Transactional Client-Server Cache Consistency: Alternatives and Performance*

Michael J. Franklin
University of Maryland

Michael J. Carey
IBM Almaden Research Center

Miron Livny
University of Wisconsin

University of Maryland College Park
Technical Report CS-TR-3511 and UMIACS TR 95-84

September 1995

Abstract

Client-server database systems based on a page server model can exploit client memory resources by caching copies of pages across transaction boundaries. Caching reduces the need to obtain data from servers or other sites on the network. In order to ensure that such caching does not result in the violation of transaction semantics, a cache consistency maintenance algorithm is required. Many such algorithms have been proposed in the literature and, as all provide the same functionality, performance is a primary concern in choosing among them. In this paper we provide a taxonomy that describes the design space for transactional cache consistency maintenance algorithms and show how proposed algorithms relate to one another. We then investigate the performance of six of these algorithms, and use these results to examine the tradeoffs inherent in the design choices identified in the taxonomy. The insight gained in this manner is then used to reflect upon the characteristics of other algorithms that have been proposed. The results show that the interactions among dimensions of the design space can impact performance in many ways, and that classifications of algorithms as simply "pessimistic" or "optimistic" do not accurately characterize the similarities and differences among the many possible cache consistency algorithms.

1 Introduction

In recent years, trends towards distributed computing and object-orientation have combined to bring about the development of a new class of database systems. These systems use a client-server computing model to provide both responsiveness to users and support for complex, shared data in a distributed environment. Client processes, which execute close to the users, allow for fine-grained interaction with the database system as required by the advanced applications that utilize object-based DBMSs. Server processes manage the shared database and provide transaction management support to allow the coordination of database access by multiple users distributed across the network.

1.1 Client-Server Database System Architectures

Client-server DBMS architectures can be categorized according to the way in which client and server processes interact. Relational client-server DBMSs are based on *query-shipping* (e.g., the ODBC model), in which

*This work was partially supported by NSF grants IRI-8657323 and IRI-9409575, by ARPA under contract DAAB07-92-C-Q508, and by a research grant from IBM.

clients send queries to servers and the servers send results back to the clients. Queries may be sent as plain text (e.g., SQL), in a compiled representation, or as calls to pre-compiled queries stored at the server. Query shipping provides efficient use of communication for the coarse-grained, query-based data access that is prevalent in relational DBMSs. Furthermore, the conversion of an existing centralized DBMS to a query-shipping system is relatively straightforward, as the structure of the DBMS engine remains largely unchanged — it is centralized at the server.

In contrast, object-oriented database systems (OODBMSs) are typically built using *data-shipping*. Data-shipping systems perform most query processing (in addition to application processing) at the clients. When a client needs data from the server, it sends a request for the specific data items (e.g., objects or pages) that it requires. The advantages of data-shipping for OODBMSs are two-fold: First, data-shipping moves the data closer to the applications, allowing efficient fine-grained interaction between the application and the DBMS. This accelerates navigation through complex persistent data structures, as supported by the programmatic interfaces of object-based DBMSs. Second, data-shipping offloads much of the DBMS function from the server to the client workstations, thereby allowing the DBMS to exploit the plentiful and relatively inexpensive resources of the workstations. Workloads that demonstrate these performance advantages include the 001 (or "Sun") engineering database benchmark [Catt92] and the more recent 007 benchmark [Care93]. Examples of data-shipping systems include include research prototypes such as ORION [Kim90], EXODUS [Fran92c, Exod93], SHORE [Care94a], and THOR [Lisk92], as well as commercial products such as GemStone [Butt91], O2 [Deux91], ObjectStore [Lamb91], Ontos [Onto92], Objectivity [Obje91], and Versant [Vers91].

1.2 Client Caching

A potential weakness of the data-shipping approach is its susceptibility to network and/or server bottlenecks that can arise due to the volume of data requested by clients. In the presence of locality (i.e., the affinity of applications at certain workstations for certain subsets of the data items), caching data items in memory at the clients can reduce the volume of data that must be obtained from servers. Therefore, *client data caching* is a fundamental technique for improving the performance and scalability of data-shipping database systems.

There are two basic types of client caching: *intra-transaction caching* refers to the caching of data within a single transaction; *inter-transaction caching* allows clients to keep data cached locally even across transaction boundaries. Intra-transaction caching is easily implemented, requiring only that a client process be able to manage its own buffer pool and keep track of the locks it has obtained. This is because the transaction mechanism will ensure that any data that has been accessed by a transaction (and hence, brought into a client's memory) is valid. In contrast, inter-transaction data caching raises the need for a cache consistency protocol to ensure that applications always see a consistent (serializable) view of the database.

Cache consistency protocols for client-server database systems have been the subject of much study over the past five years [Wilk90, Care91a, Wang91, Fran92a, Fran93a, Adya95]. Many algorithms have been proposed; these algorithms differ along numerous dimensions such as their level of optimism versus pessimism, their use of invalidation or propagation (i.e., write-invalidate or write-broadcast [Arch86]) for updated data items, and their interaction with transaction management. While the papers mentioned previously have all provided comparisons of two or more proposed algorithms, there has been little published work on systematically exploring the large design space for client-server cache consistency algorithms. As a result, it has been difficult to compare and contrast the set of proposed algorithms and to choose among them.

This paper addresses this problem by proposing a taxonomy of transactional cache consistency maintenance algorithms that encompasses the algorithms that have been proposed in the literature. The taxonomy outlines the numerous dimensions of the design space for cache consistency algorithms and shows how proposed algorithms relate to one another. The taxonomy is used to focus a detailed analysis of the performance characteristics of six candidate algorithms; the insight gained in this manner is then used to reflect upon the characteristics of other proposed algorithms. In this way, the taxonomy allows us to shed light both on the nature of the design space for transactional cache consistency algorithms and on the performance tradeoffs implied by the many of the choices that exist in that design space.

1.3 Organization of the Paper

The remainder of the paper is organized as follows: Section 2 outlines a reference architecture for a data-shipping database system and motivates the use of inter-transaction caching as a fundamental technique in the design of such systems. Section 3 presents our taxonomy of transactional cache consistency maintenance algorithms. Section 4 describes three families of proposed transactional cache consistency maintenance algorithms in greater detail, examining their performance using a detailed simulation model. Section 5 comments on the tradeoffs made by the remaining algorithms in the taxonomy, and discusses related studies of cache consistency in other environments. Finally, Section 6 presents our conclusions.

2 Caching in Client-Server Database Systems

2.1 Reference Architecture

Figure 1 shows a high-level reference architecture for a data-shipping client-server DBMS. The underlying hardware is typical of that found in today's computer-assisted work environments (e.g., CAD, CAM, CASE, etc.). As shown in the figure, applications in a data-shipping DBMS execute at the client workstations. The DBMS consists of two types of processes that are distributed throughout the network. First, each client workstation runs a Client DBMS process, which is responsible for providing access to the database for the applications running at the local workstation. For protection reasons, the applications run in a separate address space from their local Client DBMS process, though some shared-memory may be used for efficiency (e.g., for communication)[1]. Applications send database access requests to their local Client DBMS process, which executes the request, in turn sending requests for transaction support and for specific data items to the Server DBMS processes. Server DBMS processes are the actual owners of data, and are ultimately responsible for preserving the integrity of the data and enforcing transaction semantics. The Server DBMS processes manage the stable storage on which the permanent version of the database and the log reside. They also provide concurrency control and copy management functions for the data that they own. The database is assumed to be statically partitioned across the servers.

Data-shipping systems can be structured either as *page servers*, in which clients and servers interact using physical units of data (e.g., pages or groups of pages), or *object servers*, which interact using logical units of data (e.g., tuples or objects). The tradeoffs between page servers and object servers have been studied for the single user case in [DeWi90], and more recently for the multi-user case in [Care94b, Chu94]. In the single user case, page servers were shown to offer communication savings given effective clustering. The multi-user work showed how to efficiently support fine-grained (e.g., object-level) sharing within a page server context.

[1] The process boundaries described here are typical, but not strictly required. For example, in EXODUS, applications are linked into a single process with the client DBMS code.

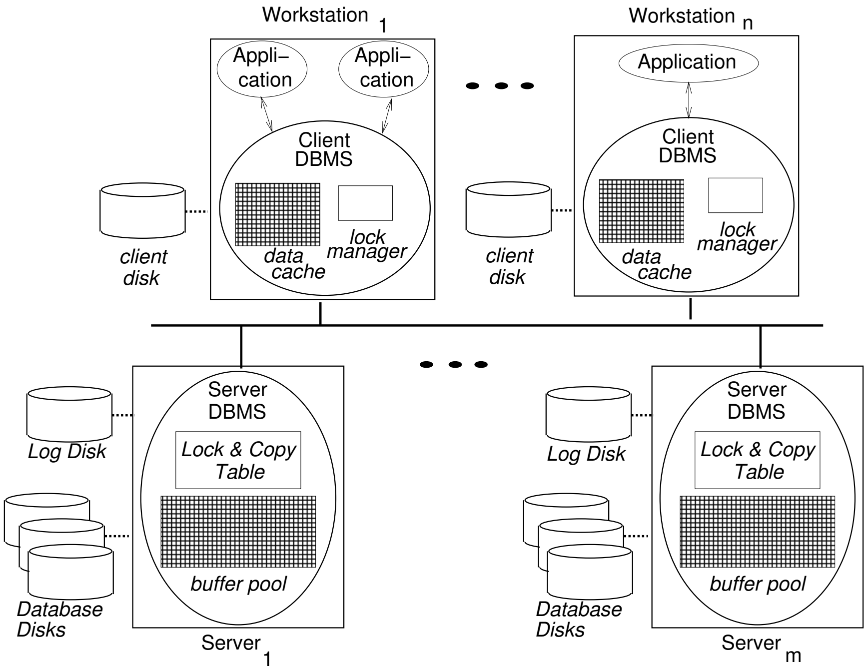

Figure 1: Reference Architecture for a Data-Shipping DBMS

Furthermore, our experience with the client-server EXODUS system [Fran92c, Exod93] has shown that a page server design provides additional advantages, such as the ability to use standard recovery algorithms (e.g., ARIES [Moha92]), and the ability to add efficient, server-based garbage collection algorithms [Amsa95]. For these reasons, the work in this paper is focused on page server architectures. It should be noted however, that many of the algorithms and results are also applicable to object servers.

In a page server DBMS, the database is divided into fixed length *pages*, which are typically on the order of four or eight KBytes. Each Client DBMS process is responsible for translating local application requests into requests for specific database pages and for bringing those pages into memory at the client. As a result, all of the pages referenced by an application are brought from the server(s) to the clients at some point. Some or all of these pages may be cached at the clients in order to reduce transaction path length and server load. A Server DBMS process is responsible for providing the most recent committed values for the data pages that it owns in response to clients' requests; of course, due to concurrency control conflicts, it may not be possible for the server to provide each requested page immediately.

An important aspect of page server systems is that all of the DBMS processes (both Client and Server) have memory that they use for buffering database pages. Therefore, the database system as a whole is responsible for managing buffers across the entire collection of machines. Similarly, the other distributed resources of the system, such as CPUs and disks, can also be exploited by the DBMS. The next section outlines the overall strategy for utilizing these resources effectively, as well as the properties of page server DBMSs that influence the design space of techniques for achieving this utilization.

2.2 Utilizing Client Resources

2.2.1 Improving Performance and Scalability

One consequence of the reference architecture presented in the previous section is that, while individual servers will typically have more CPU power and main memory than any single workstation, the majority of the *aggregate* CPU power and main memory available to the DBMS is likely to be at the clients. As a result, offloading functionality from the servers to the clients can improve both performance and scalability. By reducing their need to communicate with the server, the overall path length of transactions can be shortened, thereby improving latency. In addition, for a heterogeneous system, clients can cache data in their native format, thus avoiding the conversion costs that would be required when importing data items from a remote site. System throughput can be improved through the offloading of shared resources such as server machines and the network, both of which are potential performance bottlenecks.

The utilization of client resources can also enhance performance predictability. As Hennessy and Patterson state: "The attraction of a personal computer is that you don't have to share it with anyone. This means response time is predictable, unlike timesharing systems." [Henn90]. This is only true, however, to the extent that access to shared resources can be avoided. For example, if data must be obtained from a remote server, then the time required to respond to a user request will vary depending on the load on the server and the network. However, if the data can be obtained locally, without any server interaction, then its access time is independent of the activity at other sites.

