How to Scale Transactional Storage Systems

Liuba Shrira, Barbara Liskov, Miguel Castro and Atul Adya

Laboratory for Computer Science, MIT
Cambridge, MA 02139

Abstract

Applications of the future will need to support large numbers of clients and will require scalable storage systems that allow state to be shared reliably. Recent research in distributed file systems provides technology that increases the scalability of storage systems. But file systems only support sharing with weak consistency guarantees and can not support applications that require transactional consistency. The challenge is how to provide scalable storage systems that support transactional applications.

We are developing technology for scalable transactional storage systems. Our approach combines scalable caching and coherence techniques developed in serverless file systems and DSM systems, with recovery techniques developed in traditional databases. This position paper describes the design rationale for *split caching*, a new scalable memory management technique for network-based transactional object storage systems, and *fragment reconstruction*, a new coherence protocol that supports fine-grained sharing.

1 Introduction

The distributed applications of tomorrow will need to provide reliable service to a large number of users and manipulate complex user-defined data objects. Therefore, these applications will require large-scale distributed storage systems that provide scalable performance, high reliability, and support user-defined objects.

An important design challenge in such systems is avoiding the increasingly formidable server disk access bottleneck, a major performance impediment as systems scale to support many clients. Cooperative caching avoids disk access by taking advantage of emerging high speed local area networks to provide remote access to huge primary memories available in workstations. However, current cooperative caching techniques only work for file systems [5] and virtual memory systems [6], and do not provide support for transactions and fine-grained sharing (i.e., sharing of objects that are smaller than pages). In this paper we describe a new transactional storage system architecture, *split caching*, that adapts cooperative caching in a novel way to provide this support; our scheme achieves this goal in a way that avoids the problem of false sharing.

We assume a conventional client/server architecture: applications run at client machines and make use of objects stored at servers that provide persistent, highly-available storage. Client machines cache copies of persistent objects and make them available to applications, which run transactions that can access objects belonging to several different servers. Transaction commits are handled at servers (using a two-phase commit protocol if the transaction used objects from more than one server); when a transaction commits, its modifications are recorded persistently and become visible to other transactions. Note that the need to support transactions makes a "serverless" architecture [14] undesirable because the cost of distributed commit protocols increases with the number of participating servers.

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

Traditionally, server caches are used both to avoid disk reads and optimize disk updates. This organization is not scalable because the server cache becomes less effective in avoiding disk reads due to decreased locality in the server cache when there are many clients. Cooperative caching addresses this problem by letting clients fetch data from other client caches and by managing the aggregated client caches cooperatively. However, cooperative caching techniques [5] as used in file systems are not appropriate for transactional systems. In particular, caching modified data in client caches to optimize disk updates as in XFS [14] is not satisfactory since modified data is vulnerable to client machine crashes.

Split caching decouples the optimization of disk reads and disk updates, providing a new way of structuring the cooperative caches in transactional storage systems. Clients fetch pages from other clients' caches to avoid disk reads. Servers cache only new committed versions of recently modified objects in a large object cache called the *mcache*. The mcache is recoverable and therefore modified objects survive server crashes. The mcache allows disk updates to be performed more efficiently [13, 8]. However, when objects need to be moved from the mcache to the database on disk, it is necessary to first perform an *installation read* [13], or *iread*, to obtain the containing pages, since the mcache contains the modified objects but not the containing pages. Installation reads form a significant part of the disk load in the mcache architecture. Split caching avoids installation reads by fetching the containing pages from client caches.

Split caching relies on a cache coherence protocol called *fragment reconstruction* that avoids the penalties of false sharing. The protocol works in the presence of concurrency control techniques that support fine-grained sharing (e.g., adaptive call-back locking [3] or the optimistic approach used in Thor [1]); such techniques are desirable because they avoid conflicts due to false sharing. In such systems, when a transaction commits new versions of some objects, this may cause pages in other client caches to become *fragments*, i.e., pages containing stale copies of the objects modified by those transactions. Earlier research indicates that bringing such pages up to date by *propagating* the new object versions at commit time is not efficient [4]. Therefore, we instead invalidate those stale copies, and use fragment reconstruction to bring those pages up to date using the fragment and the information in the mcache.

