local archive store, the network archive store, and the p2p archive store.

Our experiments showed that the performance of a read-only transactions is fastest on the local archive store, with only a 5% slowdown, as compared to running the same transaction on the current database. Using a local archive store, however, does not allow the archive stores to share their excess storage, i.e., no load-balancing. A p2p archive store, on the other hand, provides automatic load-balancing and fault-tolerance. The main drawback of a p2p archive store is the delay experienced when fetching snapshot pages from a remote p2p node over the wide-area. This slows down the transaction by two orders of magnitude (120 times).

The network archive store on the local area provides sharing of storage between servers that are close to one another, as well as acceptable performance on a standard 100 Mbps network. Although we observe a 6.7 folds slowdown on T1 on a slower 10 Mbps network, the performance penalty was only 58% on a 100 Mbps network. By extrapolating, we expect the performance of the system with a network archive store on a 1 Gbps network to be close to that of the local archive store.

8.2 Future Work

In this section, we present three areas of future work to further improve the performance of the snapshot subsystem.

8.2.1 Extending the ROT to Support Snapshots

In our current implementation, the ROT is cleared whenever the FE switches between snapshot of the database, which is relatively expensive.

An alternative approach is to store snapshot information in the ROT entries themselves. The FE would need to check every use of a swizzled pointer to determine whether that ROT entry is for an object belonging to the database version currently being used. These checks are needed for all transactions, including those on the current database. Therefore the approach is feasible only if the checks can be done in a way that does not slow down transactions that run in the present.

8.2.2 Bypassing the OR for Transactions on Snapshots

When a user executes a transaction on a snapshot of the database, the FE sends a fetch request for the required snapshot pages to the OR. The OR, in turn, fetches these snapshot pages from the archive store. We can improve the performance of the system by having the FE bypass the OR and fetch the required snapshot page from the archive store. If the required snapshot page is found (there is a snapshot page with timestamp greater than or equal to the required snapshot's timestamp), it is returned to the FE directly. Otherwise, if the archive store does not have the snapshot page, it returns null. The FE then fetches the page from the OR.

This approach will improve the performance of THOR-SS with a network archive store as retrieving the snapshot pages directly from the archive store is faster than retrieving them via the OR. This approach will also improve the performance of THOR-SS with a p2p archive store. However, the performance improvement (for bypassing the OR) on a p2p archive store will be insignificant as compared to the cost of retrieving the snapshot pages from the archive store.

One problem with this approach is that the required snapshot page may not have been created by the OR, e.g., if the user executes transaction on the snapshot shortly after it was taken. In this case, the archive store returns null and the FE has to fetch the page from the OR. A solution to this problem is to send the fetch request to both the OR and the archive node at the same time.

8.2.3 Sharing of Pages at the FE

A fetch request for a snapshot page is satisfied by the oldest snapshot page with a timestamp greater than or equal to the timestamp of the requested snapshot page. In the case when the page has not been modified since the requested snapshot, the correct snapshot page would be the current page. Suppose that the FE already has a copy of the current page P in its cache and the user then executes a read-only transaction on a snapshot of the database that uses P. In addition, suppose that P was not modified since the snapshot. This means that the copy of P in the FE's cache actually satisfies the snapshot. In our current implementation, the FE does not take advantage of this fact.

An approach that allows the FE to share pages between a snapshot and the current database, or between two snapshots, can improve performance. The main challenge is to determine the correct information to provide the FE so that it can determine if the current page is the correct page for the snapshot.

Bibliography

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

[2] C. Boyapati. JPS: A Distributed Persistent Java System. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, September 1998.

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

[4] 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.

[5] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI)*, New Orleans, LA, USA, February 1999.

[6] K. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computing Systems*, 3(1):63–75, February 1985.

[7] R. Crus, F. Putzolu, and J. Mortenson. Incremental Data Base Log Image Copy. *IBM Technical Disclosure Bulletin*, 25(7B):3730–3732, December 1982.

[8] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-Area Cooperative Storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating System Principles (SOSP)*, Banff, Alberta, Canada, October 2001.

[9] M. Fischer, N. Griffeth, and N. Lynch. Global States of a Distributed System. *IEEE Transactions on Software Engineering*, 8(3):198–202, May 1982.

[10] 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.

[11] 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.

[12] 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.

[13] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1):51–81, Feburary 1988.

[14] Informix Online Administrator's Guide - Database Server Version 5.0, December 1991.

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

[16] 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.

[17] 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.

[18] D. Lewin. Consistent Hashing and Random Trees: Algorithms for Caching in Distributed Networks. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, March 1998.

[19] 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.

[20] B. Liskov, M. Castrol, 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.

[21] B. Liskov, M. Day, and L. Shrira. Distributed Object Management in Thor. In *Proceedings of the International Workshop on Distributed Object Management (IWDOM)*, Alberta, Canada, August 1992.

[22] D. Mazières. A toolkit for user-leve file systems. In *Proceedings of the USENIX Annual Technical Conference*, Boston, MA, June 2001.

[23] B. Mellish, V. Blazek, A. Beyer, and R. Wolatka. Implementing ESS Copy Services on Unix and Windows NT/2000, February 2001.

[24] R. Merkle. Protocols for Public-Key Cryptosystems. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, April 1980.

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

[26] D. Mills. Simple network time protocol (SNTP) version 4 for IPv4, IPv6 and OSI. Technical Report Network Working Group Report RFC-2030, University of Delaware, Newark, DE, USA, October 1996.

[27] C. Mohan and I. Narang. An Efficient and Flexible Method for Archiving a Data Base. In *Proceedings of the ACM SIGMOD Conference*, Washington, D.C., USA, May 1993.

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

[29] A. Muthitacharoen, R. Morris, T. Gil, and B. Chen. Ivy: A Read/Write Peer-to-Peer File System. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, Boston, MA, USA, December 2002.

[30] H. Patterson, S. Manley, M. Federwisch, D. Hitz, S. Kleiman, and S. Owara. SnapMirror: File System Based Asynchronous Mirroring for Disaster Recovery. In *Proceedings of the 1st Conference on File and Storage Technologies (FAST)*, Monterey, CA, USA, January 2002.

[31] 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.

[32] S. Quinlan and S. Dorward. Venti: A New Approach to Archival Data Storage. In *Proceedings of the 1st Conference on File and Storage Technologies (FAST)*, Monterey, CA, USA, January 2002.

[33] R. Rodrigues and B. Liskov. Rosebud: A Scalable Byzantine-Fault-Tolerant Storage Architecture. Submitted for publication.

[34] A. Rowston and P. Drushel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 18th ACM Symposium on Operating System Principles (SOSP)*, Banff, Alberta, Canada, October 2001.

[35] D. Santry, M. Feeley, N. Hutchinson, A. Veitch, R. Carton, and J. Ofir. 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.

[36] EMC SnapView/IP - Fast Incremental Backup, 2001.

[37] 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.

[38] 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.

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

[40] 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.

[41] EMC TimeFinder Product Description Guide, 1998.

[42] S. White and D. DeWitt. A Performance Study of Alternative Object Faulting and Pointer Swizzling Strategies. In *Proceedings of the 18th International Conference on Very Large Data Bases (VLDB)*, Minneapolis, MN, May 1992.