Fragment Reconstruction: A New Cache Coherence Scheme for Split Caching Storage Systems (Looking at the Doughnut and not the Hole)

Liuba Shrira, Barbara Liskov, Miguel Castro and Atul Adya

Laboratory for Computer Science
MIT
Cambridge, MA 02139

Abstract

Fragment reconstruction is a cache coherence protocol for transactional storage systems based on global caching in a network of workstations. It supports fine-grained sharing and works in the presence of object-based concurrency control algorithm. When transactions commit new versions of objects, stale cached copies of these objects get invalidated. Therefore, pages in a client's cache may become *fragments*, i.e. contain "holes" corresponding to invalid objects. When such a page is used in the global cache, the coherence protocol fills in the holes using modifications stored in a recoverable cache at the server.

Fragment reconstruction is the first coherence protocol that supports fine-grained sharing and global caching in transactional storage systems. Because it is integrated with the recoverable modification cache, it works correctly even in the presence of client failures, and can take advantage of lazy update propagation and update absorption, which is beneficial when pages are updated repeatedly. This paper describes the fragment reconstruction protocol and presents its correctness invariant wich insures that only correctly reconstructed fragments are propagated to the database.

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.

Global caching techniques [14, 6] are a promising approach in scalable storage systems, that avoids the increasingly formidable disk access bottleneck, a major performance impediment as systems scale to support many users. Global 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 global caching techniques only work for file systems [14] 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 cache coherence protocol *fragment reconstruction* that supports fine-grained sharing and global caching in a transactional storage system architecture called *split caching*. In split caching, clients exploit network speed and large aggregate client memory by fetching pages from other clients' caches to avoid disk reads. Servers cache only new versions of recently modified objects, using a large recoverable cache called the *mcache*, to perform disk updates more efficiently [17, 8]. When objects are moved from the mcache to the database on disk, it is necessary to first perform an *installation read* to obtain the containing pages; these pages are read from client caches to avoid read-

ing them from disk.

Fragment reconstruction coherence protocol supports fine-grained sharing in split caching. The protocol works in the presence of object-based concurrency control techniques (e.g., adaptive call-back locking [2] 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. Research indicates that bringing such pages up to date by *propagating* the new object versions at commit time is not efficient [3]. 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. Because the coherence protocol is integrated with the recoverable mcache, we can take advantage of lazy update propagation and update absorption, which is beneficial when pages are updated repeatedly. Importantly, the protocol works even in the presence of client failures.

Our work has been done in the context of Thor [12]. Thor supports fine-grained sharing and uses an mcache architecture to optimize updates. However, Thor clients maintain an object cache, whereas split caching architecture uses a page cache; and Thor servers have a page cache, unlike servers in the split caching architecture that only maintain the mcache. We describe a revised Thor design that incorporates fragment reconstruction and split caching. Although the design is based on Thor's optimistic concurrency control mechanism, it can be easily adapted to a system with a different concurrency control mechanism, such as adaptive call-back locking.

The paper is organized as follows. We introduce the basic Thor architecture in Section 2, describe the split caching and fragment reconstruction protocol in Section 3, and sketch a correctness proof for the protocol in Section 4. We discuss related research in Section 5 and our conclusions in Section 6.

2 Thor Architecture

We are carrying out our studies in the context of the Thor object-oriented database system [12]. 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 front-end 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 a new optimistic concurrency control and cache consistency protocol based on loosely synchronized clocks and invalidation messages [1]. When the application requests a transaction commit, the new values of the objects modified by the transaction are sent from the client cache to the server along with other concurrency control information. The server validates the transaction, using a two-phase commit protocol if a transaction accesses multiple servers. The server stores information about prepared and committed transactions in a stable write-ahead transaction log; it notifies a client that a transaction has succeeded only after the modifications have been recorded in this log. If the transaction commits, the client starts executing another transaction; if it aborts, the client run-time system first restores the cached copies of modified objects to the state they had before

the transaction started (or discards those objects if it is not possible to restore their old states).