Our work has been done in the context of Thor [2]. Thor supports object-based sharing and uses an mcache architecture to optimize updates. We describe a revised Thor design that incorporates split caching and fragment reconstruction. Although the design is based on Thor's optimistic concurrency control mechanism, it can easily be adapted to a system with a different concurrency control mechanism, such as adaptive call-back locking.

The paper is organized as follows. Section 2 describes the Thor architecture. The new split caching architecture and page reconstruction protocol are described in Section 3. We conclude in Section 4.

2 Thor Architecture

Our work is done in the context of the Thor object-oriented database system [2]. Thor has a distributed client/server architecture. Persistent objects are stored on disk at the servers. Application code runs at clients. Applications interact with Thor within atomic transactions that read and write persistent objects. Below, we give a brief overview of the Thor architecture, focusing on the parts that are important for our work. In particular, we describe the client–server protocol and the organization of disk storage management.

2.1 Client/Server Protocol

Each client has a cache that is used to store recently accessed objects. If the application requests an object that is not present in the client cache, the client sends a fetch request to the server. The server responds with a copy of the requested object. Clients cache objects across transaction boundaries.

Thor uses an optimistic concurrency control and cache consistency protocol based on loosely synchronized clocks and invalidation messages [1]. The Thor transactional consistency protocol has a number of interesting features: it uses timestamps to obtain a global ordering for transactions, and also to discard concurrency control information without causing aborts; it uses page granularity directories that keep track of cached pages and, in addition, keep track of objects on those pages that have been modified recently; it piggybacks cache invalidation information on messages already being exchanged between servers and clients. Simulation studies show that this scheme outperforms the best pessimistic scheme (adaptive callback locking [3]) on almost all workloads [9].

2.2 Server Organization

A server stores persistent objects and a stable transaction log on disk; it also has some volatile memory. The disk is organized as a collection of large pages that contain many objects. These pages are the unit of disk transer. The stable log holds commit information and object modifications for committed transactions. The server memory contains a modified object cache (the mcache) and a page cache that holds recently accessed pages. The mcache holds recently modified objects that have not yet been written to disk. As transactions commit, modifications are written to the log and also inserted in the mcache. To satisfy a fetch request, the server first checks the mcache, since it may contain the latest version of an object.

The mcache-based server architecture improves the efficiency of disk updates for small objects [13, 8]. It avoids the cost of synchronous commit time *installation reads* that obtain pages from disk in order to install the modifications on their containing pages. Studies show that installation reads performed at commit time can reduce the scalability of the server significantly [13, 15]. Instead, installation reads are performed asynchronously by a background thread that moves modified objects from the mcache to the disk using a read-modify-write cycle. First, the modified page is read from disk, if necessary. Then the system installs the modifications in the page and writes the result to disk. Once modifications have been written to disk, they are removed from the mcache and the transaction log, thus freeing up space for future transactions. If the server crashes, the mcache is reconstructed at recovery by scanning the log.

The mcache architecture reduces the number of disk update operations. It stores modifications in a very compact form, since only the modified objects are stored. This allows the system to delay writing modifications to the database longer than if the pages containing the objects were stored. Therefore, it increases *write absorption*: the mcache can accumulate many modifications to a page before an object in that page is installed on disk. This reduces the number of disk installation reads and also reduces the number of disk writes since the system can write all these modifications in a single disk operation while preserving the clustering that enables efficient disk reads.

Performance studies show that for most workloads the mcache architecture outperforms other architectures [13, 8], including conventional architectures in which the server stores modified pages, and the clients ship entire pages to the server at commit.

3 The Split Caching Architecture

The new caching architecture is based on the Thor architecture described in the previous section, except that clients fetch and evict pages instead of objects, and the servers no longer have a page cache; they only have the mcache. The new architecture allows clients to fetch pages from the caches of other clients and allows servers to avoid installation reads by taking advantage of pages in client caches.

