Highly Concurrent Cache Consistency for Indices in Client-Server Database Systems

Markos Zaharioudakis
University of Wisconsin
markos@cs.wisc.edu

Michael J. Carey
IBM Almaden Research Center
carey@almaden.ibm.com

Abstract

In this paper, we present four approaches to providing highly concurrent B^+-tree indices in the context of a data-shipping, client-server OODBMS architecture. The first performs all index operations at the server, while the other approaches support varying degrees of client caching and usage of index pages. We have implemented the four approaches, as well as the 2PL approach, in the context of the SHORE OODB system at Wisconsin, and we present experimental results from a performance study based on running SHORE on an IBM SP2 multicomputer. Our results emphasize the need for non-2PL approaches and demonstrate the tradeoffs between 2PL, no-caching, and the three caching alternatives.

1 Introduction

Modern object-oriented database systems (OODBMSs) are typically based on a data-shipping approach. Data items are shipped from servers to clients so that query and application processing can be performed at the client workstations. As a result, data-shipping offloads DBMS function from the server to the clients, enabling the powerful client CPUs to be exploited. In addition to client CPUs, client memories can also be utilized effectively by caching the data shipped from the servers. Having the data available at the clients can reduce the number of client-server interactions, thus shortening the application pathlength and further offloading the server resources (both CPUs and disks). However, due to data caching, copies of data items may reside in multiple client caches. As a result, replica management is required, in addition to concurrency control, to ensure that all clients see a consistent (serializable) view of the database. In this paper, we will use the term *cache consistency* to refer to concurrency control and replica management together.

The data-shipping approach is well suited for object-oriented applications that support navigation through complex persistent data structures. Such applications often exhibit locality of reference, and they usually apply CPU-intensive methods on the objects that they access. In this domain, data caching has been shown to offer significant performance gains despite the potential overheads associated with cache consistency protocols [Wilk90, Care91, Wang91, Fran92]. However, in addition to navigation, OODBMSs support associative queries on large collections of objects as well. Indices are critical for the efficient processing of such queries, as they provide fast paths to the user objects.

In this paper we will focus on B^+-tree indices. With respect to cache consistency, most current OODB products treat B^+-trees like user data, performing 2-phase locking on index pages. Although this approach offers great simplicity, it is overly strict, as indices have more relaxed consistency requirements than data. Furthermore, indices are often the "hot-spots" of the database and, as a result, 2-phase locking (2PL) can lead to high contention. Thus, there is a need for advanced protocols that provide high index concurrency. This need was recognized early in the context of single-site DBMSs [Baye77], and there is now a rich variety of highly concurrent centralized B^+-tree algorithms [Lehm81, Sagi86, Shas88, Moha89, Moha90, Lome93]. However, the problem of B^+-tree cache consistency in the context of client-server OODBMSs has received little attention to date. In this paper, we present four alternative approaches that provide high index concurrency in this context. All of our approaches, as well as the 2PL approach, have been implemented in the SHORE OODB system [Care94]. In order to explore the relative tradeoffs between these alternatives, we conducted an experimental performance study by running SHORE on an IBM SP2 shared-nothing parallel machine; results from this study are presented later in this paper.

1.1 Design Overview

In designing our index cache consistency alternatives we tried to satisfy the following five (sometimes conflicting) goals. (1) High concurrency – our distributed algorithms should maintain the high concurrency achieved by modern centralized B^+-tree algorithms. (2) Local operation – B^+-tree operations should be performed at the clients, as much as possible, so that the servers are offloaded and the scalability of the system is improved. (3) Efficiency – the cache consistency overhead should be low. A good measure of this overhead is the number of messages related to cache consistency actions, so we would like to minimize the number of such messages. (4) Simplicity – our algorithms should be simple and practical. (5) Completeness – in addition to ensuring the structural consistency of the indices (i.e., the definition of a B^+-tree is never violated), our algorithms should also take transactional consistency requirements into account. For example, if degree 3 consistency is required, a B^+tree algorithm should support repeatable reads and phantom avoidance. Furthermore, transaction rollback and crash recovery should be handled correctly.

Four highly concurrent index cache consistency algorithms will be studied here. The first algorithm does not allow client caching of B^+-tree pages. Instead, it takes a *function-shipping* approach: All B^+-tree operations, which are invoked at the clients, are propagated to the server. The

SIGMOD '97 AZ,USA

server performs the index operations and returns the results, if any, to the requesting clients. The second algorithm follows the *data-shipping* approach, allowing the caching of index pages and performing both index read operations and index updates at the clients. Under this algorithm, client-server interactions during B^+-tree operations take place in terms of low-level physical requests (e.g., requests for locks, latches, or missing B^+-tree pages). Our last two algorithms adopt a *hybrid* approach that combines function-shipping and data-shipping. The hybrid algorithms allow caching of leaf pages only, and only use the cached leaves for index reads. If a read operation cannot complete locally (due to a missing leaf), the client propagates the operation to the server, which performs the operation and then ships the missing leaf to the requesting client. In order to utilize the cached leaves for reads, each client builds and maintains private trees on top of the cached leaves. In contrast to reads, B^+-tree updates are always immediately propagated to the server. The two hybrid algorithms differ from each other in that one employs the idea of *relaxed index consistency* [Gott96, Moha95], allowing clients to access potentially inconsistent cached leaves under some conditions.

The rest of this paper is organized as follows. Section 2 establishes the background necessary for understanding the description of our algorithms. Section 3 presents the five alternative algorithms themselves. Section 4 surveys related research work. Section 5 presents the results of a performance study of the algorithms. Finally, Section 6 summarizes our conclusions.

2 Background

As mentioned earlier, SHORE served as our implementation platform. SHORE is based on a peer-servers architecture that generalizes the more traditional client-server model. Although the peer-servers model provides a number of advantages [Care94], it makes algorithmic descriptions somewhat cumbersome, as each peer-server may act both as a client and as a server. For simplicity, we will use the more conventional client-server terminology in this paper – we will assume that there is a single server and a number clients, each running at most one application at a time[1]. The rest of this section describes the cache consistency protocol used for SHORE data objects, and gives a brief overview of SHORE's centralized B^+-tree algorithm that served as the basis for our highly concurrent distributed protocols.

2.1 Cache Consistency for Data Objects

SHORE uses the PS-AA protocol [Zaha96] to provide object-level cache consistency for user objects. PS-AA is based on *callback locking* [Howa88, Lamb91, Fran92]. Therefore, it guarantees that copies of objects in client caches are valid, and can be read without server intervention; only a local shared (SH) lock is needed on a cached object before reading it. In SHORE, caches are page-based. However, as we will see shortly, individual objects inside a cached page may be marked as "unavailable". Thus, an object is actually cached at a client if its containing page is cached and the object is not marked as "unavailable". To read an object that is not locally cached, a client requests the object from the server. The server first requests an SH lock on the object. When the lock is granted, the server ships to the requester a copy of the *page* that contains the requested object. However, before shipping the page, the server marks as "unavailable" any objects in the page that are currently write-locked by other clients. The server also updates its *copy table*, which is used to keep track of the locations of cached pages. When the requesting client receives the page from the server, it places it in its cache. If the page is already cached, however, and the cached copy contains any uncommitted updates, the client merges the incoming copy with its cached one (so that it will not overwrite its own dirty objects).

To update an object X, a client A must always acquire the server's permission. To do so, the client sends an exclusive (EX) lock request to the server. The server obtains the EX lock on X and then issues *callback* requests to all clients (except client A) that have a cached copy of the page P that contains X. At a client B, such a callback will either invalidate (actually purge) the whole page P or invalidate the individual object X by marking it as "unavailable". Full page invalidation occurs only if client B does not hold locks on any of the objects in P. If client B holds a lock on object X itself, then a "callback blocked" reply is sent back to the server (to be used for deadlock detection) and the callback waits for client B to commit (or abort). When the callback completes, it sends an acknowledgement to the server. The server waits for all the callbacks to complete and then grants update permission to client A. If all the callbacks were successful in invalidating page P, the server grants an EX *page* lock to client A. In such a case, client A can update any object in P without further server intervention; it only has to obtain local object-level EX locks to keep track of the objects it has updated. If another client later wishes to access page P, client A will be asked by the server to downgrade its EX page lock and obtain individual EX object locks at the server. When a client commits, it first installs its updates at the server and then releases all of its locks. If a client aborts, it invalidates all of its cached dirty objects; any updates that reached the server prior to the abort are undone by the server.