When a transaction commits successfully, the updates performed by that transaction become immediately visible and will affect future fetches. Earlier fetches could not reflect the changes and therefore client caches may contain obsolete information. Transactions that use the obsolete information will abort. To reduce the probability of such aborts, servers notify clients about obsolete information by sending them *invalidation messages*. When a client receives such a message, it removes the obsolete information from its cache and aborts the current transaction if it used the obsolete information. Invalidation messages allow such aborts to happen quickly, avoiding wasted work, and offload detection of the aborts from the server to the client.

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 [2]) on almost all workloads [9].

2.2 Server Organization

The servers have disk for storing persistent objects, a stable transaction log and some memory. The disk is organized as a collection of large pages that contain many objects, and that are the items read and written to disk. The stable log holds commit information and object modifications for committed transactions. The server memory is partitioned into a page cache and a modified object cache (the *mcache*). The page cache holds pages that have been recently accessed by clients. The mcache holds recently modified objects that have not yet been written to disk. As transactions commit, modifications are streamed to the log and also inserted in the mcache.

To satisfy a fetch request, the server first looks in the mcache, since if an object has been modified recently, its newest version is stored there. If the object is not found in the mcache, the server looks in the page cache. If the object is not in the cache, the server reads the object's page from disk, stores it in the cache, and replies to the client when the disk read completes.

The mcache-based server architecture improves the efficiency of disk updates for small objects [17, 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. Installation reads performed at commit time can reduce the scalability of the server significantly [17, 18]. 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. 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 [17, 8], including conventional architectures in which the server stores modified pages, and the clients ship entire pages to the server at commit.

3 Fine-grained Sharing in Split Caching

This section describes the split caching architecture and presents the new fragment reconstruction coherence protocol that suports fine-grained sharing in split caching. The split caching architecture is based on Thor. As in Thor, clients send the new versions of modified objects to the server when transactions commit. The server stores them in the transaction log and the mcache. However, unlike Thor, a server does not have a page cache; it only has the mcache. Furthermore, clients fetch and evict pages instead of objects. The split caching architecture extends Thor by allowing clients to fetch pages from the caches of other clients; and by allowing servers to avoid installation reads by taking advantage of pages in client caches.

The split caching architecture assumes that clients can communicate faster with each other than they can access the disk. This is true for modern local area networks and disks. It also assumes that the users of client machines are willing to cooperate. Clients are workstations with reasonably large primary memory. This is a reasonable assumption for two reasons: clients must have enough primary memory to cache their working set otherwise they will not perform well. Increase in primary memory is supported by current trends in workstation technology. Servers also have primary memory, but the aggregate memory at the clients is significantly larger than the server memory.

Split caching architecture is based on the following three observations.

1. Earlier research has shown that when there is a large number of clients with caches, and when they are sharing a large database whose size is much larger than the server memory, the server cache is relatively ineffective [16].

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].

3. Fine-grained concurrency control is much better than coarse-grained concurrency control because it avoids the problem of false sharing.

The first two points motivate split caching. Since the server page cache is ineffective, servers only have an mcache. Clients cache pages and these are used for fetches and installation reads, thus avoiding disk reads at the server.

The third point motivates the need for the fragment reconstruction coherence protocol. When transactions commit it is possible that they modify objects that reside in pages in other client's caches. At that point we could bring those pages up to date by sending the new states of the modified objects to those clients. Earlier research has shown that such propagation is not always a good idea. For example, it is wasted work if that client is not going to use the modified object. Therefore, we do not propagate the objects to the clients. Instead, the server just informs the relevant clients to mark the old versions of objects as invalid. We use a new coherence protocol, *fragment reconstruction*, to determine whether it is possible to rebuild a page from the client cache and the mcache. A client page is used to satisfy a fetch or an installation read only if the page can be reconstructed.

The details of our approach are described in the following sections. We describe the caching architecture, discuss cache management

at clients, describe the cache coherence protocol, and then discuss some possible optimizations.

3.1 Split Caching

As discussed earlier, split caching relies solely on the client caches to avoid disk reads, and uses the server cache solely to optimize disk updates. The entire server memory is dedicated to the mcache. In addition, the server maintains directories in which it records information about which pages are cached at which clients.