2.2.2 Correctness and Availability Considerations

While the potential gains to be realized by exploiting client resources are large, they are ultimately limited by correctness and availability constraints. Workstation-server database systems must be capable of providing the same level of transaction support as more traditional database architectures (i.e., the ACID properties [Gray93]), including serializability. Because (as described in the following section) client caching is essentially a form of dynamic data replication, correctness criteria for managing replicated data are applicable in this environment. The extension of serializability to replicated data is called *one-copy serializability* [Bern87]. A one-copy serializable execution of transactions on a *replicated database* is equivalent to some *serial* execution of those transactions on a *non-replicated* database.

In terms of availability, client-server environments have inherent asymmetries that impact the extent to which responsibility for data and transaction management can be placed at the client workstations. These asymmetries include both physical and environmental ones. The physical differences result from economies of scale that dictate the placement of certain resources at either the client or server machines. For example, duplexed log disk storage or non-volatile memory can be placed more cost-effectively at server machines rather than at each client machine. The environmental differences result from the fact that client machines are often managed by users, while server machines are typically administered by an operations staff and are located in machine rooms rather than on desktops. More importantly, as portable client machines become increasingly popular, it cannot be assumed that a client is always reliably connected to the network. As a result, care must be taken so that the crash or disconnection of one client workstation does not impact the availability of data for applications running at other client workstations. This constraint limits the extent to which clients can be allowed to manage data independently (i.e., without server interaction).

2.3 Transactional Client Caching

The previous subsections have made the following two points:

1. There are substantial gains to be obtained through the utilization of client resources.
2. Methods that attempt to realize these gains must do so within the correctness and availability constraints of the client-server DBMS environment.

The combination of these two factors leads to the adoption of *client caching* as a fundamental technique in the design of a page server DBMS. For a page server system, client caching refers to the ability of clients to retain copies of database pages locally once they have been obtained from the server. In this context, client caching can be thought of in the following manner:

$$\text{Client Caching} = \text{Dynamic Replication} + \text{Second-Class Ownership}$$

Dynamic replication means that page copies are created and destroyed based on the runtime demands of clients. When a client needs to access a page, a copy of that page is placed in the client's cache if one does not already exist. Page copies are removed from a client's cache in order to make room for more recently requested ones or, under some caching algorithms, because they become invalid. This is in contrast to *static* replication schemes, in which the replication of data is determined as part of the physical database design process (e.g., [Ston79]).

It is well known that replication can reduce data availability for updates in the presence of certain failures (e.g., network partition) in a distributed environment [Davi85]. *Second-class ownership* allows consistency to be preserved without sacrificing availability. Second-class ownership refers to the fact that the cached copies of pages are not considered to be the equals of the actual data pages, which are kept at the server[2]. Specifically, the server always retains locally any and all data that is necessary for ensuring transaction durability (e.g., data pages, logs, dirty page information, etc.), so that client-cached pages can be destroyed at any time without causing the loss of committed updates. This notion is crucial to data availability, as it allows the server to consider a client to be "crashed" at any time, and thus, to unilaterally abort any transactions active at that client. As a result, the system as a whole is never held hostage by an uncooperative or crashed client.

2.4 Cost Factors for Transactional Cache Consistency

While the potential benefits of caching are significant, there are potential pitfalls as well. In order to ensure that caching does not affect the correctness of client database accesses, a cache consistency maintenance algorithm is required. Such protocols can be complex and may entail substantial overhead. The costs of maintaining cache consistency can be influenced by several factors such as: 1) the amount of work (e.g., communication) required for control actions, 2) whether this work is done synchronously or asynchronously, 3) transaction blocking or aborts due to consistency actions, and 4) the extent to which clients can successfully utilize their caches.

In addition to being highly dependent on the choice of cache consistency maintenance algorithm, the performance impact of these cost factors is often workload-dependent. The challenge in designing cache

[2]The term "second-class ownership" is derived from a similar concept called "second-class replication" used in the CODA distributed file system [Kist91]. The two notions are similar in that a distinction is made between the "worth" of different types of copies. They differ, however, in that the second-class replicas of CODA are used to increase availability by allowing access to *inconsistent* data, whereas our notion of second-class copies is used to enhance the availability of *consistent* data.

consistency maintenance protocols is thus to provide correctness while minimizing the negative performance impact over a range of workloads. Because of the wide range of tradeoffs underlying the performance of cache consistency maintenance algorithms, many such algorithms have been proposed in the literature. The remainder of this paper describes the design space of transactional cache consistency maintenance algorithms and analyzes a number of proposed algorithms with respect to these cost factors.

3 A Taxonomy of Algorithms

Cache consistency algorithms for page server DBMSs combine aspects of distributed concurrency control, transaction management, and replicated data management. As a result, there is a wide range of options for the design of such algorithms. This section provides a taxonomy of transactional cache consistency algorithms that encompasses the major algorithms that have appeared in the literature, including [Wilk90, Care91a, Wang91, Fran92a, Fran93a, Adya95]. All of these algorithms provide one-copy serializability and are applicable to page server DBMSs (although some were originally proposed for object servers).

The taxonomy is shown in two parts in Figures 2 and 3. The figures show where proposed algorithms fit in the taxonomy. Branches of the taxonomy for which (to the best of our knowledge) no algorithms have been published, are shown using dashed boxes in the figures. At the highest level of the taxonomy (which divides the design space into Figures 2 and 3), algorithms are classified according to their policy for *Invalid Access Prevention.* This policy dictates whether or not the clients can trust the validity of their cache contents. As explained in the following, algorithms that use *avoidance* for invalid access prevention ensure that all cached data is valid, while those that use *detection* allow stale data to remain in client caches and ensure that transactions that are allowed to commit have not accessed such stale data.

There are many possible ways to organize the design space for cache consistency algorithms, and at first, it might seem odd to use the the avoidance/detection distinction as the most fundamental decision point in the taxonomy. A different, and possibly more intuitive approach is to divide the taxonomy along the lines of concurrency control and replicated data management, as has been done for algorithms in the shared disks environment [Rahm91]. Because the two concepts are so closely inter-related, however, dividing a taxonomy at the highest level along these lines can result in substantial duplication of mechanism within the taxonomy, hurting its descriptive effectiveness. Another approach that is often used informally is to classify algorithms at the coarsest level as either "pessimistic" or "optimistic". As will be seen in the following sections, such a binary classification is not meaningful for many algorithms; in fact, many of the algorithms that have been proposed use combinations of both pessimism and optimism that interact in complex ways. By using the invalid access prevention policy as the coarsest level of differentiation in the taxonomy, such hybrid algorithms can be easily accommodated.

The next section describes the avoidance-based and detection-based approaches for invalid access prevention. Because this choice is a major distinction among algorithms, the properties upon which the lower levels of the taxonomy are based differ depending on which invalid access prevention policy is used. The lower levels of the taxonomy for each option are then elaborated in the sections that follow.

3.1 Invalid Access Prevention

Transactional cache consistency maintenance algorithms must ensure that no transactions that access *stale* (i.e., out-of-date) data are allowed to commit. In a transactional world, a data item is considered to be