2.2 Centralized B^+-tree Management

Typical centralized B^+-tree managers provide support for four operations: Insert, Delete, Scan_Init, and Scan_Next. Insert and Delete are used to insert or delete a given B^+-tree entry (where an entry is a key-value/object-id pair). Scan_Init and Scan_Next are used to perform a range scan. Scan_Init is given the range specification and locates the first qualifying entry. Scan_Next returns the next qualifying entry each time it is called. The rest of this section gives an overview of SHORE's approach to handling these operations, assuming that transactions require degree 3 consistency. We should note that SHORE's B^+-tree management is largely based on the ARIES/IM algorithm [Moha89].

Typically, to update a B^+-tree or to initialize a scan, a path is first traversed from the tree root to the target leaf for the operation. In ARIES/IM (and in SHORE), this traversal is performed using latch-coupling with SH latches[2]. No tree traversal is required to get the next qualifying entry during a range scan, as Scan_Next remembers the location (leaf and slot within leaf) of the last retrieved entry and proceeds horizontally (left-to-right) from that location to find the next entry. However, if the leaf remembered by Scan_Next has been updated, Scan_Next re-searches that leaf to re-locate the last retrieved entry (if possible) and

[1] This way we will be able to talk in terms of clients rather than transactions; for example, we will say that a client A is reading an object X, meaning that the actual transaction currently running at A is reading that object.

[2] Latches can be thought of as short-term, page-level locks.

proceed from there; if the last retrieved entry is not in the remembered leaf anymore, Scan_Next re-initializes the scan by calling Scan_Init (starting from the last retrieved entry). In any case, after the appropriate leaf is located, it is latched in SH mode for scans or in EX mode for updates. Next, for degree 3 consistency, a lock must be acquired on the relevant B^+-tree entry[3]. Specifically, for range scans, the entry to be retrieved must first be locked in SH mode. Furthermore, after the last qualifying entry is retrieved, one additional SH lock must be acquired on the next entry. For updates, an EX lock must be acquired on the first entry that is greater than the one being inserted or deleted (we call this the "next entry"). Finally, we note that the EX lock requested during insertions is an *instant*[4] one, in contrast to the commit-duration locks required for deletions and scans.

```
1. Insert(Root, NewEntry)
2. {
3.     Leaf = traverse(Root, NewEntry);
4.     latch(Leaf, EX);
5.     NextEntry = find_next_entry(Leaf, NewEntry);
6.     while (1) {
7.         lock_instant(NextEntry, EX, no_block);
8.         if (lock not granted immediately) {
9.             unlatch(Leaf);
10.            lock(NextEntry, EX, block); /* try again */
11.            latch(Leaf, EX);
12.            if (Leaf still correct leaf for insertion) {
13.                oldNext = NextEntry;
14.                NextEntry = find_next_entry(Leaf, NewEntry);
15.                if (oldNext != NextEntry) {
16.                    release lock on oldNext;
17.                    continue; /* back to line 6 */
18.                }
19.            } else {
20.                goto line 3;
21.            }
22.        }
23.        break;
24.    }
25.    release lock acquired in line 10, if any;
26.    insert NewEntry in Leaf;
27.    unlatch(Leaf);
28. }
```

Figure 1: B+-tree Insertion Protocol (ARIES/IM)

In addition to using latches for index pages (rather than 2-phase locks), ARIES/IM improves concurrency further by releasing such latches if it must wait for a lock. This technique also prevents deadlocks that involve latches. However, it makes B^+-tree operations more complex, as illustrated in Figure 1, which shows the insertion protocol in more detail. (Figure 1 is still somewhat simplified; for example, it does not show how page splits are handled.) As shown, after the target leaf is located, it is EX latched and searched to find the next entry. Next, an instant, "non-blocking" EX lock is requested on that entry. Non-blocking means that if a lock conflict exists, the lock call will return immediately rather than blocking the requesting transaction. If there is no conflict, the insertion completes by putting the new entry into the leaf and then unlatching it. Otherwise, the leaf is unlatched and a regular EX lock is requested on the next entry. After the lock is eventually granted, the leaf is re-latched and a check is made to see if it is still the correct leaf for the insertion. If so, a second check is made to see if the next entry is still the same. If both checks succeed, the lock is released and the operation completes as before. If the leaf is still valid but the next entry has changed, the operation restarts from line 6 in Figure 1. If the leaf itself is not valid, the operation restarts from the beginning (line 3). Notice that the leaf check in line 12 will succeed if the leaf did not change while waiting for the lock, or if the new entry is *bounded* within the leaf, i.e., it will not become the first or the last entry in the leaf.

ARIES/IM and SHORE differ in how they handle structure modification operations (SMOs), i.e., page splits and page deletions. When a page is split or deleted, its parent must be updated. ARIES/IM propagates SMOs bottom-up when they occur, i.e., as part of the B^+-tree operation that caused the SMO. In contrast, SHORE takes a lazier approach, allowing unpropagated SMOs to linger until the SMO is encountered later during a top-down traversal. SHORE ensures that trees are always traversable, even given unpropagated SMOs, by linking the pages at *all* tree levels (not just the leaves) horizontally as in [Lehm81].

3 Index Cache Consistency

The problem of index consistency in a data-shipping, client-server system can be decomposed into several dimensions, including: (a) what kind of index pages are cached at the clients (e.g., none, leaves only, or both leaves and interior nodes), (b) which index operations are performed at the clients (vs. handed over to the server), (c) at what level do clients interact with the server (e.g., B^+-tree function shipping vs. low-level physical requests), and (d) whether strict or relaxed index consistency is used. In Section 1.1 we mentioned four different algorithms that we have designed along these dimensions. All four algorithms provide high index concurrency by extending SHORE's variant of ARIES/IM. We have also implemented an algorithm that performs 2-phase locking on index pages, essentially treating indices like data. Below we describe each of these algorithms in more detail. Due to space restrictions, certain important implementation issues, such as logging, transaction rollback, and crash recovery issues will not be explicitly discussed.

3.1 The 2-Phase Locking Approach (2PL)

During B^+-tree operations, 2PL traverses the tree acquiring commit-duration locks (instead of short-term latches) on the tree pages that it visits. As a result, 2PL does not need to acquire any object locks during index operations for degree 3 consistency. For example, in the case of an index update, the EX lock acquired on the updated leaf guarantees that no other active transaction has performed or will perform a range scan through that leaf until the updating transaction finishes. (Of course, transactions acquire locks on the objects that they access, but object locking is not part of index operations as in ARIES/IM.) In its client-server version, 2PL allows client caching of B^+-tree pages and performs both scans and updates at the clients. For cache consistency, our implementation of 2PL uses a page-level callback-locking protocol (it uses the PS algorithm of [Zaha96]). Therefore, clients can read locally cached index pages without server intervention. Missing index pages are, of course, fetched from the server. To update an index page, a client must always request an EX page lock from the server. In this case, the

[3] To be precise, the lock is acquired on the object pointed to by the B^+-tree entry.

[4] Instant locks are used to check for any current lock conflicts and to make the requesting transaction wait if such a conflict *is* detected. However, no actual lock is retained on the data item.

server obtains the requested lock and then calls back the page from all other clients that are caching it.

3.2 The No-Caching Approach (NC)

NC is the simplest of our four highly concurrent index consistency algorithms. As its name implies, NC does not allow client caching of index pages. As a result, all B^+-tree operations are executed at the server using SHORE's centralized B^+-tree algorithm. When the server receives a Scan_Init request, it actually pre-computes the whole scan. The server then replies to the requesting client with the complete list of qualifying entries. The client caches this list and uses it to answer subsequent Scan_Next requests without further server intervention. Notice that the list of qualifying entries cannot change before the client commits, as the objects in the list are locked at the server during the pre-computation of the scan. After the client retrieves the last qualifying entry, the cached list is discarded.