Split caching uses client caches for two purposes. First, to avoid disk accesses at the server due to client fetches. When there is a miss in a client cache, the client 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. If so the server redirects the fetch to this client, which sends the page directly to the first client. In addition, the server forwards all updates in the mcache for that page to the first client, which uses them to update the page. If the page cannot be obtained from a client cache, the server reads it from disk in the usual way, and sends the page and updates to the page in the mcache to the requesting client. In response to a fetch, the client gets a description of what data to expect (a page and updates or just a page or just updates) followed by the data itself, and applies the updates as required. In the case when the page comes from the server disk, this organization enables a very efficient path from the server disk directly onto the network and into the client cache. This avoids page copying through the server memory and can be beneficial in reducing the server load and improving server scalability.

Second, client caches help the server in reducing disk accesses due to installation reads. When a modification has to be moved from the mcache to the data disk, the server checks the client directories to determine whether that modified page exists in some client cache. If so, the server fetches the page from the client (else it simply reads the page from disk), updates the page with the modifications stored for it in the mcache, writes the page to disk, and removes the modifications from the mcache.

3.2 Fragments

As discussed earlier in this section, 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*. Support for fine-grain sharing by performing invalidations at the object level allows the client to avoid extra aborts; if the client is required to invalidate the page on which x was located, it may have to abort unnecessarily (if the client had used other objects on that page). This approach also avoids discarding frequently used objects that happen to reside on the page. As a result, this scheme allows eviction to be based on general algorithms such as LRU, although the algorithms might also take into account page "population" (e.g., a page that is very fragmented might be a more desirable victim than one that is full).

3.3 Integrating Cache Coherence and Modified Object Cache

To ensure correctness, the cache coherence protocol has to be coordinated with the mcache. The system must ensure that applying updates to a fragment obtained from a client results in an up-to-date copy of the corresponding page. 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.

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.

Now we discuss how the system works. When the server responds to a fetch request from client A, 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). In either case, the server then marks the page as complete in the directory for client A. Figure 1 shows the case where client B has a fragment cached for page P.

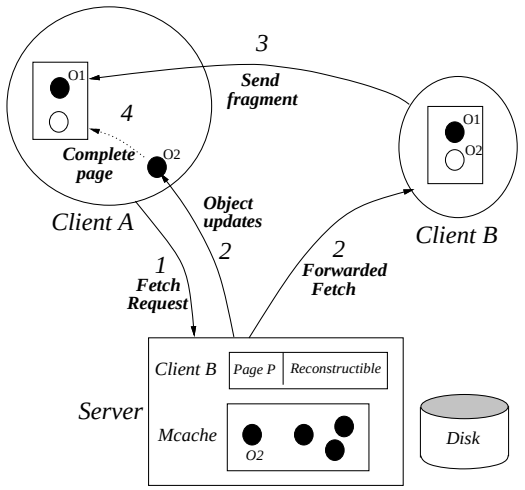

Figure 1: Redirecting an object fetch

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 when there is cache management. If client B receives a request for a page and it does not have the copy of the commited 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 P. In this case, it replies to the server, which marks the page as absent from client B's cache. To improve performance, clients can inform the server when they evict a page by piggybacking that information on the next message sent to the server.

When the server commits a transaction for a client, this has no impact of the statuses of pages stored at that client, but it can affect statuses 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.

Fetching pages from clients for avoiding installation disk reads is done in a similar manner. 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 A has a complete or reconstructible version of the page. After installing the modifications, the server removes the changes from the mcache and checks its directories to see if any client has reconstructible copies of page P. Directory entries for 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. Figure 2 depicts the case where A caches a reconstructible fragment of page P.

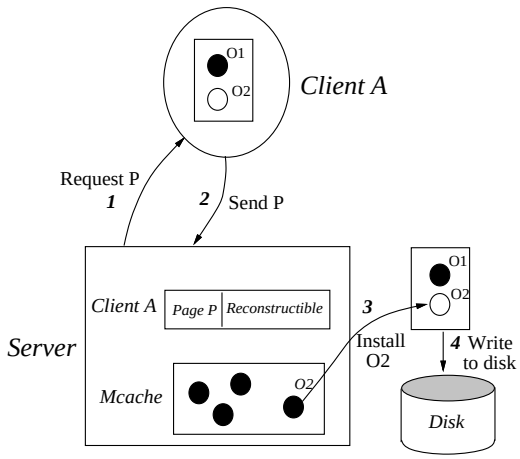

Figure 2: Installation read avoidance

3.4 Optimizations

Now we discuss some optimizations to the protocol. Our first optimization avoids pages from 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 can send 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. Note that 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 after many updates 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 numbers the committing transactions, furthermore for each entry in a client's directory, it stores the number of the latest transaction whose modifications are reflected in the client's cached copy. We rejected such a 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 Correctness of Fragment Reconstruction