The new architecture assumes the following environment. We assume that clients can communicate faster with each other than they can access the disk. This is true for fast switched local area networks like ATM and modern disks. We also assume that the users of client machines are willing to cooperate. Furthermore, we assume that clients are workstations with large primary memory. This is a reasonable assumption both because of current workstation technology trends, and because clients must have enough primary memory to cache their working set; otherwise, they will not perform well. Servers also have primary memory, but the aggregate memory at the clients is significantly larger than the server memory.

The new architecture is based on the following observations:

1. Earlier research has shown that when there is a large number of clients with caches, and when clients are sharing a large database whose size is much larger than the server memory, the server cache is relatively ineffective [11]. It is unlikely to satisfy client requests with pages fetched earlier for that client since they are already present in that client's cache. Furthermore, it is unlikely to satisfy client requests with pages fetched for another client because the probability that the needed information is in the server cache is small. The aggregate memory at the clients will do at least as well as the server cache, and is likely to do better because it is much bigger than the server cache.

This observation has motivated the work on cooperative caching in the xFS file system [5] and in traditional client/server databases [7].

2. Work on the mcache has shown that for most workloads, performance improves as memory is shifted from the server cache to the mcache [8]. This result is consistent with the first observation: since the server page cache is not very effective, it makes sense to shift the memory to the mcache (where it can be used to delay installing changes in the database).
3. Installation reads form a significant part of server disk load when the aggregate client memory exceeds the server cache; by the time the server needs to propagate the modification to the disk, it is unlikely to find the containing page in the server cache and needs an installation read. If the server cache is highly ineffective, the disk load due to the installation reads equals the the disk load due to writes.
4. Fine-grained concurrency control is much better than coarse-grained concurrency control because it avoids the problem of false sharing.

The first three points together with our environment assumptions lead to the split caching idea. Since the server page cache is not effective in avoiding disk reads, the server does not have a page cache; it only has a large mcache. This optimizes server disk writes. Clients fetch and evict pages and these pages are used for fetches and installation reads. Like other cooperative caching schemes, split caching allows the clients to fetch pages from other client caches. In addition, like the hybrid caching scheme [12], split caching allows the servers to avoid installation reads by taking advantage of pages in the client cache. In contrast to hybrid caching, split caching fetches pages at installation time, rather then commit time, which is more efficient when pages are updated repeatedly. This avoids disk reads ar the server.

The fourth point motivates the need for fine-grained sharing. Our architecture supports fine-grained sharing by providing a transactional caching and coherence protocol that works with Thor's object-based concurrency control scheme. When a client modifies a set of objects and commits its transaction at a server, it causes copies of these objects in other client caches to be invalidated. The server uses the directory information to determine which clients need to be informed and sends invalidation messages to these clients. When a client receives an invalidation message for an object x, it marks x as invalid (if the current transaction has used x, the transaction is aborted). These invalid objects in a client's page are termed *holes* and a page with holes is called a *fragment*. Fragments support fine-grain sharing and avoid the penalties of false sharing. They avoid unnecessary synchronization conflicts; if the client is required to invalidate the entire page on which x was located, it may have to abort unnecessarily (if the client had used other objects on that page). Also, fragments allow retaining useful objects in the cache; if the client discards entire invalidated page, it may discard frequently used objects that happen to reside on the page.

To use fragments in split caching, we must ensure that applying updates to a fragment obtained from a client cache results in an up-to-date copy of the corresponding page; otherwise, inconsistent pages may be written to disk. This condition may be violated if the fragment is missing updates that have already been installed on disk and are no longer in the mcache. Our cache coherence protocol, *fragment reconstruction*, guarantees that such situations do not occur, i.e., whenever a page is reconstructed using a fragment and the mcache, the protocol ensures that the page is up-to-date.

To ensure correctness, the cache coherence protocol has to be coordinated with the mcache. As mentioned, the server maintains directories that record information about what pages are cached at each client. For each page it records a status: *complete*, *reconstructible*, or *unreconstructible*. A complete page at a client has the latest versions of all its objects. A reconstructible page may contain old versions of some objects, but new versions of these objects are stored in the mcache. An unreconstructible page may contain old versions of some objects for which the mcache does not contain new versions.