As described in Section 2.2, to perform an insertion, the server must acquire an instant EX lock on the next entry. According to Section 2.1, obtaining an EX lock on an object X involves sending callbacks to all other clients that cache the containing page of X. However, no such callbacks are required under NC for the next-entry instant EX lock. The following argument explains why. The purpose of this lock is to ensure that no other client has performed a range scan that would include the new entry if it were repeated (as 3 consistency must guarantee repeatable reads). Therefore, the purpose of the instant EX lock is to check for conflicts with SH locks that were acquired during scans. However, since scans are always performed at the server, scan-related SH locks are always acquired at the server. As a result, it is sufficient to request the instant EX lock at the server only. Deletions are similar to insertions. However, the next-entry EX lock required during deletions is a commit-duration lock, thus giving the client permission to update that object later if it wishes. As a result, during deletions the server *must* send lock callbacks to any qualifying clients to ensure that the next-entry object has not been accessed there at all.

3.3 The Full-Caching Approach (FC)

The most straightforward approach to highly concurrent index caching would be for each client to run SHORE's version of ARIES/IM as if the tree were local, and to rely on the underlying lock and cache managers to provide the necessary pages, locks, and latches from the server. However, such an approach has a high message overhead. For example, a client running the insertion algorithm of Figure 1 would have to send at least three messages to the server: one for the EX latch on the leaf, one for the EX lock on the next entry, and one at the end to release the EX latch and install the leaf update at the server. The FC algorithm described below optimizes this approach in two ways. First, FC postpones the installation of index updates at the server until commit time and does not retain any EX latches at the server, thus eliminating the need for explicit latch-release messages. Second, FC reduces the message count further by using messages that carry combined lock and latch requests. We will first describe FC's latching protocol, ignoring locks for a while, and then we will explain FC's multi-request messages in the context of the Insert operation.

Like PS-AA, FC is a callback-based algorithm. Therefore, a cached B^+-tree page can be locally SH-latched and read without server intervention. When the server receives a client read request for a missing B^+-tree page, it first latches[5] the page in SH mode. Then, if it has the latest version of the page, it ships it to the client and releases the latch. Otherwise, the server sends an *ownership* callback to the *owner* of the page, i.e., to the client that updated the page last. At the owner, the callback is treated as a request for an SH latch on the page. After the latch is granted, the owner releases its ownership of the page, and ships a copy of it back to the server. The server installs this up-to-date copy in its own buffer pool, and then ships it to the requesting client and releases the SH latch. To update a B^+-tree page, a client A must get the server's permission unless it is already the owner of the page. If the client is not the owner, it requests an EX latch from the server. The server first obtains the latch and then sends *latch* callbacks to all the other clients that are caching the page. At a receiving client, a latch callback will purge the page after making sure that it is not in use (i.e., latched) there. If the receiving client is the current owner of the page, a copy of the page will be included in the callback reply as well. After all the callbacks complete, the server marks client A as the new owner of the page, releases the EX latch, and grants update permission to client A. Finally, when a client commits, it sends to the server copies of the dirty index pages that it owns, thus transferring the ownership of these pages to the server.

Let us now describe how FC performs an insertion. FC's insertion protocol is a distributed version of the one shown in Figure 1. A client A first traverses the tree, fetching any missing index pages from the server, as described above. The leaf L reached by the traversal is latched locally, and the next entry is located. Let X be the next-entry object and P be its containing page. If leaf L is currently owned by client A and the client already holds an EX lock on object X (or page P), the insertion completes locally. Otherwise, the local latch on leaf L is released and a combined EX-latch/EX-lock request is sent to the server. The server obtains the requested latch and lock (making sure that the latch is not held if the lock request has to wait). The server then sends latch and lock callbacks to all other clients that are caching either leaf L, page P, or both. Latch and lock callbacks that have the same destination are sent within the same message. At a receiving client, the latch and lock callbacks behave as described before, but with one difference: the lock callbacks do not need to invalidate object X or page P, as the EX lock on X will not (normally) be retained. If no lock conflict is detected during the callback operation, the server marks client A as the new owner of leaf L, releases both the latch on L and the lock on X, and sends its reply to client A. If, however, a lock conflict is detected, then the server, upon receiving the "callback-blocked" message from the conflicting client, releases the leaf latch and waits until the conflict expires. The server will then re-latch the leaf and repeat the latch/lock callback operation. During this second round, lock callbacks will act in their "normal" way, i.e., they *will* invalidate object X (or page P). After the second callback round is done, the server sends its reply to client A, but this time it does not release the lock on object X. The server may also include a copy of leaf L in its reply, if client A is not caching L any more. (The cached copy of leaf L at client A may get called back if another client is also trying to update L at the same time.)

When client A gets the server's reply, it puts the new

[5] Actually, the server uses locks (instead of latches) to control access on B^+-tree pages without having to read them from disk. This is because latches are associated with buffer frames, so to latch a page it must be resident in the buffer pool. In describing FC we will talk in terms of "latches" in order to distinguish the 2-phase locks acquired on user data from the short-term physical locks used for index pages.

copy of leaf L (if any) in the cache and EX-latches the leaf locally. Next, it checks whether L is still the correct leaf for the insertion and whether the next entry is still the same. If any of these checks fails, the client will restart the operation, either from the root or from leaf L, as shown in Figure 1. Otherwise, the insertion completes locally. Notice that in the case of a lock conflict, the client does not attempt to release the lock on the next-entry (i.e., it skips line 25 of Figure 1) because this would require one more message.

3.4 Hybrid Caching (HC)

In general, the FC algorithm may require multiple client-server interactions to perform a single B^+-tree update. Furthermore, FC is a rather complicated algorithm[6]. These concerns motivated the design of the hybrid caching algorithms described in this section. Like NC, hybrid caching takes a function-shipping approach – clients propagate some whole B^+-tree operations to the server. Unlike NC, HC supports caching of leaf pages, and it uses the cached leaves to perform scans at the clients. We have designed two algorithms under the HC approach. The first uses strict consistency, meaning that clients always access the latest versions of B^+-tree leaves. The second is more relaxed, allowing clients to access out-of-date cached leaves as long as the entries retrieved from there are the "correct" ones.

3.4.1 Strict Hybrid Caching (HC-S)

In HC-S, a client accesses its cached leaves of a B^+-tree through a local, in-memory index that it builds on top of these leaves. Figure 2 shows two clients, each caching two leaves of a tree stored at the server. As shown, cached leaves and their corresponding server versions are the same. In contrast, internal tree nodes across different sites are not related. To perform a Scan_Init operation, a client first traverses its local tree to a cached leaf L. (If there is no cached leaf, the Scan_Init operation is propagated to the server.) Next, the client checks whether the starting key value is *bounded* within leaf L, i.e., if the starting key value is less or equal to the last key value in L, and greater than the first key value in L. If the check succeeds, the client has indeed reached the correct leaf and can proceed by retrieving the first qualifying entry after locking it in SH mode. (The lock may be requested from the server if the associated object is not locally cached; the next sub-section explains this case in more detail.) If the starting key value is greater than the last key value in L and the right sibling of L is cached, the client will visit that sibling and find the desired entry there (or determine that there is no qualifying entry). In all other cases, the client propagates the Scan_Init operation to the server, which performs the operation and returns to the client the leaf containing the first satisfying entry. When the client receives the leaf, it installs it in its local B^+-tree and retrieves the desired entry. To install a new leaf into the local tree, the client traverses the tree top-down in order to find the correct place for the leaf. Scan_Next is performed in a similar manner. The client accesses the last remembered leaf, if that leaf is still cached. If not, the client propagates the Scan_Next operation to the server. The operation is also handed over to the server if the right sibling of the last remembered leaf needs to be accessed and that leaf is not locally cached. The server will perform the Scan_Next operation and return a leaf to the client.

As mentioned earlier, HC-S performs all updates at the server. After locating the target leaf, the server locks the next-entry and sends latch and lock callbacks for the target leaf and the next-entry object, respectively. Like FC, HC-S uses multi-request callback messages to combine the latch and lock callbacks in a single callback round. Also, in the case of insertions, lock callbacks do not invalidate any data except during a second callback round (required only if a lock conflict was detected in the first round). If an SMO occurs, the server will call back all the leaves that participate in the SMO. After an update is done, the server sends a copy of the target leaf back to the updating client. In the case of a leaf deletion, the server does not send any leaf to the requesting client; instead, it instructs the client to invalidate the deleted leaf (if cached) and to update the "next" and "prev" pointers of the neighbor leaves (if cached) accordingly. When a client receives a leaf that is already cached, it checks whether the boundary entries of the leaf have changed. If not, then the new copy overwrites the cached one. Otherwise, the local parent of the cached leaf may have to be updated; in this case, the cached copy is discarded and the new copy is installed into the tree in the usual top-down manner.