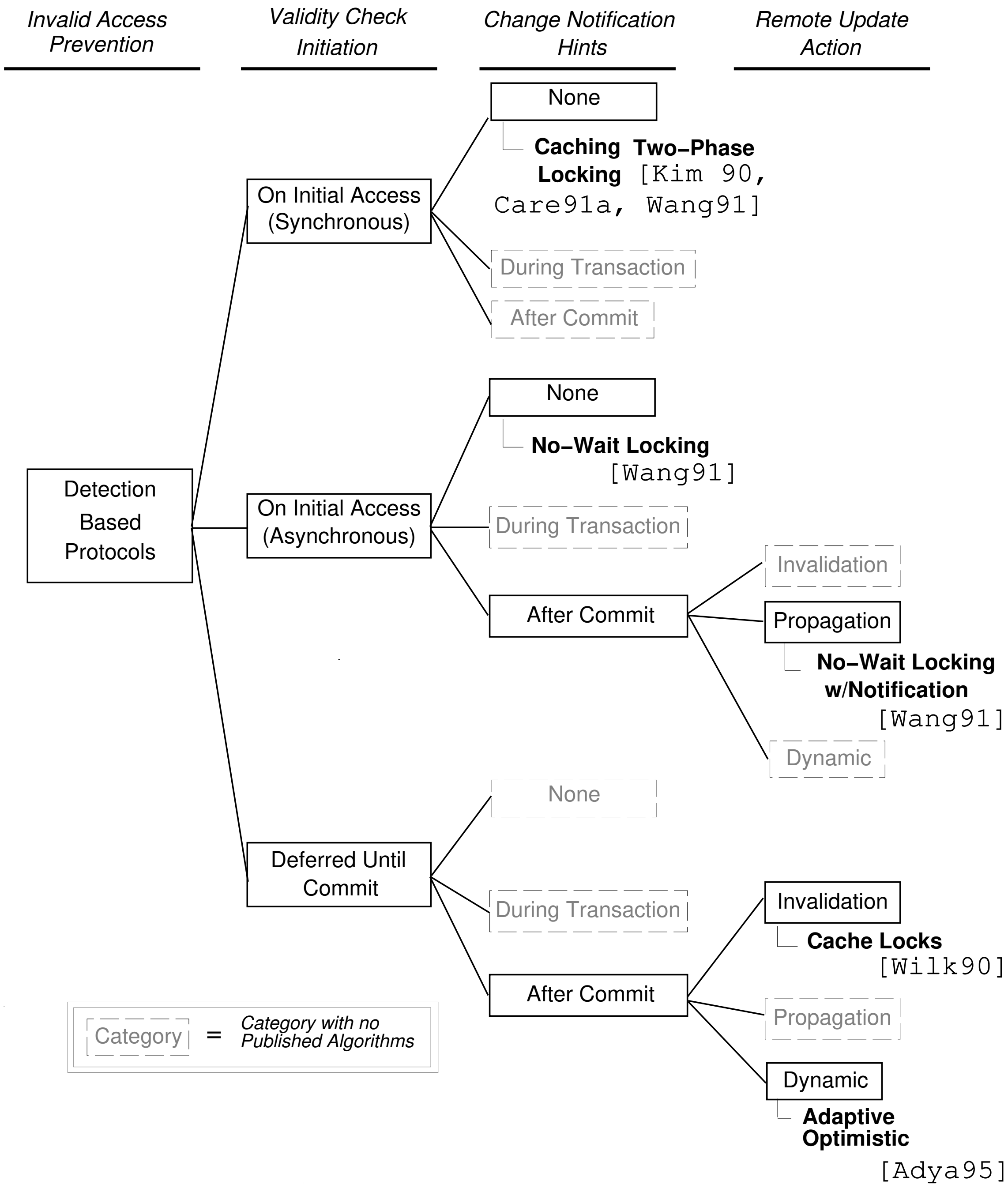

Figure 2: Taxonomy of Detection-Based Algorithms

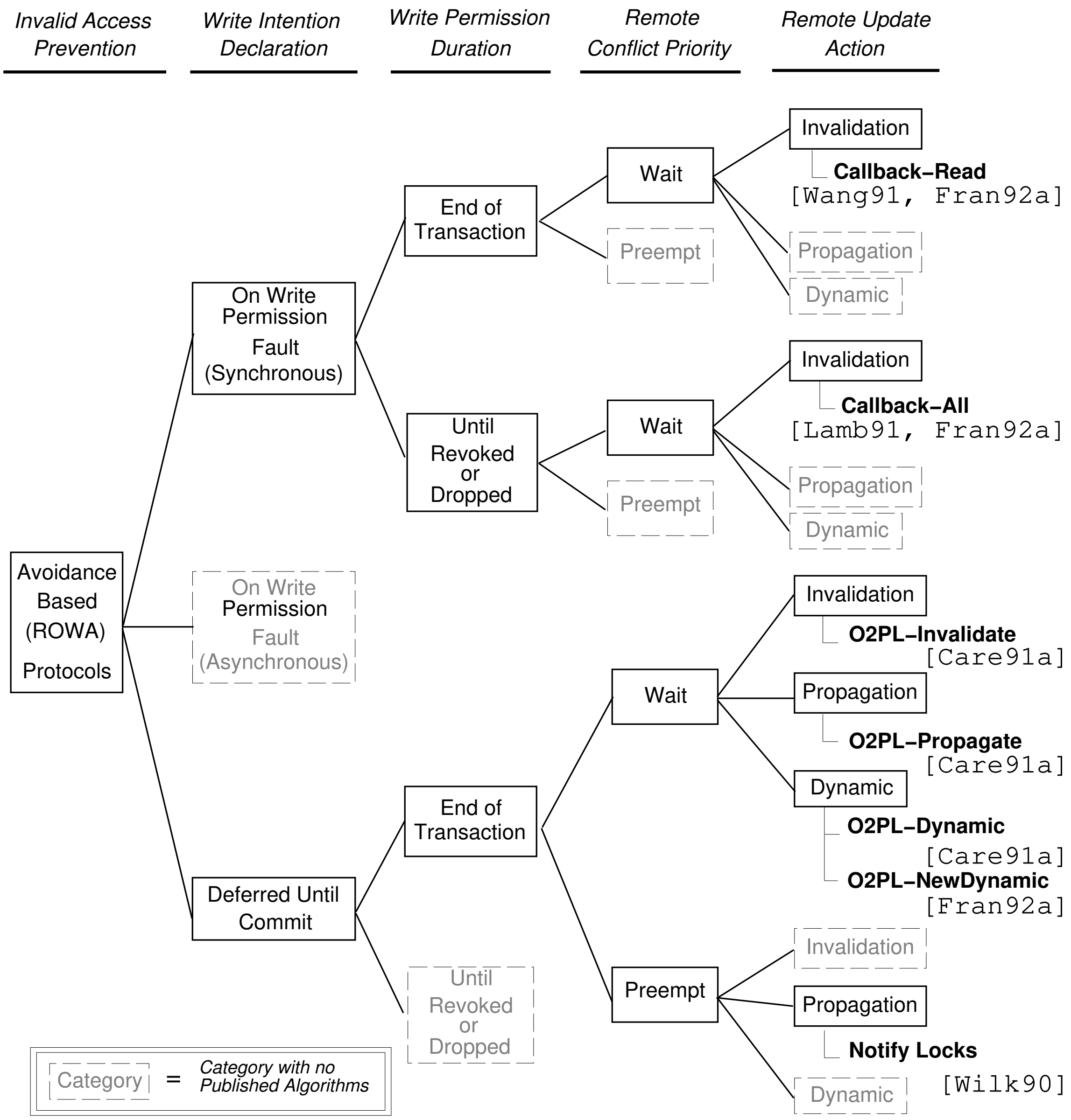

Figure 3: Taxonomy of Avoidance-Based Algorithms

stale if its value is older than the item's latest *committed* value.[3] In this taxonomy we partition consistency maintenance algorithms into two classes according to whether their approach to preventing stale data access is *detection-based* or *avoidance-based*. Qualitatively, the difference between these two classes is that detection-based schemes are *lazy*, requiring transactions to check the validity of accessed data, while avoidance-based schemes are *eager*; they ensure that invalid data is quickly (and atomically) removed from client caches.

Detection-based schemes allow stale data copies to reside in a client's cache for some period of time. Transactions must therefore check the validity of any cached page that they access before they can be allowed to commit. The server is responsible for maintaining information that will enable clients to perform this validity checking. Detection-based schemes are so named because access to stale data is explicitly checked for and detected. In contrast, under avoidance-based algorithms, transactions *never* have the opportunity to access stale data. Avoidance-based algorithms use a read-one/write-all (ROWA) approach to replica management. A ROWA protocol ensures that all existing copies of an updated item have the same value when an updating transaction commits. Avoidance-based algorithms can thus be said to *avoid* access to stale data by making such access impossible. In a ROWA scheme, a transaction is allowed to read *any* copy of a data item (which will typically be the one in its local client cache, if such a copy exists). Updates, however, must be reflected at *all* of the copies that are allowed to exist in the system beyond the updating transaction's commit point.[4]

Before proceeding, it should be noted that it is possible to augment detection-based algorithms with avoidance-based techniques. In fact, three of the detection-based algorithms cited in Figure 2 use *asynchronous* update notifications (i.e., they asynchronously install new values or remove stale values at remote clients) in order to reduce the probability of having stale data in the client caches. These are the algorithms with entries in the "Remote Update Action" column in Figure 2. These three algorithms lie strictly in the detection-based camp, however, as the notifications are sent only as "hints". That is, all three ultimately depend on detection to ensure that committing transactions have not accessed any stale data. Of course, the use of avoidance obviates any possible need for detection, so there is no augmentation in the opposite direction within the taxonomy.

3.2 Detection-based Algorithms

A number of detection-based algorithms (shown in Figure 2) have been proposed and studied in the literature [Kim90, Wilk90, Care91a, Wang91, Adya95]. The main argument for the detection-based approach is simplicity. Because their consistency actions involve only a single client and the server, the detection-based approaches allow the cache management software on the clients to be greatly simplified compared to the ROWA approach. For example, using detection, the system software can be structured such that clients do not ever have to receive asynchronous messages from the server. The EXODUS storage manager [Exod93] chose a detection-based approach largely for this reason. Also, servers can be made responsible for maintaining consistency information, thus relieving clients of that burden. The disadvantage of detection-based approaches, however, is a greater dependency on the server, which can result in additional overhead. There are three levels of differentiation in the detection-based side of the taxonomy: validity check initiation, change

[3] It is important to note that values become stale only as the result of the commit of an update transaction. Some consistency algorithms allow multiple transactions to simultaneously access different values of the same page, provided that serializability is not violated, essentially yielding a dynamic form of multiversion concurrency control [Bern87].

[4] As described in Section 2.3, the use of second-class replication allows the server to unilaterally eliminate any unreachable copies from the protocol so that transaction processing can continue.

notification hints, and remote update action.

3.2.1 Validity Check Initiation

The coarsest level of differentiation for the detection-based half of the taxonomy is based on the point (or points) during transaction execution at which the validity of accessed data is checked. The validity of any accessed data must be determined before a transaction can be allowed to commit; consistency checks for all data touched by a transaction must therefore begin and complete during the execution of the transaction. In the taxonomy, three classes of validity checking strategies are differentiated:

- Synchronous, on each initial access to a page (cached or otherwise) by a transaction.
- Asynchronous, with checking initiated on the initial access.
- Deferred, until a transaction enters its commit processing phase.

All three classes have the property that once the validity of a client's copy of a data item is established, that copy is guaranteed to remain valid for the duration of the transaction. To implement this guarantee, the server must not allow other transactions to commit updates to such items until a transaction that has received a validity guarantee finishes (commits or aborts). As a result, transactions must obtain permission from the server before they are allowed to commit an update to a data item.[5]

Synchronous validity checking is the simplest of the three classes. On the first access that a transaction makes to a particular data item, whether cached or not, the client must check with the server to ensure that its copy of the item is valid. This is done in a synchronous manner — the transaction is not allowed to access the item until its validity has been verified. Once the validity of the client's copy of the item has been established (which may involve the sending of a new, valid copy to the client), the copy is guaranteed to remain valid at least until the transaction completes. Asynchronous validity checking is similar, but the transaction does not wait for the result of the check. Rather, it proceeds to access (or write) the local copy under the assumption that the check will succeed. If this optimism turns out to be unfounded, then the transaction must abort. Finally, deferred validity checking is even more optimistic than asynchronous checking. No consistency actions pertaining to cached data are sent to the server until the transaction has completed its execution phase and has entered its commit phase. At this point, information on all the data items read and written by the transaction is sent to the server, and the server determines whether or not the transaction should be allowed to commit.

These three classes provide a range from pessimistic (synchronous) to optimistic (deferred) techniques. Therefore, they represent different tradeoffs between checking overhead and possible transaction aborts. Deferring consistency actions can have two advantages. First, and most significantly, consistency actions can be bundled together in order to reduce and/or amortize consistency maintenance overhead. Secondly, the consistency maintenance work performed for a transaction that ultimately aborts is wasted; deferred consistency actions can avoid some of this work. There are also potential disadvantages to deferring consistency actions, however. The main disadvantage is that deferral can result in the late detection of data conflicts, which will cause the (late) abort of one or more transactions. Such aborts can hurt performance due to wasted work. In addition, transaction aborts may result in significant lost work for users in the highly

[5]Although it is not strictly necessary, all of the algorithms shown in Figure 2 use the same initiation method for update permission requests as they do for validity checks. If this were not the case, validation and update requests would require separate dimensions in the taxonomy.

interactive OODBMS environments for which page servers are often intended. The asynchronous approach is an attempt at a compromise; it aims to mitigate the cost of interaction with the server by removing it from the critical path of transaction execution, while at the same time lowering the abort rate and/or cost through the earlier discovery of conflicts.

3.2.2 Change Notification Hints

The emphasis on optimistic (i.e., asynchronous and deferred) techniques found in the literature on detection-based algorithms is an artifact of the cost tradeoffs of the page server environment. Communication with the server is an inherently expensive operation, so designers of detection-based algorithms often rely upon optimism in an attempt to reduce this cost. Optimistic techniques are oriented towards environments in which conflicts are rare and the cost of detecting conflicts is high. While there is currently no definitive understanding of page server DBMS workloads, it is generally assumed that such workloads have lower levels of conflict than more traditional DBMS workloads, such as transaction processing [Catt91]. In a transactional caching environment, however, the notion of conflict must take into account not only *concurrent* data sharing, but also *sequential* sharing. Sequential sharing arises when transactions that *do not run concurrently* access the same data. Because caching strives to retain data at a site even after a transaction has completed, the cache consistency maintenance algorithm must also deal effectively with this type of sharing. Recent studies of file system workloads [Rama92, Sand92] indicate that sequential sharing may, in fact, be quite common in the types of situations in which page servers are intended to be used. If this is the case, then the naive use of optimistic techniques could result in unacceptably high abort rates.

Two approaches to reducing the potential for aborts in optimistic techniques have been proposed. One is to simply treat "hot spot" data differently, e.g., by switching to a more pessimistic protocol for such data (e.g., [Adya95]). The details of such adaptive algorithms have not yet been published, however. The other approach is to borrow techniques from the avoidance-based (ROWA) algorithms on the other side of the taxonomy to reduce the amount of stale data that resides in client caches. We call such techniques *change notification hints*. As can be seen in Figure 2, three of the algorithms found in the literature use some form of change notification hints (or simply, "notifications"). A notification is an action that is sent to a remote client as the result of an update that may impact the validity of an item cached at that client. Removing or updating a stale copy reduces the risk that a subsequent transaction will be forced to abort as a result of accessing that data item.

Notifications can be sent asynchronously at any time during the execution of an updating transaction, or even after such a transaction commits. In fact, sending notifications before commit can be dangerous if the notifications actually update the remote copies rather than simply removing them; if the transaction on whose behalf the notification was sent eventually aborts, then the remote updates will have to be undone, adding significant complexity (e.g., cascading aborts) and expense to the algorithm. Early notifications that simply purge copies from remote caches are less problematic; still, they too can cause unnecessary aborts at remote sites if active transactions have already accessed the invalidated copies there. Because of these complexities, all three of the algorithms shown in Figure 2 that use change notification hints send them only after the updating transaction has committed.

3.2.3 Remote Update Action

The final level of differentiation in the detection-based half of the taxonomy is concerned with the action taken when a notification arrives at a remote site. There are three options here: propagation, invalidation, and choosing dynamically between the two. Propagation results in the newly updated value being installed at the remote site in place of the stale copy. Invalidation, on the other hand, simply removes the stale copy from the remote cache so that it will not be accessed by any subsequent transactions. After a page copy is invalidated at a site, any subsequent transaction that wishes to access the page at that site must obtain a new copy from the server. A dynamic algorithm can chose between invalidation and propagation heuristically in order to optimize performance for varying workloads. The concepts of propagation and invalidation are analogous to the notions of write-broadcast and write-invalidate (respectively) found in multiprocessor cache consistency algorithms [Arch86].

3.3 Avoidance-based Algorithms

Avoidance-based algorithms form the other half of our taxonomy of transactional cache consistency maintenance algorithms. The avoidance-based side of the taxonomy is shown in Figure 3. As stated previously, avoidance-based algorithms enforce consistency by making it impossible for transactions to ever access stale data in their local cache. They accomplish this by directly manipulating the contents of remote client caches as the result of (or prior to) client updates. Because consistency actions manipulate page copies in remote client caches, the client software must include additional mechanisms to support these actions (e.g., some schemes require that clients have a full function lock manager). While this additional responsibility increases the complexity of the client-side software, the goal is to reduce reliance on the server and thereby offload shared resources.