Demonstrating the correctness of the fragment reconstruction protocol is a good way to understand the protocol itself, and is an interesting topic in its own right. We now discuss this issue informally (a detailed proof appears in [13]). Our basic approach is to establish an equivalence between the recoverable states of two systems: the *basic* Thor system configured (for simplicity) with a zero-size page cache at the server and the *extended* Thor system using fragment reconstruction. We show that each system satisfies its

own simple invariant, and that the extended invariant implies the basic invariant. As a result, we show that any sequence of committed transactions will produce the same recoverable states in both the basic and extended systems, so the two systems "appear the same" to any sequence of application transactions. It follows that the extended system is a correct optimization of the basic system.

In the basic Thor system, the mcache-based database update protocol guarantees the *basic invariant* that at any point in a system execution sequence that contains a given set of committed transactions, all committed updates have either been propagated to the database on disk or reside in the (recoverable) mcache. It is easy to see that the mcache-based database update protocol that 1) obtains a page P from the database, 2) installs all updates to page P from the mcache, 3) writes the updated page P to the database, and 4) removes updates to page P from the mcache, maintains the basic invariant.

The extended Thor system uses pages from the client caches and modifications in the mcache to update the database disk. The *extended invariant* for the extended system needs to guarantee that if the cache of client A provides a page to avoid an installation read then this page and the mcache combined include all the committed updates. Since a system satisfying the extended invariant also satisfies the basic invariant, the extended invariant insures that when a system with fragment reconstruction uses the cached copy of the page to update the disk, this has the same effect as reading the page from disk.

The extended invariant presented below constrains how the page status information recorded in the server page directories reflects the committed updates contained in the mcache and in pages cached at the clients. It highlights the point that when a cached page P is forwarded to another client or server, a clean copy of a page is provided by the fragment reconstruction protocol. A clean copy contains updates of committed transactions, and holes corresponding to invalidations caused by commits of transactions of other clients, but does not contain updates of aborted transactions and also does not contain updates of uncommitted transactions.

> The Extended Invariant:
>
> If a page p cached at client A is marked as complete in the directory, then either the client A has discarded page p (or some object on page p) and this information has not yet been recorded in the directory, or client A has a clean version of page p that contains all the committed updates to page p and no uncommitted updates.
>
> If a page p cached at client A is marked as reconstructible in the directory, then either client A has discarded page p (or some object in p) and this information has not yet been recorded in the directory, or client A has a clean version of page p that when combined with the committed updates to page p residing in the mcache contains all the committed updates to page p and no uncommitted updates.

For simplicity, assume that client caches maintain clean copies of modified pages that do not include modifications of the current transaction. (The implementation optimizes cache space and only maintains clean copies of modified objects; it reconstructs a clean page when asked to forward the page to another client or return it to the server.) The optimistic concurrency control protocol in [1] insures that when a client aborts a transaction all uncommitted effects in the client cache are undone. By examining the coherence protocol actions it is easy to see that the extended invariant is preserved by the directory update operations resulting from client and server fetches, transaction commits, and database updates.

5 Related Work

Our work on the fragment reconstruction coherence protocol builds on techniques for supporting fine-grained sharing and techniques for providing global caching. To put our work in perspective, we consider related work on global caching in file systems and transactional storage systems, and work on avoiding false sharing problems in distributed shared memory systems.

Work in the xFS file system [14] explores global caching in a scalable serverless storage architecture. The xFS coherence protocol improves on earlier file-level coherence protocols by supporting disk block level sharing. In contrast, the fragment reconstruction protocol supports sharing of fine-grained objects.

Franklin et al. [7] studies global caching in a client/server database. Clients interact with each other via servers. Servers redirect fetch requests between clients. The study evaluates several global memory management algorithms. This work is similar to ours in that it considers a transactional database. The difference is that Franklin et al. consider coarse-grained page-based concurrency control and do not address the problem of false sharing. Another difference is that their system uses locking while our system uses optimistic concurrency control.