We end our description of HC-S with a few remarks regarding the local trees that clients build on top of cached leaves. The root of a client tree is created the first time that the client accesses the associated server tree. Additional client-tree nodes are created as new leaves arrive from the server and cause existing client-tree nodes to split. Purging a leaf from the client cache (either due to a latch callback or because of page replacement) is easy because, as described in Section 2.2, the parent client-tree node does not need to be updated immediately. A client-tree node gets destroyed when all of its children are destroyed or purged (if the children are cached leaves).

3.4.2 Relaxed Hybrid Caching (HC-R)

The relaxed index consistency technique was introduced in [Moha95, Gott96]. Here we apply this technique in the context of our hybrid caching approach. The resulting algorithm, HC-R, is similar to HC-S. In particular, range scans are performed exactly the same with both algorithms; differences arise only in the execution of updates. We will first describe these differences and then use two examples to give the rationale behind relaxed index consistency and our particular implementation of it. In Section 4 we will explain how HC-R differs from the algorithms in [Moha95, Gott96].

The first difference between HC-R and HC-S is related to the next-entry EX lock requested during insertions. As described above, HC-S does not retain this lock, and does not invalidate the next-entry object (or its page) from any remote caching client unless a lock conflict is detected. In contrast, HC-R *must* acquire a *commit-duration* EX lock on the next-entry, thus always invalidating the next-entry object (or its page). The second difference is that HC-R does *not* try to call back the target leaf of an update, unless an SMO takes place. If an SMO does occur, HC-R, like HC-S, will call back all the leaves involved in the SMO. Otherwise, HC-R will only remember the id of the target leaf. The information about which leaves have been updated by which clients is maintained at the server. Let S be the set of leaves that have been updated by a client A. When client A commits (or aborts), the server will call back all of the leaves in

[6] To a large degree, FC's complexity stems from logging and recovery issues. Such issues are much easier to handle with the hybrid caching approach because updates are executed at the server.

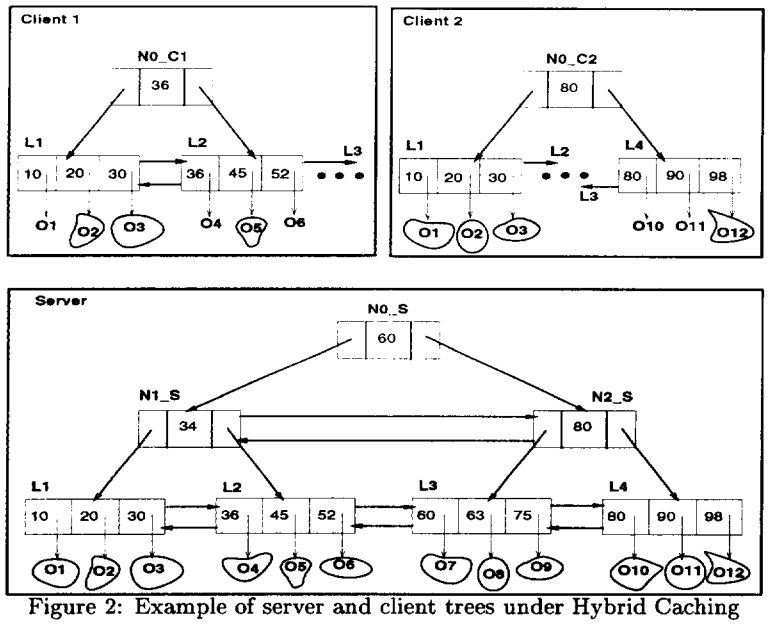

Figure 2: Example of server and client trees under Hybrid Caching

set S. This is done in a single callback round during which the server sends a callback request to any clients (other than A) that are caching any leaf in S; such a callback will invalidate all leaves in S that are cached at the receiving client.

We now turn to the first example that explains why relaxed index consistency actually works. Let us start with the situation shown back in Figure 2. As shown there, client A caches leaves L1 and L2 of a B^+-tree, and objects O2, O3, and O5; client B caches leaves L1 and L4 of the same tree, and objects O1, O2, O3, and O12. Now, let us assume that client A inserts a new entry, entry [25, O13]. Figure 3 shows the client and server states after the insertion is complete. We see that the new entry appears at client A and at the server, and that leaf L1 was not called back from client B; as a result, client's B copy of L1 is now out-of-date. However, object O3, the next-entry object, *has* been called back from client B and is now EX-locked by client A. (Locks are indicated in Figure 3 by shading the associated objects.) Next, supose that client B initiates a range scan for key values greater than 20. Client B traverses its local tree and locates entry [30, O3] in leaf L1 as the first qualifying entry, apparently missing entry [25, O13] which also qualifies. However, before actually retrieving entry [30, O3], client B must lock object O3; it is this lock that gives HC-R the opportunity it needs to correct things at client B.

Since client B does not cache object O3, it must request the SH lock on O3 from the server. Before doing so, client B releases its SH latch on leaf L1 in anticipation that the lock request may block at the server. In our example, the lock request will indeed block due to the EX lock held on O3 by client A. When client A commits, the server calls back leaf L1 from client B, and then releases all of A's locks. When client B finally receives the lock reply from the server, it tries to re-latch leaf L1, thus realizing that it is no longer cached. As a result, client B will restart the Scan_Init operation. During this second attempt, client B caches leaf L4 only, so, it will reach entry [80, O10]. Since 80 is greater than the scan's starting key value (20), the client will abandon its effort to perform the Scan_Init operation locally and it will hand it over to the server. The server will locate the correct entry ([25, O13]) in leaf L1, lock the associated object (O13) in SH mode, and ship a fresh copy of leaf L1 to client B. Client B will then install the leaf in its local tree and will finally retrieve the correct entry, even though it started with an out-of-date leaf. Notice that, in a different scenario, if client B had originally been interested in key values between 5 and 20, it would have been able to perform its scan entirely without server intervention using its cached version of leaf L1, despite it being an out-of-date version.

The above example shows that HC-R depends on commit-duration next-entry locking for correctness. As shown, client B avoided missing a newly inserted qualifying entry only because the inserting client A acquired a commit-duration lock on the next-entry. However, degree 3 consistency normally requires only an *instant* next-entry lock during insertions. Furthermore, lower degrees of consistency require more relaxed next-entry locking or no such locking. Therefore, although HC-R reduces the number of latch callbacks, it is rather strict with respect to data locking, especially when degree 3 consistency is not required. In contrast to HC-R, all the other highly concurrent alternatives can be easily modified to support lower degrees of consistency.

The second example explains why HC-R should not postpone calling back the leaves involved in SMOs until commit (or abort) time. Figure 4 shows what can go wrong if SMO-related callbacks *were* postponed. Client A caches leaf L, and initially, it has the same version of L as the server. (For simplicity, Figure 4 does not show the OIDs in the leaf entries). Suppose that client B inserts a new entry in L that forces the leaf to split and its entries to be re-distributed between L and its newly allocated sibling L'. Notice that we have let client B delay its callback of leaf L. As a result, leaf L becomes out-of-date at client A. Next, a second insertion is performed by client C. Client C inserts an entry with key value 11 in leaf L' and, as a result, exclusively locks the next-entry object, i.e., the object with key value 15. Finally, client A initiates a range scan for keys greater than 6. Client A uses its cached copy of L to (correctly) retrieve the entry with key value 10 as the first qualifying entry. The next qualifying entry, according to client's A version of L, is the one with key value 15. At this point client A will block because the entry is exclusively locked by client C. Suppose that client C commits before client B. Then, client