In addition to their need for additional client support, avoidance-based algorithms also require extra information to be maintained at the server. Specifically, all of the avoidance-based algorithms described here require that the server keep track of the location of all page copies. In order to satisfy the "write all" requirement of the ROWA paradigm, it must be possible to locate all copies of a given page. One way to accomplish this is through the use of broadcast, as in the snooping protocols used in caching algorithms for small-scale multiprocessors [Good83]. Reliance on broadcast is not a viable option in a page server DBMS environment, however, due to cost and scalability issues. As a result, a "directory-based" approach [Agar88] must be used. As discussed in Section 2.3, the server is the focal point for all transaction management functions and is responsible for providing clients with requested data; as a result, the avoidance-based algorithms covered here all maintain a directory of client page copies at the server.

There are four levels in the avoidance-based half of the taxonomy, as shown in Figure 3: write intention declaration, write permission duration, remote conflict priority, and remote update action. Two of these dimensions, write intention declaration and remote update action, are analogous to dimensions that appeared in the detection-based side of the taxonomy.

3.3.1 Write Intention Declaration

As with the detection-based algorithms, the avoidance-based algorithms can be categorized according to the time at which transactions initiate consistency actions. The nature of their consistency actions, however, is somewhat different than in the detection-based schemes. Because of the ROWA protocol, transactions executing under an avoidance-based scheme can always read any page copy that is cached at their local client. Thus, interaction with the server is required *only* for access to pages that are not cached locally and for updates to cached pages. Interactions with the server to obtain copies of non-cached pages must, of course, be done synchronously. On a cache miss, the client requests the desired page from the server. When the server responds with a copy of the page, it also implicitly gives the client a guarantee that the client will be informed if another client performs an operation that would cause the copy to become invalid.

While all of the avoidance-based algorithms use the same policy for handling page reads, they differ in the manner in which consistency actions for *updates* are initiated. When a transaction wishes to update a cached page copy, the server must be informed of this *write intention* sometime prior to transaction commit so that it can implement the ROWA protocol. When the server grants write permission on a page to a client, it guarantees that, for the duration of the permission, the client can update that page without again having to ask the server[6]. The duration of write permissions is addressed in Section 3.3.2.

A *write permission fault* is said to occur when a transaction attempts to update a page copy for which it does not possess write permission. The taxonomy contains three options for when clients must declare their intention to write a page to the server:

- Synchronous, on a write permission fault.
- Asynchronous, initiated on a write permission fault.
- Deferred, until the updating transaction enters its commit processing phase.

In the first two options, clients contact the server at the time that they first decide to update a page for which they do not currently posses write permission. As in the detection-based case, such requests can be performed synchronously (where the transaction waits for the server to acknowledge the registration of the write permission) or asynchronously (where the transaction does not wait). In the third option, declarations of write intentions are deferred until the transaction finishes its execution phase (if the updated data can be held in the client cache until then).

The tradeoffs among synchrony, asynchrony and deferral for write intentions are similar in spirit to those previously discussed for the detection-based algorithms: synchronous algorithms are pessimistic, deferred ones are optimistic, and asynchronous ones are a compromise between the two. The magnitude of these tradeoffs, however, are quite different for avoidance-based algorithms. The global (ROWA) nature of these algorithms implies that that consistency actions may be required at one or more remote clients before the server can register a write permission for a given client (or transaction). Therefore, consistency actions can involve substantial work here. The relatively high cost of consistency maintenance actions on the avoidance-based side of the taxonomy tends to strengthen the case for deferral of such actions. It is also important to note that with avoidance-based algorithms, the remote consistency operations are in the critical path of transaction commit. That is, an update transaction cannot commit until all of the necessary consistency operations have been successfully completed at remote clients.

3.3.2 Write Permission Duration

In addition to *when* write intentions are declared, avoidance-based algorithms can also be differentiated according to *how long* write permission is retained for. There are two choices at this level of the taxonomy: write permissions can be retained only for the duration of a particular transaction, or they can span multiple transactions at a given client. In the first case, transactions start with no write permissions, so they must eventually declare write intentions for all pages that they wish to update. At the end of the transaction, all

[6] A "permission", while similar to a "write lock", differs in two significant ways. First, permissions are granted to client sites rather than to individual client transactions. Second, permissions are not subject to a two-phase constraint. As shown in Section 4.1.2, the notion of permissions provides additional flexibility in the design of cache consistency maintenance algorithms.

write permissions are automatically revoked by the server. In the second case, a write permission can be retained at a client site until the client chooses to drop the permission (e.g., after removing a copy of a page from its buffer pool). In the latter case, the server may also ask a client to drop its write permission on a page copy (in conjunction with the performance of a consistency action).

3.3.3 Remote Conflict Priority

The third level of differentiation for avoidance-based algorithms is the priority given to consistency actions when they are received at remote clients. There are two options here: wait and preempt. A *wait* policy states that consistency actions that conflict with the operation of an ongoing transaction at a client must wait for that transaction to complete. In contrast, under a *preempt* policy, ongoing transactions can be aborted as the result of an incoming consistency action. Under the wait policy, the guarantees that are made to clients regarding the ability to read cached page copies are somewhat stronger than under the preempt policy. This is because the wait policy forces a remote writer to serialize behind a local reader if a conflict arises, whereas writers always have priority over readers under the preempt policy, so conflicting readers are aborted.

3.3.4 Remote Update Action

The final level on the avoidance-based side of the taxonomy is based on how remote updates are implemented. The options here are the same as in the detection-based case, namely: invalidation, propagation, and choosing dynamically between the two. As stated previously, the propagation of updates to remotely cached copies can be problematic if consistency actions are sent to remote sites during a transaction's execution phase. As a result, both of the published algorithms in the taxonomy that send remote consistency actions during the execution phase rely on invalidation as the mechanism for handling updates remotely.

An important difference between remote update actions under the avoidance-based algorithms and under the detection-based ones (discussed earlier) is that in the avoidance-based case, the remote operations are initiated and must be completed on behalf of a transaction *before the transaction is allowed to commit.* This is necessary to maintain the ROWA semantic guarantees. Therefore, if update propagation is used, all remote sites that receive the propagated update must participate in a two-phase commit with the server and the client at which the transaction is executing. In contrast, invalidation does not require two-phase commit, as data is simply removed from the remote client caches in this case.

4 A Performance Comparison of Three Algorithm Families

The taxonomy presented in the previous section illuminates the wide range of options that are available to designers of transactional cache consistency maintenance algorithms. As defined in this paper, all such algorithms provide the same functionality — they support one-copy serializability in the presence of dynamic caching, so performance issues are a primary consideration in choosing among them. In this section, we examine the performance implications of a number of the choices identified in the taxonomy.

To date, our work has focused primarily on algorithms from the avoidance-based half of the taxonomy [Care91a, Fran92a]. In this section, we consolidate the results of those studies and re-examine their conclusions in the context of the design choices identified in the taxonomy. We first describe six candidate algorithms from three different algorithm families. We then provide an overview of a detailed simulation model and a set of four workloads used to examine the relative performance of those algorithms. Finally, performance results from a series of simulations are analyzed to shed light on the relevant design decisions from the taxonomy. The insights gained through this process are then used in Section 5 to reflect on the performance characteristics of the remaining design choices and algorithms in the taxonomy.

4.1 Algorithms

The algorithms that we focus on in this section come from three families: Server-based two-phase locking (S2PL), Callback Locking (CBL), and Optimistic 2PL (O2PL). While these algorithms differ in many ways, they all stem from the fundamental observation that because cached data is dynamically replicated data, techniques originally devised for managing replicated data can be adapted to manage cached copies. In the following, we briefly describe each of these three algorithm families (see [Fran93a] for a more detailed description) and then identify pairs of algorithms that can be used to isolate the impact of a number of the design choices described in Section 3.

4.1.1 Server-Based Two-Phase Locking (S2PL)

Server-based two-phase locking algorithms are detection-based algorithms that validate cached pages synchronously on a transaction's initial access to the page. Server-based 2PL schemes are derived from the *primary copy* approach to replicated data management [Alsb76, Ston79]. Before a transaction is allowed to commit, it must first access a specially designated copy (i.e., the primary copy) of each data item that it reads or writes. In a page server DBMS (with no server replication), the primary copy of any page is the one that resides at the server. For reads, the client's copy of the page must be verified to have the same value as the server's copy. For writes, the new value created by the transaction must be installed as the new value of the primary copy.

The performance results examined here include an algorithm called *Caching 2PL* (C2PL). In C2PL, cache consistency is maintained using a "check-on-access" policy. All page copies are tagged with a version number that uniquely identifies the state of the page.[7] When a transaction attempts a page access for which it has not obtained the proper lock (i.e., read or write), it sends a lock request to the server and waits for the server's response. If the page is cache-resident at the client, then the cached copy's version number is included in the lock request message. If any transactions hold conflicting locks, then the lock request blocks at the server until those locks are released. When the server grants a read lock to a client, it also determines whether or not the client has an up-to-date cached copy of the requested page. If not, then the server piggybacks a valid copy of the page on the lock response message returned to the client. C2PL uses strict two-phase locking — all locks are held until transaction commit or abort. Deadlocks are detected through a centralized scheme at the server, and are resolved by aborting the youngest transaction involved in the deadlock. C2PL is one of the simplest algorithms that supports inter-transaction caching, and therefore, algorithms similar to C2PL have been implemented in several systems, including the ORION-1SX prototype [Kim90] and the EXODUS storage manager [Exod93]. An algorithm similar to C2PL has also been studied in [Wang91].

For comparison purposes, the performance study also includes results for an algorithm called *Basic 2PL* (B2PL) that allows only *intra-transaction* caching. B2PL works similarly to C2PL, except that under B2PL, the client's buffer pool is purged upon transaction termination. Since every transaction starts with an empty

[7] Data pages are typically tagged with such numbers, called Log Sequence Numbers (LSNs), in systems that use the Write-Ahead-Logging protocol for crash recovery [Gray93].

buffer pool, no page copies ever need to be validated with the server. Comparing the performance of B2PL to that of C2PL (and the other algorithms) isolates the degree of performance improvement that is due to inter-transaction caching.

4.1.2 Callback Locking (CB)

Callback Locking algorithms are similar to C2PL, in that they are extensions of two-phase locking that support inter-transaction page caching. In contrast to the detection-based C2PL algorithm, however, Callback Locking algorithms are avoidance-based. Therefore, locally cached page copies are always guaranteed to be valid, so transactions can read them without contacting the server (i.e., only a local read lock is required). On a cache miss, the client sends a page request message to the server. The server returns a valid copy of the requested page when it determines that no other active clients believe they have write permission for the page. In Callback Locking, write intentions are declared synchronously — a client must have write permission on a page before it can grant a local write lock to a transaction. Because write permissions are obtained during transaction execution, transactions can commit after completing their operations without performing any additional consistency maintenance actions. We have studied two Callback Locking variants: Callback-Read (CB-R), where write permissions are granted only for the duration of a single transaction (i.e., they are treated like traditional write locks), and Callback-All (CB-A), where write permissions are retained at clients until being called back or until the corresponding page is dropped from the cache. Both variants use invalidation for handling remote updates.

With Callback Locking (as with all avoidance-based algorithms), the server keeps track of the locations of cached copies throughout the system. Clients inform the server when they drop a page from their buffer pool by piggybacking that information on the next message that they send to the server. The server's copy information is thus conservative — there may be some delay before the server learns that a page is no longer cached at a client. Transactions obtain locks from the local lock manager at the client site on which they execute. Read lock requests, as well as requests for write locks on pages for which the client has obtained write permission, can be granted immediately without contacting the server. Write lock requests on pages for which write permission has not yet been obtained cause a "write permission fault". On a write permission fault, the client must register its write intention with the server and then wait until the server responds that the permission has been granted before continuing.

When a write intention declaration arrives at the server, the server issues callback requests to all sites (except the requester) that hold a cached copy of the requested page. At a client, such a callback request is treated as a request for a write lock on the specified page. If the request cannot be granted immediately, due to a lock conflict with an active transaction, the client responds to the server by saying that the page is currently in use. When the callback request is eventually granted at the client, the page is removed from the client's buffer and an acknowledgement message is sent to the server. When all callbacks have been acknowledged to the server, the server registers the write permission on the page for the requesting client and informs the client that it has done so. Any subsequent read or write requests for the page by transactions from other clients will then be blocked at the server until the write permission is released by the holding client or else revoked by the server.