A coherence scheme similar to fragment reconstruction is described in [5]. In this scheme, the database server manages a cache of recent updates. Before a client accesses an object, it contacts a server to retrieve the updates needed to bring the cached copy of the object up to date. This scheme does not support global caching, i.e., clients can not fetch pages from other clients. In addition, it is is based on locking, whereas fragment reconstruction is based on optimistic concurrency control and invalidations.

Work in distributed shared memory systems has addressed the problems caused by false sharing. These systems allow concurrent accesses to shared pages by supplying synchronization primitives to support weaker consistency models, e.g., release consistency [11] in Munin, lazy release consistency [10] in TreadMarks, and multiple writer protocols [4] in both Munin and Trademarks. This work differs from our work because it does not consider transactional updates to persistent data.

The distributed shared memory coherence work by Feeley et al. [15] is similar to ours because it considers transactional updates. This approach assumes a single-server, main-memory-based transactional store. Transactions use distributed locking. To keep caches coherent, at transaction commit time, clients propagate the commit log containing fine-grained updates to other clients. This approach does not consider global caching. The coherence protocol by Feeley et al. needs to propagate updates to cached pages at commit time to tolerate client failures. In contrast, our mcache-based coherence protocol propagates updates lazily and benefits from update absorption; furthermore it works correctly even in the presence of client failures.

6 Conclusions

This paper presents the new transactional fragment reconstruction coherence protocol integrated with Thor's mcache. The fragment reconstruction protocol supports fine-grained sharing in split caching object storage architecture. It is the first fine-grained coherence protocol that supports global caching in a transactional storage system.

The fragment reconstruction coherence protocol is attractive because it is simple. In addition, it is applicable to object storage architectures different from Thor. In particular, fragment reconstruction could be used in a more traditional caching architecture where servers cache pages. It could also be used with different concurrency control mechanisms although the concurrency control mechanism affects how coherence works, e.g., a system that uses locking [2] might have to send a page with holes in response to a fetch (since some of its objects are locked

right now).

The paper presents the correctnes invariant for fragment reconstruction protocol that guarantees that the protocol preserves the correctness of the concurrency control and recovery protocols in Thor. This is important, since it ensures that as far as application transactions can tell, the system with fragment reconstruction is a correct optimization of the current Thor system.

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

References

[1] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. In *International Conference on Management of Data*. Association for Computing Machinery SIGMOD, June 1995.

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

[3] 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 International Conference on Management of Data*, pages 357–366, 1991.

[4] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Techniques for reducing consistency-related communication in distributed shared memory systems. *To appear in ACM Transactions on Computer Systems*, August 1994.

[5] A. Delis and N. Roussopoulos. Performance and scalability of client-server database architecture. In *Proceedings of the 18th Conference on Very-Large Databases*, 1992.

[6] M. Feeley, W. Morgan, F. Pighin, A. Karlin, H. Levy, and C. Thekkath. Implementing global memory management in a workstation cluster. In *SOSP*, 1996.

[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, Massachusetts Institute of Technology, 1995.

[9] R. Gruber. *Optimism vs. Locking: A Study of Concurrency Control for Client-Server Object-Oriented Databases.* PhD thesis, Massachusetts Institute of Technology, 1996.

[10] P. Keleher, A. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *ISCA*, May 1992.

[11] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Henessy. The directory based cache coherence protocol for the dash multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, 1990.

[12] 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 1994 ACM SIGMOD*, 1996.

[13] L.Shrira. A Correctness Proof for Fragment Reconstruction. In *umpublished manuscript*, 1996.

[14] M.D.Dahlin, R.Y. Wang, T.E.Anderson, and D.A. Patterson. Cooperative caching:using remote client memory to improve file system performance. In *Proceedings of Operating Systems Design and Implementation*, 1994.

[15] M.Feeley, J.Chase, V.Narasayya, and H.Levy. Integrating coherency and recoverability in distributed systems. In *Usenix Symposium on Operating System Design and Implementation*, 1994.

[16] 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.

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

[18] Seth J. White and David J. DeWitt. Implementing crash recovery in quickstore: a performance study. In *ACM SIGMOD International Conference on Management of Data*, pages 187–198, 1995.