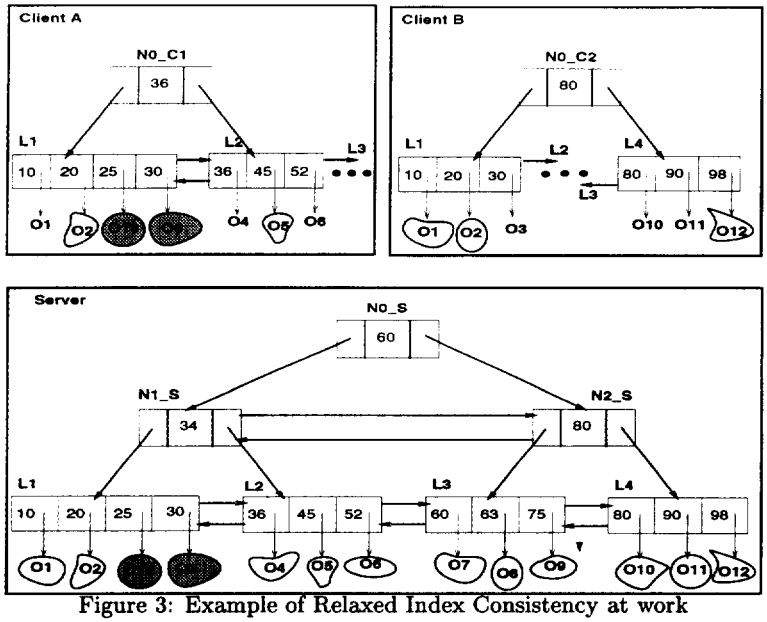

Figure 3: Example of Relaxed Index Consistency at work

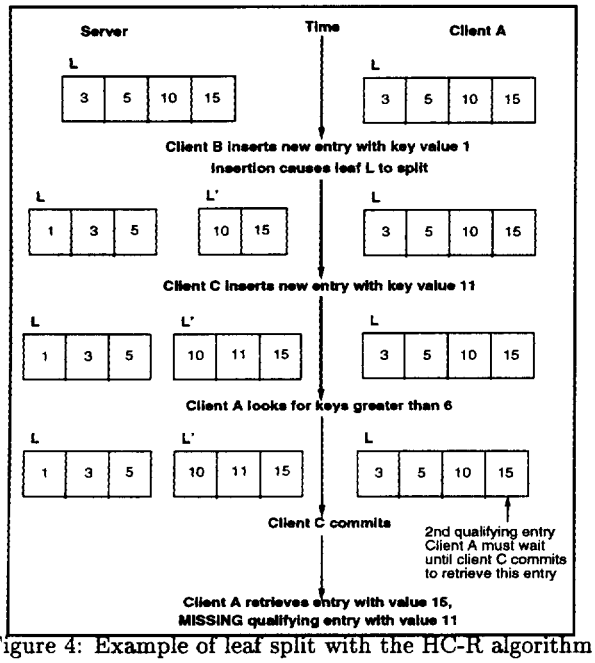

Figure 4: Example of leaf split with the HC-R algorithm

A will unblock and retrieve entry 15, thereby missing entry 11. The problem is that client C inserted a qualifying entry in a different leaf than the one scanned at client A. This can happen only due to a leaf split, and the problem is remedied in HC-R by immediately calling back leaves that split.

4 Related Work

The study that is most closely related to ours appears in [Gott96]. Two index cache consistency algorithms – called Callback Locking (CBL) and Relaxed Index Consistency (RIC) respectively – were investigated there. CBL is similar to our FC algorithm in many respects. Both support client caching of index pages, perform both updates and scans at the clients, use the same latching protocol based on callbacks and page ownership, and assume the use of callback-based cache consistency for user data. There are, however, several differences between the two algorithms as well. First, CBL does not use multi-request messages to combine lock and latch requests. Second, CBL uses *page*-level locking for user-data cache consistency, whereas FC supports fine-grained sharing via PS-AA (Section 2.1). As shown in [Zaha96], page-level locking can significantly increase data contention. CBL can lead to high index contention as well, as it allows leaf latches to be held during (potentially frequent) page-level lock conflicts. Finally, CBL does not realize the fact that the next-entry lock requested during insertions needs only be an instant one; with CBL, this lock is always a commit-duration lock, thus causing data invalidations through callbacks. To summarize, FC is based on CBL, but it optimizes CBL in many important ways.

RIC modifies CBL by applying the relaxed index consistency technique. The implementation of this technique in RIC is somewhat different than in our HC-R algorithm. In HC-R, clients expect to be informed through callbacks about out-of-date leaves. In contrast, RIC follows a detection-based approach; clients themselves are responsible for detecting out-of-date index pages. To do so, RIC maintains index coherency information, at both the server and the clients, in the form of index page version numbers. The server updates its index coherency information whenever it receives EX latch requests from the clients and clients receive fresh index coherency information from the server every time they ask the server for a data lock. In RIC, the server piggybacks (in lock replies) such information for every index in the database, even if a client's lock request is not related to an index operation[7]. Clients cache the version numbers that they receive from the server and use them to check the validity of their cached copies of index pages. Compared to HC-R, RIC is a fairly complicated algorithm due to the management of its version numbers and to the fact that RIC allows cached internal tree nodes to become inconsistent. As a result, additional measures are required for correctness, as inconsistent tree paths can lead to the wrong target leaves; the description in [Gott96] is somewhat under-specified in

[7]We should point, however, that RIC uses index-wide surrogate version numbers as well to often reduce the amount of index coherency information that has to be piggybacked.

this respect. In contrast, HC-R solves this problem by not caching internal nodes at all. RIC has a high space overhead as well due to its in-memory caching of version numbers. Finally RIC is likely to suffer when high index and/or data concurrency is required, as, like CBL, it uses page-level locking and does not release latches during lock conflicts.

Another relaxed index consistency algorithm appears in [Moha95]. The algorithm presented there is an extension of ARIES/IM for the shared-disk (SD) environment. In a SD system there is no server site; each participating site reads pages directly from the disks into its own memory cache. However, the system includes a server-like component called the "Global Lock Manager" (GLM) that coordinates the concurrent actions of the sites. A combination of callback-based and detection-based schemes were used in [Moha95] to provide cache consistency. Callbacks are used for the internal B^+-tree nodes, and a detection-based protocol is used for the tree leaves as well as for the user objects. According to this protocol, all object locks (both shared and exclusive) are acquired at the GLM. In addition to locks, the GLM maintains cache coherency information for data and leaf pages in the form of version numbers. As in RIC, the GLM piggybacks coherency information on lock replies. Although the [Moha95] algorithm offers valuable insight into the problem of index cache consistency, it is not directly applicable in our environment because a number of its details depend on the use of detection-based cache consistency for user-data objects. We chose a callback-based focus in SHORE because it has been shown elsewhere that callbacks perform better than detection in the context of client-server OODB systems [Fran92].

Related work in the context of client-server systems has also been done by Basu and Keller. [Basu95] presents two schemes – one "centralized" and one "distributed". In fact, these two schemes are over simplified versions of our NC and FC algorithms, respectively. Neither of the two schemes supports degree 2 or 3 consistency, as they do not perform any kind of locking during index operations. (Object-level locks are acquired outside the scope of index operations when the actual objects are accessed.) Furthermore, the distributed scheme uses a simple latching protocol (not based on page ownership) that involves extra messages at the end of index update operations for releasing latches at the server and for installing index updates in the server's buffer pool.

5 Experimental Performance Study

As detailed in Section 3, we now have the five index cache consistency alternatives running in the SHORE system. To explore the tradeoffs between these alternatives, we conducted an experimental performance study running SHORE on an IBM SP2 shared-nothing parallel machine. This section describes the experiments conducted and presents some of the obtained results.

5.1 Experimental Platform Configuration

Quantity	Value
Number of concurrent applications	10
Per-client cache size	100% or 25% of DB size
Server cache size	100% or 50% of DB size
Size of a disk page	4,096 bytes
Round-trip latency of small messages	1.5 msec (avg)
Service cost for a random disk read	12 msec (avg)

Table 1: Experimental Platform Configuration

Table 1 describes our experimental configuration. In all of our experiments we ran 10 concurrent instances of our application program. Applications create and execute transactions one after another according to workload specifications to be described shortly. To run each experiment we used 11 SP2 nodes, each running a SHORE peer-server process. One of the peer servers acted as a "real" server, managing a database stored on a local disk. The rest of the peer servers acted as clients; each was linked with one of the application programs. As shown in Table 1, we used two settings for the server and the client cache sizes: first, the server and the client caches were made large enough to store the whole database; next, the server cache size was halved and the client cache size was reduced by a factor of four. Table 1 also includes the round-trip message latency and the cost of disk reads, as measured in our SP2 system.