If a read request for a page arrives at the server and a write permission for the page is currently registered for some other client, then the server action is algorithm-dependent. Under Callback-Read (CB-R), where Write Permission Duration is only until the end of a transaction, the read request is simply blocked at the server until the termination of the current transaction at the client holding the permission. In contrast, under Callback-All (CB-A), the server sends a *downgrade* request to that client. A downgrade request is similar to a callback request, but rather than responding by removing the page from its buffer, the client simply acknowledges to the server that it no longer has write permission on the page. At a remote client, a downgrade request for a page copy must first obtain a read lock on the page in order to ensure that no transactions active at the client are currently holding write locks on the page. The downgrade request blocks at the client if a conflict is detected, in which case a message is sent to the server informing it of the conflict. Global deadlocks can arise due to callback and downgrade requests. These deadlocks can always be detected at the server, however, because clients inform the server when they block such requests. As in the S2PL algorithms, deadlocks are resolved by aborting the youngest involved transaction.

At the end of a transaction, the client sends *copies* of any cached updated pages to the server. This is done only to simplify recovery, as no other sites can access a page while it is cached with write permission at a site. Thus, it is technically possible to avoid sending a copy of dirty page back to the server until the write permission on the page is downgraded or the page is dropped [Fran93b].

Callback-style algorithms originated in the operating systems community for maintaining cache consistency in distributed file systems such as Andrew [Howa88] and Sprite [Nels88], both of which provide weaker forms of consistency than that required by database systems. More recently, a Callback Locking algorithm that provides transaction serializability has been employed in the ObjectStore OODBMS [Lamb91]. An algorithm similar to CB-R was also studied in [Wang91].

4.1.3 Optimistic Two-Phase Locking (O2PL)

The third family of caching algorithms that we have studied is Optimistic Two-phase Locking (O2PL). The O2PL algorithms are derived from a concurrency control protocol that was originally developed for replicated distributed databases [Care91b]. The O2PL algorithms are avoidance-based, but they are more "optimistic" than Callback Locking because they defer write intention declaration until the end of a transaction's execution phase. We have developed and analyzed several O2PL variants that differ in their implementation of remote update actions. In this paper we focus on two such variants: O2PL-Invalidate (O2PL-I), which always invalidates remote copies, and O2PL-Propagate (O2PL-P), which always propagates updated page copies to remote clients that are caching the updated pages.

Under O2PL, each client has a local lock manager from which the proper lock must be obtained before a transaction can access a data item at that client. No locks are obtained at the server during the execution phase of a transaction.[8] Transactions update pages in their local cache, and these updated pages are retained at the client (unless they are aged out) until the transaction enters its commit phase. When an updating transaction is ready to enter its commit phase, it sends a message to the server containing the new copies of such pages. The server then acquires exclusive locks on these pages on behalf of the finishing transaction. The locks obtained at the server are held until the transaction completes, as they will allow the server to safely install the new page values.

Once the required locks have been obtained at the server, the server sends a message to each client that has cached copies of any of the updated pages. These remote clients obtain exclusive locks on their local copies (if present) of the updated pages on behalf of the committing transaction. If any of their transactions currently holds a read lock on a local copy, then the update transaction will have to wait for the reader transaction(s) to complete before it can continue commit processing. Once all of the required locks have

[8] Actually, a non-two-phase read lock (i.e., latch) is obtained briefly at the server when a data item is in the process of being prepared for shipment to a client, to ensure that the client is given a transaction-consistent copy of the page.

been obtained at a remote site, that site performs consistency actions on its copies of the updated pages: Under O2PL-I, the client simply purges its copies of the updated pages, releases its local locks on those pages, and then sends an acknowledgement message to the server. In contrast, under O2PL-P, remote clients must enter a two-phase commit protocol with the server. First, each client sends a message to the server indicating that it has obtained the necessary local locks. This message acts as the "prepared" message of the commit protocol. When the server has heard from all involved clients, it sends copies of the updated pages to those sites. These messages initiate the second phase of the commit protocol. Upon receipt of the new page copies, the clients install them in their buffer pools and then release the locks on those pages.[9]

Because O2PL is distributed and locking-based, distributed deadlocks can arise in O2PL-I and O2PL-P. Each client therefore maintains a local waits-for graph which is used to detect deadlocks that are local to that client. Global deadlocks are detected using a centralized algorithm in which the server periodically requests local waits-for graphs from the clients and combines them to build a global waits-for graph.[10] As in the previously described algorithms, deadlocks are resolved by aborting the youngest transaction involved.

4.1.4 Evaluating the Tradeoffs

The three families of cache consistency maintenance algorithms described in the preceeding sections cover a number of the design alternatives identified in the taxonomy presented in Section 3. Table 1 summarizes the portion of the design space covered by these algorithms and shows which algorithms can be compared in order to examine the performance tradeoffs implied by a given decision. The focus of our work in this area has been on avoidance-based algorithms, so the majority of the tradeoffs investigated come from that side of the taxonomy. However, because several of the avoidance-based dimensions have analogs on the detection-based side of the taxonomy, the comparisons presented here can shed light on a number of the detection-based tradeoffs as well. The algorithms that are not directly addressed in this section will be discussed in Section 5.

Design Choice	**Algorithms to Compare**
Invalid Access Prevention	C2PL (Detection) vs. CB-A (Avoidance)
Write Intention Declaration	CB-R (Synchronous) vs. O2PL-I (Deferred)
Write Permission Duration	CB-R (Single Transaction) vs. CB-A (Until Revoked or Dropped)
Remote Update Action	O2PL-I (Invalidation) vs. O2PL-P (Propagation)

Table 1: Design Choices and Relevant Comparisons

Invalid Access Prevention (C2PL vs. CB-A) As described in Section 3, the top-level design choice in the taxonomy is the policy used for preventing access to invalid data. *Detection* requires the validity of all accessed data to be explicitly confirmed prior to transaction commit, while *Avoidance* ensures that

[9]It should be noted that the receipt of a propagated page copy at a client does not affect the page's LRU status at that site.

[10]Note that deadlocks involving consistency actions can be resolved early, rather than waiting for periodic detection, as any conflict detected between two consistency actions or between a consistency action and an update will ultimately result in a deadlock [Care91b].

transactions never have the opportunity to access stale data. The S2PL algorithms are detection-based, whereas the CBL and O2PL algorithms are all avoidance-based. Among these algorithms, comparing the performance of C2PL and CB-A can provide the clearest insights into this tradeoff. Both algorithms allow inter-transaction caching, neither propagates updated pages, and both initiate their consistency actions synchronously.

Write Intention Declaration (CB-R vs. O2PL-I) For avoidance-based algorithms, the next level of differentiation is concerned with the timing of Write Intention Declarations. As described in Section 3.3.1, avoidance-based algorithms can be pessimistic and require update transactions to declare their write intentions synchronously when a permission fault occurs, or they can be more optimistic and allow the deferral of these declarations until the update transaction enters its commit phase. The CBL algorithms belong to the pessimistic or synchronous camp, while the O2PL algorithms are more optimistic. Comparing the performance of CB-R and O2PL-I provides insight into this tradeoff, as both algorithms retain write permissions only until the end of transaction and both use invalidation for remote update actions.

Write Permission Duration (CB-R vs. CB-A) The next level of choice for avoidance-based algorithms is that of Write Permission Duration. As discussed in Section 3.3.1, write permissions can be associated with a single transaction, or they can be retained by a client site across multiple transactions. The tradeoffs between these two approaches can be directly observed by comparing the performance of CB-R and CB-A, which differ only in this aspect.

Remote Update Actions (O2PL-I vs. O2PL-P) The final choice to be examined here is that of the action performed on remote copies of updated pages. As stated in Section 3.3.4, two options are *invalidation*, which purges such copies from remote caches, and *propagation*, which sends new valid copies of such pages to the remote sites that contain cached copies of them. Comparing the performance of O2PL-I and O2PL-P, which differ only in this respect, will help to isolate the tradeoffs between these two options.

4.2 A Client-Server Performance Model

4.2.1 The System Model

Figure 4 shows the structure of our simulation model, which was constructed using the DeNet discrete event simulation language [Livn90]. It consists of components that model diskless client workstations and a server machine (with disks) that are connected over a simple network. Each client site consists of a *Buffer Manager* that uses an LRU page replacement policy, a *Concurrency Control Manager* that is used either as a simple lock cache or as a full-function lock manager (depending on the cache consistency algorithm in use), a *Resource Manager* that provides CPU service and access to the network, and a *Client Manager* that coordinates the execution of transactions at the client. Each client also has a *Transaction Source* which initiates transactions one-at-a-time at the client site according to the workload model described in the next subsection. Upon completion of one transaction, the source waits for a specified think time and then submits the next transaction. If a transaction aborts, it is resubmitted with the same page reference string. The number of client machines is a parameter to the model.

The server machine is modeled similarly to the clients, but with the following differences: First, the server's Resource Manager manages disks as well as a CPU. Second, its Concurrency Control Manager has the ability to store information about the location of page copies in the system and also manages locks.

Third, there is a *Server Manager* component that coordinates the server's operation; this is analogous to the client's Client Manager. Finally, there is no Transaction Source module at the server since all transactions originate at client workstations.

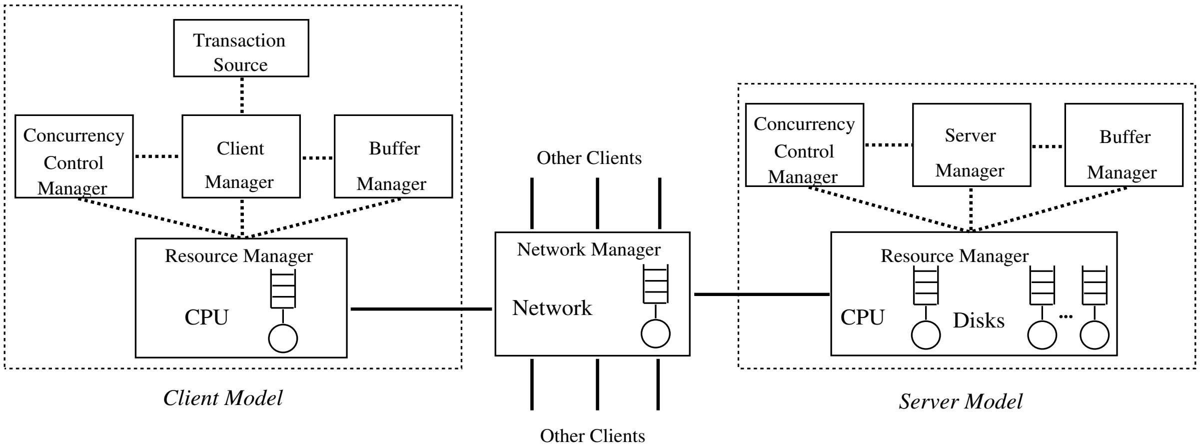

Figure 4: Performance Model of a Client-Server DBMS

Table 2 describes the parameters that are used to specify the system resources and overheads and shows the settings used in this study. The simulated CPUs of the system are managed using a two-level priority scheme. System CPU requests, such as those for message and disk handling, are given priority over user (client transaction) requests. System CPU requests are handled using a FIFO queueing discipline, while a processor-sharing discipline is employed for user requests. Each disk has a FIFO queue of requests; the disk used to service a particular request is chosen uniformly from among all the disks at the server. The disk access time is drawn from a uniform distribution between a specified minimum and maximum. A very simple network model is used in the simulator's *Network Manager* component; the network is modeled as a FIFO server with a specified bandwidth. We did not model the details of the operation of a specific type of network (e.g., Ethernet, token ring, etc.). Rather, the approach we took was to separate the CPU costs of messages from their on-the-wire costs, and to allow the on-the-wire message costs to be adjusted using the network bandwidth parameter. The CPU cost for managing the protocol to send or receive a message is modeled as a fixed number of instructions per message plus an additional charge per message byte.

4.2.2 Client Workloads

Our simulation model provides a simple but flexible mechanism for describing client workloads. The access pattern for each client can be specified separately using the parameters shown in the first column of Table 3. Transactions are represented as a string of page access requests in which some accesses are for reads and others are for writes. Two ranges of database pages can be specified: a hot range and a cold range. The probability of a page access being to a page in the hot range is specified; the remainder of the accesses are directed to cold range pages. For both ranges, the probability that an access to a page in the range will involve a write (in addition to a read) is specified. The parameters also allow the specification of the average number of instructions to be performed at the client for each page read or write, once the proper lock has been obtained.