The fragment reconstruction protocol maintains the changing page status as clients and servers fetch pages, transactions commit, and updates are propagated to disk. It uses page status to decide if a cached page can be used to satisfy fetches. The following sections discuss the actions of the fragment reconstruction protocol.

3.1 Redirecting Fetches

The server redirects fetches to client caches to reduce the number of disk accesses. When there is a miss in client A's cache, A requests the page of the missing object from the server. The server checks its directories to determine whether the page is present in the cache of another client. It redirects the request to another client B only if B's directory information indicates that B's page is either complete or reconstructible. In either case, the server asks B to send the page to client A. If the page is marked as reconstructible, it also sends the updates in the mcache for that page to client A (this client must wait for the updates from the server and the fragment from client B before it can proceed). If the page cannot be obtained from a client cache, the server reads it from disk in the usual way. The server then marks the page as complete in the directory for client A.

If the requested page has uncommitted modifications, client B does not send them to client A; it sends only the latest *committed* versions of objects. To do this, a client makes a copy of an object the first time it is modified in a transaction. These copies are discarded when the transaction commits or possibly due to a cache management policy decision. If client B receives a request for a page and it does not have the copy of the committed state for one of the objects in the page, it replies to the server saying that the page is unavailable. The server will then obtain the page from disk or another client and send it to A.

Client B may be unable to satisfy the fetch request if it has discarded the page. In this case, it replies to the server, which marks the page as absent from client B's cache. To improve performance, a client can inform the server when it evicts a page by piggybacking that information in its next message sent to the server.

When the server commits a transaction for a client, this has no impact of the status of pages stored at that client, but it can affect the status of pages at other clients, since they may become out-of-date for some objects. The server checks to see whether any complete pages at other clients have been affected by the transaction, i.e., whether the transaction has modified objects in a complete page at some other client. All such pages are marked reconstructible.

3.2 Avoiding Installation Reads

The server fetches pages from clients to avoid installation disk reads. When the server wants to install object modifications for page P from the mcache, it fetches page P from a client A if the directories indicate that A has a complete or reconstructible version of the page. Otherwise, the server simply reads the page from disk. The server installs the modifications and removes them from the mcache. It then checks client directories to determine if any client has reconstructible copies of page P. Directory entries of page P for all such clients are marked as *unreconstructible*. As a result, such pages will not be used in future fetches or installation reads. Marking of these cached pages as unreconstructible is necessary because after discarding the modifications from the mcache, the server can no longer bring the pages up to date using the mcache.

3.3 Optimizations

Now we discuss some optimizations to the protocol. Our first optimization avoids pages becoming unreconstructible; unreconstructible pages are not desirable because they can no longer be used in the global cache and only benefit the client caching them. When a server decides to install object updates, instead of just asking a client A for the fragment, it sends the mcache updates for the page to A. The client installs the modifications to the page and sends back the complete page to the server. The server then marks the page as complete at that client. This technique prevents a page from becoming unreconstructible at client A. This optimization is similar to update propagation but is not exactly the same. The server sends the mcache updates relatively rarely, e.g., the updates may be sent to a client after many modifications to the same page have been absorbed in the mcache. Note that the server need not update all the cached copies for the page; it just needs to ensure that there exists at least one reconstructible page in the system.

Suppose that a client has a page fragment and needs an object that is missing from the page. The client does not need to receive the whole page from another client. If the client informs the server that it has the page, and the server determines that the page is reconstructible, the server can simply send the updates in the mcache to the client and mark the page as complete. This optimization avoids a network roundtrip delay of getting the page from another client.