5.2 Workload Model

The access pattern for each application is generated using the workload model detailed in Table 2. As indicated in Table 2, the model consists of two basic workloads – UNIFORM and HOTCOLD – differing in their data and index sharing patterns. We start by describing the underlying database used by each workload; we then explain how the actual transactions are built in each case.

The database parameters are given in the top section of Table 2. In both workloads, the user-data portion of the database consists of 200,000 objects that have unique logical ids ranging from 1 to 200,000. Consecutively numbered objects are placed next to each other on the disk, with each disk page storing 20 objects. Each object consists of three integer attributes (plus a "dummy" attribute used for padding). The first attribute is the object's own logical id; the third attribute has a randomly selected integer value; finally, the value of the second attribute is chosen randomly from the range specified by the *HotSecondKeyRange* or the *ColdSecondKeyRange* parameter, depending on whether the object is "hot" or "cold". In UNIFORM, all objects are cold, as applications have no particular data affinity. In HOTCOLD, each application has an affinity for its own preferred set of database objects, directing most of its accesses to that specific set; in this case, an object is hot if it belongs to the preferred object-set of an application. Different applications have disjoint sets of hot objects. Furthermore, the hot objects of each application are tightly packed together on disk; thus, the hot regions of different applications are physically disjoint as well.

In addition to user objects, the database includes a number of indices. Each object is indexed by two indices – one clustered and one unclustered. Clustered indices are built on the primary key of objects, whereas unclustered indices index the objects on their second attribute. In UNIFORM, there is one clustered and one unclustered index, each spanning the entire set of user objects. The HOTCOLD database is somewhat more complicated; it has one clustered and one unclustered index per hot region, plus an additional pair of such indices on the common area that is cold with respect to all applications. In both workloads, the unclustered indices may contain duplicate key values: Given that the number of potential key values for each unclustered index (determined by the *HotSecondKeyRange* or the *ColdSecondKeyRange* parameter) is half the number of objects pointed to by the index, the average number of objects per key value is 2. Finally, we note that the unclustered indices have a space utilization of 60%, translating to an average of 84 entries

Parameter	Meaning	UNIFORM	HOTCOLD
Database Parameters			
NumObjects	Number of data objects in database	200,000	200,000
ObjectsPerPage	Number of data objects per page	20	20
HotAreaSize	Size of each hot region	-	10,000 objects
HotSecondKeyRange	Key-value range for the 2nd attribute of hot objects	-	1 to 5,000
ColdSecondKeyRange	Key-value range for the 2nd attribute of cold objects	1 to 100,000	1 to 50,000
Transaction Mix Parameter			
IndexProb	Prob. of creating index transaction	0.6	0.6
Index Transaction Parameters			
ScanProb	Prob. of performing range scan	0 or 1	0 or 1
ClusteredProb	Prob. of using clustered index	0 or 1	0 or 1
HotIndexProb	Prob. of an application using its hot index	-	1
ColdIndexProb	Prob. of using the common cold index	1	0
NumLookups	Mean number of key lookups per index lookup trans. (same for both clustered and unclustered indices)	100	100
ScanClustSize	Mean number of index entries scanned per clustered index scan	2000	2000
ScanUnclustSize	Mean number of index entries scanned per unclustered index scan	100	100
Navigational Transaction Parameters			
NavigSize	Mean no. of objects accessed per trans.	320	320
PageLocality	Mean number of objects accessed per page	4	4
HotAccProb	Prob. of accessing a hot page	0	0.8
UpdateProb	Prob. of updating an object	varies	varies
IndexUpdateProb	Prob. of updating the second attribute of an object	0 or 0.5	0 or 0.5

Table 2: Database and Transaction Parameters

per index leaf; the clustered indices have (by construction) a space utilization of 90%, with 126 entries per leaf.

We now explain how the workload's transactions are generated. There are two types of transactions in our model: "index" transactions that access objects through indices, and "navigational" transactions that access objects directly. During an experiment, a given application may generate both kinds of transactions; the percentage of index and navigational transactions per application is controlled by the *IndexProb* parameter. Each index transaction uses a *single* index to perform either a number of key-value lookups[8] or a single range scan. Thus, index transactions are subtyped further depending on the operations that they perform and the particular index that they use. The mixture of index transactions per application is controlled by the *ScanProb*, *ClusteredProb*, *HotIndexProb*, and *ColdIndexProb* parameters in Table 2. The *NumLookups*, *ScanClustSize*, and *ScanUnclustSize* parameters control the size of index transactions. Finally, we note that index transactions do not update the objects that they access.

The bottom section of Table 2 contains the parameters for the navigational transactions. The size of such transactions is controlled by two parameters: *NavigSize*, which specifies the average number of data pages directly accessed by a navigational transaction, and *PageLocality*, which specifies the average number of objects accessed per data page. In the HOTCOLD workload, applications that run navigational transactions direct 80% of their accesses to their own hot area; their other accesses go anywhere in the rest of the database. In UNIFORM, all data pages are accessed with the same probability. In both workloads, the probability that an accessed object will be updated is specified by the *UpdateProb* parameter. The *IndexUpdateProb* parameter is used to determine which attribute of an object is to be updated (if the object was chosen for an update); the choice is between the second and the third attributes of the object. An update of the second attribute leads to two updates (a deletion followed by an insertion) of the associated unclustered index in order to move the object from its old to its new position in the index. As navigational transactions may access any object in the database, they may also update any of the available unclustered indices; in contrast, the clustered indices are never updated.

[8]Key-value lookups are actually short range scans where the starting and the stopping key values are the same.

5.3 Plan of Attack

In the last section we defined navigational transactions and various types of index transactions. In general, applications may create and run any type of transactions during an experiment. However, as indicated by the settings in Table 2, we simplified the transaction mix in most of our experiments by restricting applications to running navigational transactions and a *single* type of index transactions. Specifically, we mixed navigational transactions together with index transactions that perform either (i) clustered index lookups, (ii) unclustered index lookups, (iii) a clustered index scan, or (iv) an unclustered index scan. In the HOTCOLD workload, where there are many clustered and unclustered indices, applications were set up to access their own hot indices only. Furthermore, in the experiments where a clustered index was used by the index transactions, navigational transactions did not perform any index updates (i.e., only the third, non-indexed attribute of the objects was updated). Each workload and transaction mix was run using both settings of cache sizes shown in Table 1. As a result, we ran a total of 16 experiments, each with a two-type transaction mix. Due to the lack of space, only a small but representative subset of our results will be presented here in detail. In all of the experiments, the performance metric is the average response time per transaction type.

5.4 UNIFORM Experiments

We begin with a UNIFORM workload with navigational transactions and index transactions that perform lookups in the unclustered index. Figures 5 and 6 show the response times for the two types of transactions in the main-memory case, i.e., when both server and client caches can hold the whole database. A general trend is that response time goes up as the write probability increases, since more updates bring more contention and more messages for requesting