Table 3 summarizes the workloads that are examined here. The PRIVATE workload has a per-client private hot region that is read and written by each client and a shared cold region that is accessed in a read-only manner by all clients. This workload is intended to model an environment such as a large CAD system,

Parameter	Meaning	Setting
PageSize	Size of a page	4,096 bytes
DatabaseSize	Size of database in pages	1250
NumClients	Number of client workstations	1 to 25
ClientCPU	Instruction rate of client CPU	15 MIPS
ServerCPU	Instruction rate of server CPU	30 MIPS
ClientBufSize	Per-client buffer size	5% or 25% of DB
ServerBufSize	Server buffer size	50% of DB size
ServerDisks	Number of disks at server	2 disks
MinDiskTime	Minimum disk access time	10 millisecond
MaxDiskTime	Maximum disk access time	30 milliseconds
NetBandwidth	Network bandwidth	8 or 80 Mbits/sec
FixedMsgInst	Fixed no. of inst. per message	20,000 instructions
PerByteMsgInst	No. of addl. inst. per msg. byte	10,000 inst. per 4Kb
ControlMsgSize	Size in bytes of a control message	256 bytes
LockInst	Instructions per lock/unlock pair	300 instructions
RegisterCopyInst	Inst. to register/unregister a copy	300 instructions
DiskOverheadInst	CPU Overhead to perform I/O	5000 instructions
DeadlockInterval	Global deadlock detection frequency	1 second (for O2PL)

Table 2: System and Overhead Parameter Settings

Parameter	PRIVATE	HOTCOLD	UNIFORM	FEED
TransSize	16 pages	20 pages	20 pages	5 pages
HotBounds	p to p+24, $p = 25(n\text{-}1)+1$	p to p+49, $p = 50(n\text{-}1)+1$	-	1 to 50
ColdBounds	626 to 1,250	rest of DB	all of DB	rest of DB
HotAccProb	0.8	0.8	-	0.8
ColdAccProb	0.2	0.2	1.0	0.2
HotWrtProb	0.2	0.2	-	1.0/0.0
ColdWrtProb	0.0	0.2	0.2	0.0/0.0
PerPageInst	30,000	30,000	30,000	30,000
ThinkTime	0	0	0	0

Table 3: Workload Parameter Settings for Client n

where each user has a portion of the design that they are currently working on while accessing additional design parts from a shared library of components. The HOTCOLD workload has a high degree of locality per client and a moderate amount of sharing and data contention among clients. UNIFORM is a low-locality, moderate write probability workload which is used to examine the consistency algorithms in a case where caching is not expected to pay off significantly. This workload has a higher level of data contention than HOTCOLD. Finally, the FEED workload represents an application involving a highly directional information flow, such as one might expect in a stock quotation system; one site produces data while all the other sites consume it.

4.3 Experiments and Results

In this section we present results from several performance experiments involving the algorithms described in Section 4.1. The main performance metric presented is system throughput (measured in transactions per second).[11] The throughput results are, of course, dependent on the particular settings chosen for the various physical system resource parameters. For example, the relative performance of the algorithms in a disk-bound system may differ greatly from that in a CPU-bound system. Thus, while the throughput results show performance characteristics in what we consider to be a reasonable environment, we also present various auxiliary performance measures, such as message and disk I/O counts, to provide additional insights into the fundamental tradeoffs among the algorithms.

Auxiliary metrics that are presented as "per commit" values are computed by taking the total count for the given metric (e.g., the total number of messages routed through the network) over the duration of the simulation run and then dividing by the number of transactions that committed during that run. As a result, these averages also take into account work that was done on behalf of aborted transactions. To ensure the statistical validity of the results presented here, we verified that the 90% confidence intervals for transaction response times (computed using batch means) were sufficiently tight. The size of the confidence intervals was within a few percent of the mean in all cases, which is more than sufficient for our purposes.

In the sections that follow, we focus on a system configuration in which each client has a large cache (25% of the active database size) and the network bandwidth is set to the lower value in Table 2 (8 Mbits/sec). The network speed was chosen to approximate the speed of an Ethernet, reduced slightly to account for bandwidth lost to collisions, etc. The large client cache size tends to reduce the performance impact of server I/O. Therefore, the combination of these settings tends to emphasize the performance impact of message behavior, which plays a role in all four of the tradeoffs listed in Table 1. However, I/O and transaction aborts also factor into the comparisons and will be discussed where appropriate. Finally, although this section focuses on results from a limited set of experiments, it should be emphasized that we have run numerous experiments with a variety of different parameter settings and workloads. Many of these experiments are described in [Care91a, Fran92a, Fran93a].

4.3.1 The PRIVATE Workload

We first examine performance results for the PRIVATE workload. As described in Section 4.2.2, PRIVATE has a high degree of locality per client, and it has no read-write or write-write data sharing. Figure 5 shows the total system throughput for the PRIVATE workload as the number of clients in the system is increased from 1 to 25. In this experiment, the invalidation-based O2PL algorithms and Callback locking algorithms provide the best performance. The detection-based C2PL algorithm has lower throughput, and B2PL, which does not allow inter-transaction caching, has the poorest performance by a significant margin. In this experiment (and in most of the others), B2PL suffers due to high message volumes and server disk I/O levels because it is unable to exploit client memory for storing data pages across transaction boundaries.

In order to see what insight these results can provide into the performance tradeoffs for this workload, it is helpful to examine pairs of algorithms as discussed in Section 4.1.4. The first tradeoff we examine here is based on the choice of Invalid Access Prevention. In this experiment, the avoidance-based algorithms all significantly outperform the detection-based C2PL algorithm throughout the range of client populations. This behavior is due to the server CPU overhead and the path-length resulting from the number of messages sent per transaction. Focusing on C2PL and CB-A, as can be seen in Figure 6, C2PL requires nearly 40 messages per transaction (on average) in this experiment, while CB-A requires only 12. This difference is

[11] Because the simulation uses a closed queueing model, throughput and response time are equivalently informative metrics.

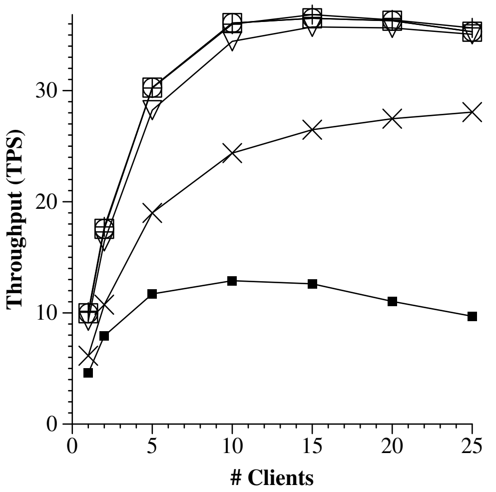

Figure 5: Throughput
(PRIVATE, 25% Client Cache, Slow Net)

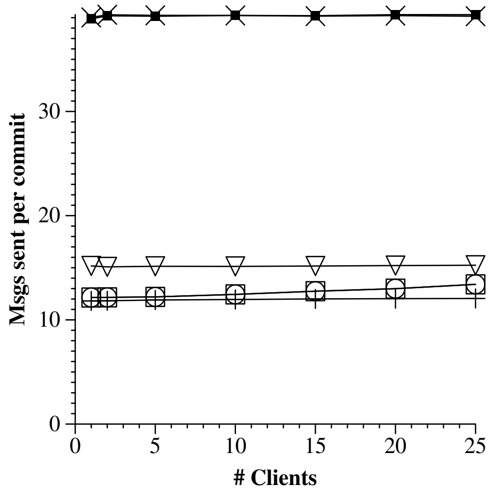

Figure 6: Messages Sent/Commit
(PRIVATE, 25% Client Cache, Slow Net)

because the pessimistic, detection-based C2PL algorithm sends a message to the server on every initial page access — even for pages that it has cached locally. In fact, C2PL sends the same number of messages as the non-caching B2PL algorithm, though it sends many fewer bytes because fewer of the replies from the server contain page copies than for B2PL. This difference in message requirements in the absence of data contention is one reason why most of the detection-based algorithms that have been proposed (see Figure 2) include some amount of optimism.

The next design decision to be examined is the choice of Write Intention Declaration timing. Because of the lack of read-write and write-write sharing in this workload, however, this choice has only a minor impact on performance here. As can be seen in Figure 5, CB-R performs only slightly below O2PL-I under this workload. With no data conflicts, write intention declarations only require a round trip message to the server as no remote clients ever need to be contacted, so O2PL-I gains only a small savings in messages by deferring its write intention declarations until commit time. Returning to the message counts shown in Figure 6, it can be seen that while the message requirements for CB-R remain constant as clients are added, there is a slight rise in the message requirements for O2PL-I. This rise is due to the cost of distributed deadlock detection, which is not required by CB-R. Finally, it should be noted that the absence of data conflicts means that the differences in abort rates between pessimism (CB-R) and optimism (O2PL-I) are simply not an issue for this workload.

The tradeoffs for the choice of Write Permission Duration can be seen by comparing the performance and message sending behavior of CB-A, which retains permissions across transaction boundaries, and CB-R, which gives up a write permission when the transaction that obtained it completes. Under the PRIVATE workload, CB-A declares a write intention on a page copy at most once for the duration of the page copy's residence in the cache, as permissions are never called back by remote clients under this workload. Thus, CB-A consistently sends fewer messages than CB-R. This results in a message savings and a slight throughput advantage for CB-A in this case (in fact, CB-A performs as well as O2PL-I does here).

Finally, it should be noted that the of Remote Update Action choice does not impact performance under the PRIVATE workload. This is again due to the absence of read-write and write-write sharing. No remote updates ever occur, so O2PL-I and O2PL-P provide similar throughput here.

4.3.2 The HOTCOLD Workload

Figure 7 shows the throughput results for the HOTCOLD workload with the large client caches and slow network. As described in Section 4.2.2, HOTCOLD has high locality per client, but unlike the PRIVATE workload, it also has read-write and write-write sharing among the clients. Despite this sharing, however, the relative throughput for each of the algorithms (except for O2PL-P, which is discussed below) is similar to what was observed in the PRIVATE case. That is, the avoidance-based algorithms perform better than C2PL, and the non-caching B2PL algorithm has the worst performance.

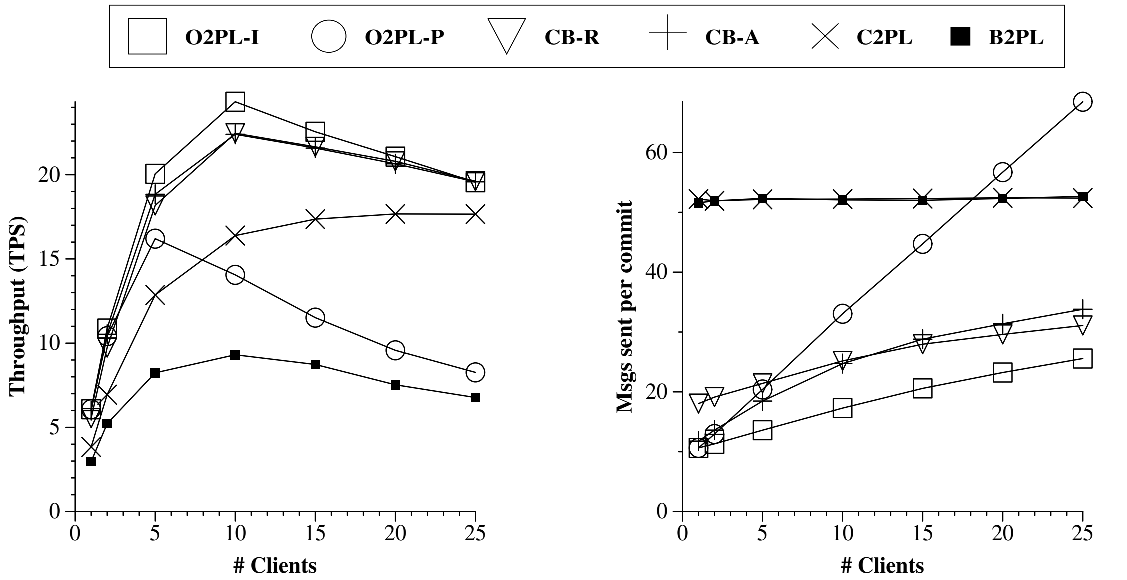

Figure 7: Throughput
(HOTCOLD, 25% Client Cache, Slow Net)

Figure 8: Messages Sent/Commit
(HOTCOLD, 25% Client Cache, Slow Net)

The introduction of read-write and write-write sharing, however, raises several additional tradeoffs for cache consistency maintenance. Many of the tradeoffs can be seen in Figure 8, which shows the number of messages sent per committed transaction. The first tradeoff that we discuss is that between detection-based and avoidance-based Invalid Access Prevention. As can be seen in the figure, the message counts for the detection-based C2PL algorithm are independent of the number of clients in this case, while the avoidance-based algorithms all send more messages per transaction as clients are added (unlike under the PRIVATE workload results of Figure 6). The additional messages used by the avoidance-based algorithms are for implementing remote update actions (callbacks, invalidations, propagations, etc.) at clients that possess cached copies of affected pages. As more clients are added, the number of cached copies for any given page increases, so more messages for remote update actions are required. However, it should be noted that the number of messages sent by the avoidance-based CB-A algorithm remains substantially lower than for the detection-based C2PL algorithm throughout the range of client populations explored in this experiment.