Finally, it is possible to distinguish "degrees" of fragmentation. Suppose that a client A fetches page P from a server (with some updates from the mcache); the server marks this page complete for client A. The server now receives an update for object x on page P from another client and marks P as reconstructible for client A. If client A now asks for page P, the server should send object x only (rather than all the updates for P in the mcache). This optimization can be implemented by maintaining extra information in the directory entries for each client. The server increments a counter whenever a transaction commits; in a client's directory entry, it stores the number of the latest transaction whose modifications are reflected in the client's cached copy. We rejected this scheme in favor of the one described here because our scheme is simpler and requires less storage at servers. Furthermore, the extra information will probably not have much impact on system performance: it may reduce the message size for fetch replies but does not reduce the number of messages in the system.

4 Conclusions

This paper considers the challenge of providing scalable storage systems for transactional applications and describes the new scalable transactional storage system architecture we are developing in the Thor system. The new architecture is based on the split caching global memory management scheme, and the transactional fragment reconstruction coherence protocol.

Split caching is of interest because it provides the first global memory management architecture for distributed transactional object storage systems. Like cooperative caching in serverless file system [5], split caching increases the scalability of the storage system by exploiting remote memory access. In addition, the decoupled optimization of disk reads and disk writes in split caching architecture also reflects transactional storage reliability requirements.

The fragment reconstruction coherence protocol is attractive because, like the log-based coherency protocol of Feeley et.al, [10], it avoids the penalties of false sharing in a transactional object storage system. However, the fragment reconstruction protocol supports storage systems with multiple servers and cooperative caching. In addition, because the fragment reconstruction protocol is integrated with the recoverable mcache, it can take advantage of lazy update propagation and absorption, which improves performance when pages are updated repeatedly. Importantly, it works correctly even in the presence of client failures.

Earlier work shows that cooperative caching is effective in improving the scalability of a storage system, and that support for fine-grain sharing avoids the performance penalties of false sharing. Split caching and fragment reconstruction integrate these two techniques in a new context of a transactional storage system, and we hypothesize, should retain the performance benefits of both techniques. We are currently implementing split caching and fragment reconstruction in Thor and exploring how they interact with mcache management.

5 Acknowledgments

We gratefully acknowledge Maurice Herlihy for his comments on the paper.

References

[1] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari. Efficient Optimistic Concurrency Control Using Loosely Synchronized Clocks. In *Proceedings of ACM SIGMOD*, May 1995.

[2] 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 ACM SIGMOD*, 1996.

[3] M. Carey, M. Franklin, and M. Zaharioudakis. Fine-Grained Sharing in a Page Server OODBMS. In *Proceedings of ACM SIGMOD*, 1994.

[4] Michael J. Carey, Michael J. Franklin, Miron Livny, and Eugene J. Shekita. Data caching tradeoffs in client-server DBMS architectures. In *Proceedings of the ACM SIGMOD*, pages 357–366, 1991.

[5] M. Dahlin, R. Wang, T.Anderson, and D. Patterson. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In *Proceedings of OSDI*, 1994.

[6] M. Feeley, W. Morgan, F. Pighin, A. Karlin, H. Levy, and C. Thekkath. Implementing global memory management in a workstation cluster. In *Proceedings of SOSP*, 1995.

[7] M. Franklin, M. Carey, and M. Livny. Global Memory Management in Client-Server DBMS Architectures. In *Proceedings of 18th VLDB Conf.*, 1992.

[8] S. Ghemawat. *The Modified Object Buffer: A Storage Management Technique for Object-Oriented Databases.* PhD thesis, MIT, 1995.

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

[10] M.Feeley, J.Chase, V.Narasayya, and H.Levy. Integrating Coherency and Recoverability in Distributed Systems. In *Proceedings of OSDI*, 1994.

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

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

[13] James O'Toole and Liuba Shrira. Opportunistic Log: Efficient Installation Reads in a Reliable Object Server. In *Proceedings of OSDI*, 1994.

[14] T.Anderson, M. Dahlin, J. Neefe, D. Patterson, and R. Wang. Serverless Network File Systems Performance. *ACM Transactions on Computer Systems*, 14(1), 1996.

[15] Seth J. White and David J. DeWitt. Implementing Crash recovery in Quickstore: a performance study. In *Proceedings of ACM SIGMOD*, pages 187–198, 1995.