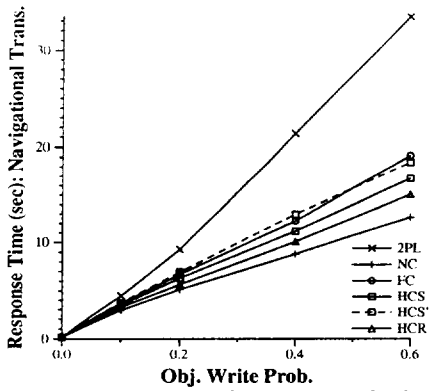

Figure 5: UNIFORM, Large Caches
Navigation + Unclustered Index Lookups

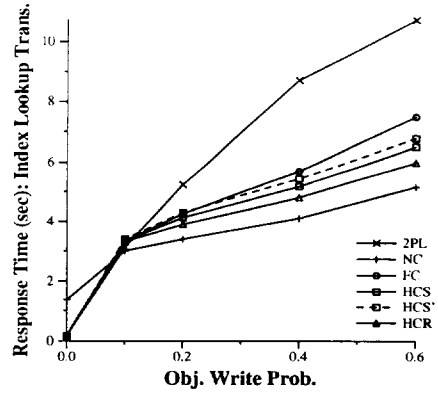

Figure 6: UNIFORM, Large Caches
Navigation + Unclustered Index Lookups

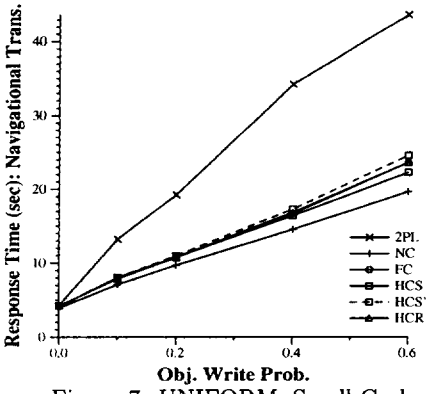

Figure 7: UNIFORM, Small Caches
Navigation + Unclustered Index Lookups

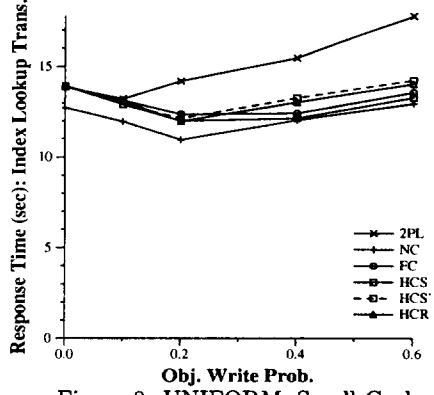

Figure 8: UNIFORM, Small Caches
Navigation + Unclustered Index Lookups

EX lock and latches, for performing callbacks, and for refetching invalidated data and index pages. We now focus on the performance differences between the five alternatives. We start by looking at the basic 2-phase locking (2PL) approach. At zero write probability there is no contention and the whole database is cached at each client; as a result, 2PL and the other three index caching alternatives perform the same. As the write probability increases, contention increases rapidly for 2PL due to its commit-duration locks on index pages; thus, 2PL soon becomes the worst approach.

Next, we compare strict hybrid caching (HC-S) with full-caching (FC). As shown in Figures 5 and 6, FC is somewhat worse than HC-S at high write probabilities. In this region, only a small portion of the index remains cached at each client, as index pages get invalidated at a high rate. As a result, FC suffers because it usually needs two messages to perform an index update (one message to fetch the missing leaf page, and another to request an EX latch on the leaf and an EX lock on the next-entry object). In contrast, HC-S, which propagates all index updates to the server, requires exactly one client-server interaction per index update. Although index updates are performed only by the navigational transactions, the extra messages sent by FC affect its performance for index transactions as well, as they increase contention for the server's CPU. (Due to the absence of any disk I/Os in this experiment, the server's CPU is the system bottleneck.) An additional source of FC's degradation are the ownership callbacks that are sometimes sent by the server during index page requests under FC.

We continue by comparing HC-S with the relaxed hybrid caching algorithm (HC-R), which allows clients to use out-of-date cached leaf pages. When there are no index updates the two algorithms are identical by design. As the write probability increases, HC-R gradually becomes better than HC-S, as it avoids many latch callbacks during index updates by postponing such callbacks until commit time, when they can be sent in a batch manner. Figures 5 and 6 show (dashed line) another version of HC-S that does *not* combine latch and lock callbacks during index updates; instead, it performs two sequential callback rounds, one for the leaf page and then another for the next-entry object. As shown, the modified HC-S algorithm performs somewhat worse than regular HC-S, illustrating the value of sending the latch and lock callbacks in parallel.

Finally, we turn to the no-caching approach (NC). As shown in Figure 6, index transactions under NC perform poorly at zero write probability, as they execute all index lookups at the server, thus sending a lot of messages and loading the server's CPU; in contrast, the index caching alternatives perform all lookups at the clients, where the whole database is cached. Interestingly, NC becomes the best approach as the write probability increases. As described in Section 3.2, NC does not send any callbacks for the next-entry lock during index insertions. Avoiding such callbacks helps NC by reducing both its message count and its contention (as potential lock conflicts that could arise at the remote caching clients need not be detected by NC). NC reduces its message count further by not sending any latch callbacks, as there are no cached index pages to callback. Furthermore, as the write probability increases, the number of index pages cached by the other alternatives drops sharply (due to invalidations); as a result, the performance

of the those alternatives worsens since there are not many index pages left to exploit at the clients.

To show how performance changes with the memory size, we also ran the same workload as above using smaller caches – here, client caches can hold 25% of the database and the server cache is 50% of the database. Figures 7 and 8 show the results from this experiment. Given the smaller server cache and the lack of access locality in the UNIFORM workload, the server's disk becomes the system bottleneck here. Thus, disk I/Os, rather than messages, play the most important role here. The only exception is 2PL, whose performance degrades rapidly with the write probability for the same reason as before, i.e., high contention. HC-S and FC perform almost the same here, as they have the same I/O behavior. In contrast to the previous experiment, HC-S is a bit better than HC-R here because in HC-S lock callbacks sent during index insertions do not usually invalidate the next-entry object and, as a result, HC-S saves a number of disk reads that are required by HC-R in order to re-fetch invalidated data pages. HC-S's superiority over HC-R is most notable in the case of index lookup transactions because these transactions access many more data pages than navigational transactions. To confirm our reasoning about HC-S vs. HC-R, we modified HC-S so that callbacks for the next-entry object will always invalidate that object. As shown in Figures 7 and 8, this sub-optimal version of HC-S (dashed line) performs about the same as HC-R.

Reducing the amount of available memory changes the relative performance of the no-caching approach as well, which now becomes the best alternative even at zero write probability. To understand the trade-offs between NC and HC-S (the best of the caching alternatives) we have to consider the mixture of data and index pages in the server and client caches. Due to the competition for memory between index and data pages, in the read-only case HC-S retains only 40% of the index cached at the clients and 55% at the server. In contrast, NC is able to cache 80% of the index at the server, as it performs all index lookups there (which increases the relative heat of the index pages). The net effect is that NC reads fewer index pages from the disk than HC-S. NC also performs somewhat fewer data page disk reads than HC-S, as it caches more data pages at the clients (due to the absence of index pages there). These I/O benefits of NC explain its superiority at zero write probability. Similar cache mix arguments account for the performance trends at higher write probabilities as well.

The last UNIFORM experiment presented here, uses the small cache setting and runs index transactions that perform range scans on the clustered index. This is a read-only experiment with respect to indices; only the third, non-indexed object attribute is updated by the navigational transactions. Figures 9 and 10 show the results from this experiment. HC-R is not included here, as it is identical to HC-S due to the lack of any index updates. For the same reason, HC-S, DC, and FC are practically the same here. In contrast, there are again important differences between the no-caching and the caching approaches. The same kind of analysis presented above, which is based on observing the page mixtures at the server and client caches, applies here as well. However, we will only emphasize the new lesson learned from this experiment. We found that an important factor contributing to NC's degradation in this workload is its high data contention. NC suffers from high contention here as the write probability increases because it pre-computes (at the server) the relatively long range scans invoked by the index transactions. As a result, scanning transactions acquire locks on the qualifying objects earlier than when they are actually needed, thus increasing the block rate and the lock waiting time of both types of transactions.

5.5 A HOTCOLD Experiment

Figures 11 and 12 show the results from a HOTCOLD experiment with the small cache setting. Here, during index transactions, clients perform lookups in their hot unclustered indices. This experiment illustrates yet another interesting case regarding the relative performance of NC. As shown in Figure 12, index transactions under NC perform poorly for the entire range of write probability. As in Figure 6, the caching alternatives perform very well at zero write probability because clients are able to cache their entire hot database area (both user objects and the associated unclustered index), thus eliminating client-server interactions. In contrast to Figure 6, NC remains worse than the caching alternatives (except for 2PL) for the entire range in Figure 12. This is because clients access their hot area most of the time and, as a result, they are able to keep most of their hot data and indices cached locally even when there are conflicting accesses among clients. (Put differently, the degree of inter-client data and index sharing is lower in HOTCOLD than in UNIFORM, where a high invalidation rate caused the number of client-cached data/index pages to decrease at high write probabilities.) Consequently, FC, HC-S, and HC-R are able to benefit from index caching even at high write probabilities and thus outperform NC in the index transactions. In the navigational transactions, however, NC is again the best approach due to its low overhead during index updates.