The next tradeoff of interest involves Write Intention Declaration. The tradeoffs between declaring Write Intentions synchronously, during transaction execution (as in CB-R), or deferring such declarations until transaction commit (as in O2PL-I) are slightly different under HOTCOLD than they were under the PRIVATE workload. Comparing the number of messages sent under HOTCOLD (Figure 8) and under

PRIVATE (Figure 6), the difference between CB-R and O2PL-I is greater here than under PRIVATE for two reasons: first, each transaction updates more pages under HOTCOLD than under PRIVATE, and second, some intention declarations result in Remote Update Actions here. Since CB-R declares intentions one-at-a-time, multiple declaration messages are sent, and it is possible that multiple callback requests will be sent to some remote clients during a transaction. In contrast, by deferring Write Intention Declarations, O2PL-I sends only a single declaration message to the server, and it sends at most one request for Remote Update Actions to each remote client. This difference has only a small impact on throughput here, which disappears as more clients are added and the server disks become the dominant resource. Finally, while the Write Intention Declaration decision also impacts the transaction abort rate (as discussed in Section 4.1.4), the abort rate does not play a significant factor in this experiment due to a fairly low level of data contention.

The tradeoffs involving Write Permission Duration are affected in an interesting way by the introduction of read-write and write-write sharing, as can be seen by comparing the message behavior of CB-R and CB-A in Figure 8. With fewer clients, CB-R, which gives up write permissions at the end of a transaction, sends more messages than CB-A, which retains permissions across transaction boundaries. However, as clients are added, the amount of sharing increases; more write permission callbacks occur, so the number of messages sent by CB-A increases at a faster rate than for CB-R. CB-A has higher message requirements than CB-R at 15 clients and beyond. Due to the fact that the disk becomes the dominant resource in this region, however, the two Callback algorithms deliver similar performance.

Finally, the choice of Remote Update Action has a very significant impact in this experiment, due to presence of read-write and write-write sharing. In contrast to the invalidation-based O2PL-I algorithm, O2PL-P suffers a substantial degradation in performance beyond five clients; it eventually performs even below the level of C2PL. The reason for O2PL-P's poor performance in this case is a dramatic increase in message volume as clients are added. At 25 clients, O2PL-P sends almost three times more data through the network (about 120 Kbytes per commit) than O2PL-I (which sends about 43 Kbytes per commit). This increase is due to the messages needed by O2PL-P to propagate updated pages to remote clients. At 25 clients, it sends propagations to an average of 13 (!) remote clients per transaction. Furthermore, the vast majority of these propagations are "wasted" — that is, the remote copies are either propagated to again, or else dropped from the cache, before the previous propagated value is ever actually used. It should be noted that the large number of involved sites is due to the large client caches; when O2PL-P is used with smaller client caches, wasted propagations are reduced, as unimportant pages tend to be quickly pushed out of the caches before another propagation occurs. This experiment demonstrates that using propagation to implement Remote Update Actions is a rather dangerous policy. Its performance is very sensitive to the size of the client caches, and it has the potential to more than nullify the performance benefits of caching in some cases.

4.3.3 The UNIFORM Workload

The third workload that we examine is the UNIFORM workload, which has no per-client locality and, as a result, has a higher level of data contention and benefits less from caching than the HOTCOLD workload. The throughput results and message counts are shown in Figures 9 and 10, respectively. In terms of throughput, UNIFORM's combination of no locality and high data contention reduces the magnitude of the performance differences among the caching algorithms. In terms of message counts, Figure 10 shows that the tradeoffs for Invalid Access Prevention are somewhat different here than in the previous cases. As in the HOTCOLD workload, the number of messages required by the avoidance-based algorithms increases with the number

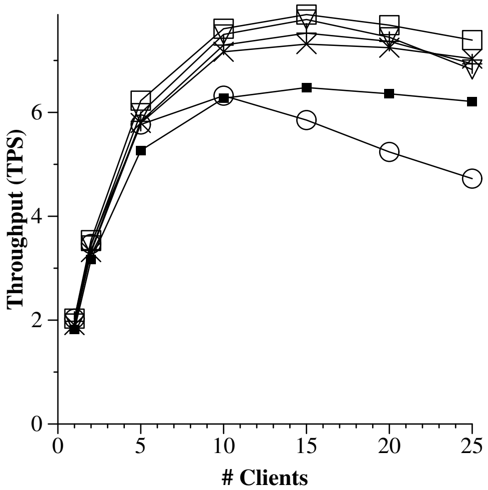

Figure 9: Throughput
(UNIFORM, 25% Client Cache, Slow Net)

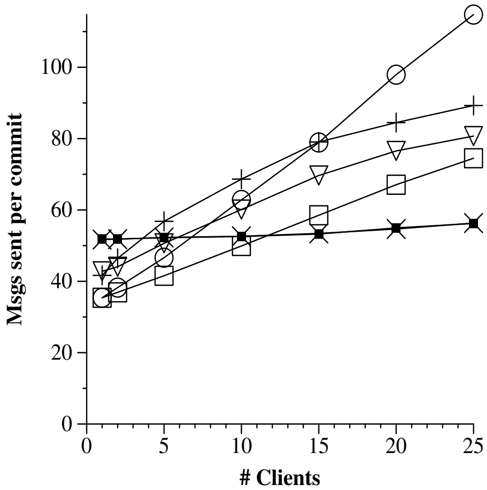

Figure 10: Messages Sent/Commit
(UNIFORM, 25% Client Cache, Slow Net)

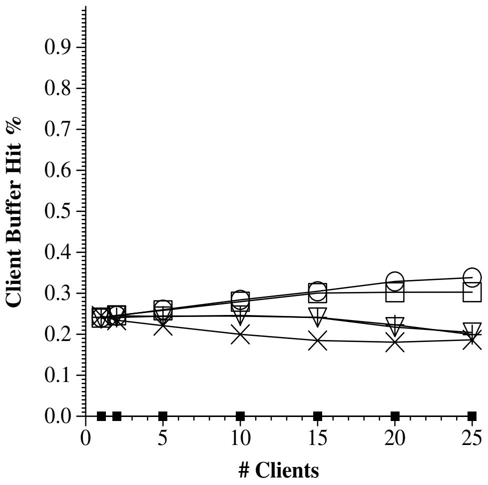

Figure 11: Client Hit Rate
(UNIFORM, 25% Client Cache, Slow Net)

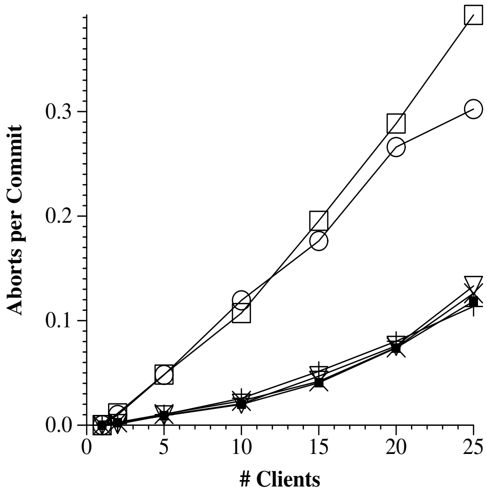

Figure 12: Aborts/Commit
(UNIFORM, 25% Client Cache, Slow Net)

of clients, whereas the requirements of C2PL remain nearly constant (increasing slightly due to aborted transactions). Unlike the HOTCOLD case, however, all of the avoidance-based algorithms require more messages than the detection-based C2PL algorithm beyond 5-10 clients.

To understand why detection leads to fewer messages than avoidance in this low-locality situation, it is useful to examine the message tradeoffs made by the avoidance-based algorithms. Under CB-A (as well as the other avoidance-based algorithms), the permission to read a page is effectively cached along with the page. Thus, when a client wishes to read a page that it has in its cache, it can do so without contacting the server. Compared to C2PL, this saves two messages per initial read. However, if some remote client wishes to write a page that a client has cached, then a callback message will arrive and a reply must be sent. This is a net cost of two messages per write compared to C2PL. Furthermore, CB-A has analogous tradeoffs for pages on which write permissions are retained: it enjoys a savings of two messages if the page is written locally, and pays a price of two messages if the page is read remotely. CB-A's avoidance approach therefore yields a net loss if a page is less likely to be read locally than written remotely, and the retention of write permissions is a net loss if the page is less likely to be written locally than read remotely. The absence of locality in the UNIFORM workload means that both of the tradeoffs made by CB-A become net losses as more clients are added. Similar tradeoffs are made by the other avoidance-based algorithms.

In addition to messages, the choice of an Invalid Access Prevention technique also has an impact on I/O requirements. Figure 11 shows the average hit rate across all client caches for the UNIFORM workload.[12] As can be seen in the figure, the avoidance-based algorithms all have higher client cache hit rates than C2PL. In this experiment, the O2PL algorithms have inflated client buffer hit rates due to aborted transactions (as indicated in Figure 12). However, while CB-A has a nearly identical abort rate to C2PL, it has a noticeably better client hit rate. The reason for this difference is that size of the client caches under C2PL is *effectively* smaller than under the avoidance-based algorithms because of the presence of *invalid* pages. These invalid pages consume cache space that could otherwise be used for holding valid pages. In contrast, since the avoidance-based algorithms remove pages from client caches as they become invalid, they allow the entire cache to be used for valid pages. This effect is greatest in the range of 10-15 clients here. Beyond this point, CB-A incurs an increasing rate of page invalidations due to the large number of clients. These invalidations reduce the extent to which CB-A is able to utilize the client caches; beyond a client population of 15, significant numbers of client cache slots simply remain empty under CB-A.

The increased data contention of the UNIFORM workload also brings out the downside of the optimistic approach of deferring Write Intention Declarations. As can be seen in Figure 12, the semi-optimistic O2PL-I algorithm aborts as many as 0.4 transactions for every transaction that it commits in this experiment. In comparison, the pessimistic CB-R algorithm aborts about one third as many transactions. Interestingly, despite this difference, O2PL-I obtains roughly 10% higher throughput than CB-A (see Figure 9). This is because the cost of aborts in this experiment is rather low due to cache hits that occur when aborted transactions run again. However, as shown in [Fran92a, Fran93a], the high abort rate of O2PL-I can cause it to have significantly worse performance than CB-R if data contention is increased further.

In terms of the Write Permission Duration, the high sharing level and lack of locality of the UNIFORM workload results in CB-R sending fewer messages than CB-A across the range of client populations (Figure 10) and thereby having a slight performance advantage across most of the range (Figure 9). As discussed previously, retaining Write Permissions is a net loss for CB-A if a page is less likely to be written locally

[12] A cache request results in a hit only if a *valid* copy of the page is found in the local cache.

than it is to be read remotely. The lack of locality and the update probability in the UNIFORM workload thus work to the disadvantage of CB-A.

The effects of using propagation for Remote Update Actions are similar here to those seen in the HOT-COLD workload. In this case, however, O2PL-P ultimately performs worse than even B2PL, which does no inter-transaction caching. Although propagation does give O2PL-P a slight advantage in terms of the client hit rate (Figure 11), the cost of sending propagations that go unused is much higher here than the benefit gained from those propagations that are indeed eventually used.

4.3.4 The FEED Workload

The last workload to be examined here is the FEED workload. As discussed in Section 4.2.2, FEED is intended to model an information service environment, such as a stock quotation system, where many clients read data from an information source. In this workload, one client acts as the source, reading and updating pages, while the remainder of the clients act as consumers, only reading the data. We include this workload primarily to demonstrate a case where using propagation as the Remote Update Action can be beneficial; thus, we focus only on the performance of O2PL-I and O2PL-P here. Figure 13 shows the throughput results for O2PL-I and O2PL-P under this workload. The dashed lines show the throughput of the source (Client #1) while the solid lines show the aggregate throughput of the remaining clients. In this workload, O2PL-P significantly outperforms O2PL-I. The reason for this is that propagation gives the consumers a much higher client cache hit rate, as is shown in Figure 14. This improvement in hit rate reduces the path length of the reader transactions. Furthermore, due to the high degree of client access locality in this workload, many fewer propagations are wasted than in the UNIFORM workload.