6 Conclusions

In this paper, we have investigated three questions related to B^+-tree management in client-server OODBMSs: (1) Can we do better than current approaches, which restrict index concurrency by performing 2-phase locking on index pages? (2) Can client caching of B^+-trees improve performance? (3) Assuming that index caching is allowed, what is a good approach to index cache consistency? To answer these questions, we presented four client-server B^+-tree algorithms that achieve high concurrency by employing techniques used in advanced centralized B^+-tree algorithms. Our first algorithm, NC, does not support client caching of indices; instead, clients hand over B^+-tree operations to the server. The second algorithm, FC, allows client caching of B^+-tree pages and performs all B^+-tree operations at the clients. The last two algorithms, HC-S and HC-R, are hybrids; they perform reads at the clients, using cached B^+-tree leaves, but hand over index updates to the server. HC-R differs from HC-S in that it allows clients to use out-of-date cached leaves under some conditions. All four algorithms, as well as a basic index caching algorithm that uses 2-phase locking (2PL), were implemented in the SHORE OODB system.

After explaining the above alternatives in detail, we presented performance results obtained by running SHORE on an IBM SP2 machine. These results justify our effort to increase index concurrency in client-server OODBSs. Specifically, in all of our experiments, the performance of the 2PL algorithm degraded dramatically as the rate of index updates was increased. With respect to the question of index caching vs. no caching, our results indicate that the caching alternatives are more robust. We found cases where NC was better than FC, HC-S, and HC-R, and we also found cases

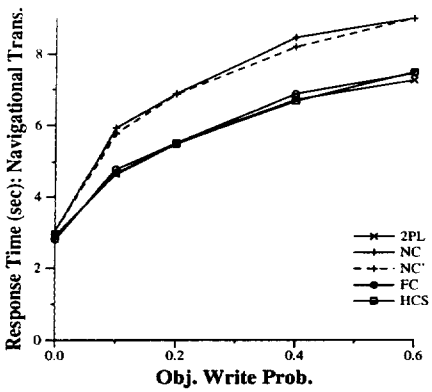

Figure 9: UNIFORM, Small Caches
Navigation + Clustered Index Scans

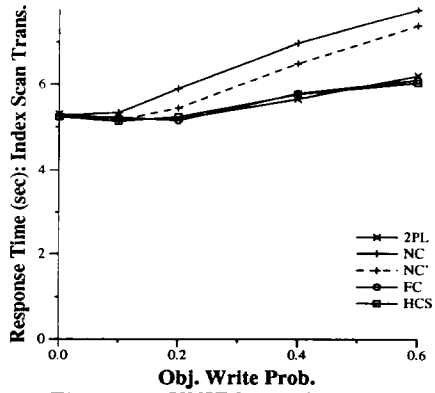

Figure 10: UNIFORM, Small Caches
Navigation + Clustered Index Scans

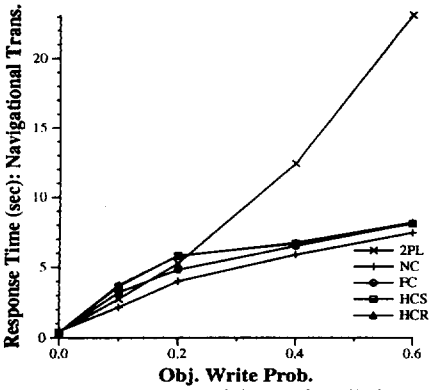

Figure 11: HOTCOLD, Small Caches
Navigation + Unclustered Index Lookups

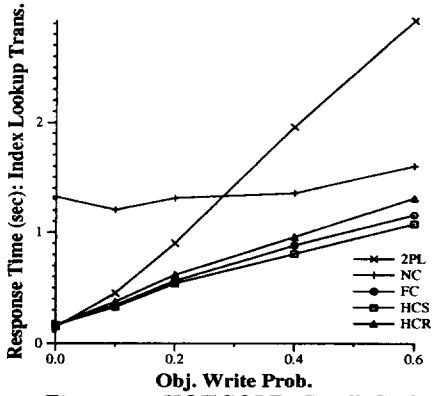

Figure 12: HOTCOLD, Small Caches
Navigation + Unclustered Index Lookups

where the reverse was true – but while the three caching alternatives were never much worse than NC, they outperformed NC by a large margin in some cases. Finally, we did not see large performance differences among the three highly concurrent caching alternatives. In terms of choosing one "best" algorithm, we favor HC-S because it provides a good balance between simplicity and performance: HC-S is much simpler than FC, and it often performed a bit better than FC as well. HC-S is preferable to HC-R as well because, as described in Section 3.4.2, HC-R is overly strict for applications that do not require degree 3 consistency.

References

[Basu95] J. Basu, A. Keller, "Centralized Versus Distributed Index Management in a Page Server OODBMS", Unpublished Manuscript, October 1995.

[Baye77] R. Bayer, M. Schkolnick, "Concurrency of Operations on B-Trees", *Acta Informatica*, Vol. 9, No. 1, 1977.

[Care91] M. Carey, M. Franklin, M. Livny, and E. Shekita, "Data Caching Tradeoffs in Client-Server DBMS Architectures", *Proc. ACM-SIGMOD Conf.*, Denver, June 1991.

[Care94] M. Carey, *et al.* "Shoring up Persistent Applications", *Proc. ACM-SIGMOD Conf.*, Minneapolis, May, 1994.

[Fran92] M. Franklin, M. Carey, "Client-Server Caching Revisited", *Proc. Int'l Workshop on Distributed Object Mgmt.*, Edmonton, Canada, Aug. 1992.

[Gott96] V. Gottemukkala, U. Ramachandran, E. Omiecinski "Relaxed Index Consistency for Data-Only Locking in a Client-Server Database", *12th IEEE Data Engineering Conf.*, Louisiana, Feb., 1996.

[Howa88] J. Howard, *et al*, "Scale and Performance in a Distributed File System", *ACM Trans. on Computer Sys.* 6(1), Feb. 1988.

[Lamb91] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb, "The ObjectStore Database System", *Comm. ACM* 34(10), Oct. 1991.

[Lehm81] P. Lehman, S. Yao, "Efficient Locking for Concurrent Operations on B-Trees" *ACM Trans. on Database Systems*, Vol 6, No 4, Dec. 1981.

[Lome93] D. Lomet, "Key Range Locking Strategies for Improved Concurrency", Digital Equipment Corporation, Feb. 1993.

[Moha89] C. Mohan, F. Levine, "ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging", IBM Research Report RJ 6846, IBM Almaden, 1989.

[Moha90] C. Mohan, "ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-Tree Indexes", *Proc. 16th VLDB Conf.*, Brisbane, Aug. 1990.

[Moha91] C. Mohan, I. Narang, "Recovery and Coherency-Control Protocols for Fast Intersystem Page Transfer and Fine-Granularity Locking in a Shared Disks Transaction Environment", *Proc. 17th VLDB Conf.*, Barcelona, Sept. 1991.

[Moha95] C. Mohan, I. Narang, "Locking and Latching Techniques for Transaction Processing Systems Supporting the Shared Disks Architecture" Unpublished Manuscript, February 1995.

[Sagi86] Y. Sagiv, "Concurrent Operations on B^*-Trees with Overtaking", *Journal of Computer and System Sciences*, Vol. 33, No. 2, 1986.

[Shas88] D. Shasha, N. Goodman, "Concurrent Search Structure Algorithms", *ACM Trans. on Database Systems*, Vol. 13, No. 1, March 1988.

[Wang91] Y. Wang and L. Rowe, "Cache Consistency and Concurrency Control in a Client/Server DBMS Architecture", *Proc. ACM-SIGMOD Conf.*, Denver, June 1991.

[Wilk90] W. Wilkinson, and M. Neimat, "Maintaining Consistency of Client Cached Data", *Proc. 16th VLDB Conf.*, Brisbane, Aug. 1990.

[Zaha96] M. Zaharioudakis, M. Carey, M. Franklin, "Adaptive, Fine-Grained Sharing in a Client-Server OODBMS: A Callback-Based Approach", to appear in ACM Trans. on Database Systems.