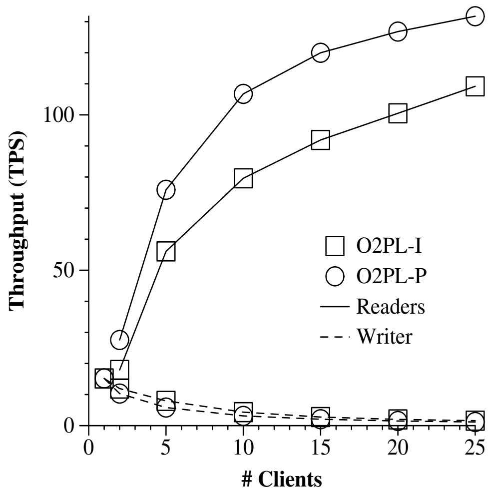

Figure 13: Throughput
(FEED, 25% Client Cache, Slow Net)

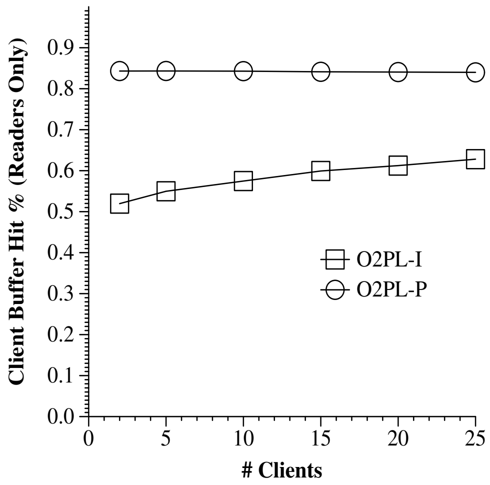

Figure 14: Client Cache Hit % (Readers only)
(FEED, 25% Client Cache, Slow Net)

The FEED workload shows the potential benefits of propagation in specialized workloads, but the HOT-COLD and UNIFORM workloads both demonstrated that propagation can substantially reduce performance in other situations. We have also developed and investigated *dynamic* algorithms that can adaptively choose between invalidation and propagation on a page-by-page basis at each client. These algorithms will be discussed briefly in Section 5.

4.4 Summarizing The Results

In this section of the paper we have compared the performance of six different cache consistency maintenance algorithms — drawn from three algorithm families — using four different workloads. Here, we briefly review the results in terms of the insight that they offer regarding the design tradeoffs for transactional cache consistency maintenance algorithms.

The tradeoffs between using avoidance and detection for Invalid Access Prevention were examined by comparing C2PL and CB-A. Both of these algorithms are pessimistic, so the avoidance/detection choice was seen to have a large impact on the number of messages sent. C2PL validates its cached data prior to accessing it, and thus sends a round-trip message to the server on every initial access regardless of whether any read-write or write-write sharing is occurring. In contrast, CB-A, being avoidance-based, is able to read its cached data without contacting the server. However, CB-A, was seen to be sensitive to the type and degree of sharing in the system, as increased data sharing results in additional callback and downgrade messages. In addition to message count differences, the results for the UNIFORM workload demonstrated that the choice of an Invalid Access Prevention method can have an impact on I/O requirements and on the volume of data transferred. This is because detection-based approaches allow out-of-date pages to remain in client caches, reducing the effective size of those caches.

The tradeoff between synchronous and deferred Write Intention Declaration (examined by comparing CB-R and O2PL-I) is a tradeoff between pessimism and optimism, affecting both the number of messages required to complete transactions and the transaction abort rate. With no read-write or write-write sharing (e.g., under the PRIVATE workload), the approaches were seen to be roughly equal in performance. If sharing is present, then deferring declarations can save messages. However, if sharing increases to the point were data contention arises, deferring declarations can lead to significantly higher abort rates; transaction aborts can result in higher resource utilization due to lost work [Fran92a, Fran93a], though this effect did not significantly hurt throughput in the workloads examined here. Whether or not aborts impact overall throughput, a high abort rate may be intolerable for users in some highly-interactive applications.

The choice of Write Permission Duration was examined by comparing the performance of CB-R, which retains write permissions only for the duration of a single transaction, and CB-A, which retains write permissions across transaction boundaries. The impact of this tradeoff is on the number of messages sent. In general, retaining write permissions is a net win if a page is more likely to be *updated* at the client that holds the permissions than to be *read* at another client. Thus, CB-A was seen to have a slight advantage over CB-R in the PRIVATE workload, while it had a slight disadvantage in the other workloads examined.

The fourth design choice analyzed was the choice of Remote Update Action, as demonstrated by O2PL-I versus O2PL-P. In the PRIVATE workload, this choice has no effect because there is never a need for a remote update action. The other workloads, however, demonstrated that this choice can have a dramatic impact on performance. In the majority of the cases, invalidation was seen to be the right choice. Propagation was shown to be dangerously sensitive to the level of sharing present in the workload, and hence to the client cache size — with larger caches, the potential for sequential sharing across clients increases. In contrast, invalidation was seen to be much more stable in its performance characteristics. The FEED workload, however, demonstrated that there are indeed cases where propagation can be useful.

5 Related Work

As explained earlier, the performance experiments described in this paper have covered only a portion of the large design space available for cache consistency maintenance. In particular, our work has focused on algorithms that lie on the avoidance-based side of the taxonomy in Section 3. The design decisions for detection-based algorithms, however, each have avoidance-based analogs. In the following, we apply the insight gained from our experiments to predict the relative performance characteristics of other published algorithms that appear in the taxonomy. We then briefly discuss related work in other domains.

5.1 Other Proposed Algorithms

5.1.1 Optimistic Detection-Based Algorithms

The first published paper to analyze transactional cache consistency algorithms for client-server OODBMSs was [Wilk90]. In that paper, two algorithms were proposed and studied. One algorithm, called *Cache Locks*, is a detection-based algorithm that defers validation of transactions until commit time. Special lock modes and long-running "envelope transactions" are used to determine when transactions have accessed stale data. Cache Locks is an optimistic algorithm in the sense that lock requests are not sent to the server during transaction execution. At commit time, a transaction sends its read and write sets to the server, and the server attempts to obtain the necessary shared and exclusive locks. Special lock modes for locks held on cached copies by envelope transactions indicate whether or not the copies accessed by the transaction were valid. If it is determined that the transaction accessed stale cached data, then it is aborted. In order to reduce the likelihood of aborts due to stale data, the server piggybacks *notifications* about modified pages on its replies to client requests. These notifications inform the client that it should mark its copies of the affected data as invalid (hence, it is an invalidation-based algorithm) and abort any ongoing transactions that have accessed those data items.

More recently, an optimistic algorithm with notifications has been proposed for the Thor system at MIT [Adya95]. This algorithm, called Adaptive Optimistic Concurrency Control (AOCC), is similar to the Cache Locks algorithm; it also includes support for transactions that access data from multiple servers (which is beyond the scope of both [Wilk90] and this paper). Rather than using lock modes to represent invalid cached copies, AOCC maintains an *invalid set* for each client in order to keep track of which copies of the data items cached at a client have been made invalid. As described in [Adya95], AOCC uses a combination of invalidation and propagation for Remote Update Actions. As in Cache Locks, notifications are piggybacked on messages sent to clients, and such notifications invalidate cached copies. However, when a transaction is aborted due to a detected inconsistency, AOCC immediately piggybacks new copies (i.e. propagations) of the invalid items on the abort acknowledgement that it sends to the client.

As discussed in the previous section, a potential problem with detection-based policies is an increase in communication due to the need to check page validity with the server. The comparisons of C2PL and CB-A showed that this cost can be significant if a pessimistic (i.e., synchronous on each initial access) approach to validity checking is used. For this reason, both Cache Locks and AOCC use the more optimistic approach of deferring such checks until transaction commit time. As shown in [Adya95], such optimism, combined with piggybacking of notifications, can significantly reduce the number of messages required for consistency checking; of course, this comes at the expense of a possible rise in transaction aborts. Whether such a tradeoff is beneficial depends on several factors including the level of contention in the workload, the cost of wasted work due to aborts, and the tolerance of the application to a higher abort rate.

One major difference between Cache Locks and AOCC is that Cache Locks uses invalidation for remote update actions while AOCC uses a combination of invalidation (in the absence of concurrent conflicts) and propagation (when a conflict has caused an abort). These propagations can be very useful in reducing the costs of transaction aborts as long as the aborted transactions are restarted immediately and tend to access the same items as they did in their previous incarnation(s); otherwise, inefficiencies that were identified for propagation in Section 4.3 may be incurred by this approach as well.

5.1.2 Notify Locks

The second algorithm proposed in [Wilk90], *Notify Locks*, is an avoidance-based algorithm. It is similar to the O2PL-P algorithm described previously in that it defers Write Intention Declaration until the end of transaction execution and uses propagation for remote update actions. When a transaction wishes to commit, it sends copies of the updated data items back to the server. The server then sends notification messages to any clients that hold copies of the updated items; these messages contain the new values of those items. A major difference between Notify Locks and O2PL-P is that with Notify Locks, the arrival of a notification *preempts* any ongoing transactions that have accessed the changed items. In contrast, O2PL-P blocks notification requests that conflict with read locks held by ongoing transactions. Because of the preemption approach used by Notify Locks, committing a transaction requires (sometimes multiple) handshakes between the client and the server to avoid race conditions at commit time. The performance tradeoffs between the wait and preempt policies, however, have not been addressed in this study. Of course, because Notify Locks uses propagation, it is clearly subject to the performance problems that we saw for O2PL-P. This effect was not detected in [Wilk90] because that study used a probabilistic cache model that assumed that cache hit probabilities were independent of cache size.

5.1.3 No-wait Locking

No-wait locking algorithms were studied in [Wang91]. No-wait algorithms are detection-based algorithms that try to hide the latency of validations at the server by performing validity checking asynchronously. As with all detection-based algorithms, transactions must abort if they are found to have accessed stale data. By initiating the validity checks before commit time, however, the window during which data can become invalid is shortened compared to Cache Locks and AOCC. As stated in Section 4.1.4, asynchrony does not reduce the total work required, and thus, will not improve performance in a highly utilized system (e.g., if the server is a bottleneck). The performance results of [Wang91] showed that an algorithm similar to CB-R typically performed as well as or better than no-wait Locking.

To reduce the possibility of stale data access, the no-wait algorithm was extended in [Wang91] with a propagation-based notification hint scheme. The performance of this algorithm, called No-Wait Locking with Notifications, was then examined. The results of that study showed (as we did in [Care91a]) that the cost of propagations typically outweighs their potential benefits. An invalidation-based notification scheme could avoid this problem, but such a scheme was not studied in [Wang91].

5.1.4 Dynamic Optimistic Two-Phase Locking

The two remaining algorithms shown in the taxonomy of Figures 2 and 3 are variants of O2PL that choose dynamically between invalidation and propagation on a page-by-page basis. The original dynamic algorithm (O2PL-Dynamic) was introduced in [Care91a]. This algorithm used a simple heuristic that would initially propagate an update to a remotely cached page copy, switching to invalidation the next time if the propagation went unused. An improved heuristic (called O2PL-NewDynamic), which initially favors invalidation over propagation, was described and studied in [Fran92a, Fran93a]. Those studies showed that by favoring invalidation, O2PL-NewDynamic was able to match the performance of O2PL-I in those workloads where it had the best performance (i.e., most workloads tested), and to approach the superior performance of O2PL-P in the FEED workload (which is the once case where it provided the best performance).

5.2 Other Related Work

In addition to work that is directly related to transactional cache consistency for client-server database systems, there has been a tremendous amount of work on cache consistency for other types of systems including database as well as non-database environments. While an in-depth treatment of this work is beyond the scope of this paper, this section briefly discusses this other related work.

5.2.1 Shared-Disk Database Systems

Transactional cache consistency is required in any database system that supports dynamic caching. One such class of systems is shared-disk (or data sharing) parallel database systems, which consist of multiple nodes with private processors and memory that share a common disk pool [Bhid88]. While similar in some respects to the client-server database systems addressed in this study, they differ in three significant ways. First, since nodes are not assigned to individual users, they are likely to have less locality of access at the individual nodes (i.e., all nodes are likely to have similar data affinities). Secondly, the cost of communication among nodes in a shared-disk environment is substantially lower than would be expected in the local area network of a page server DBMS. Thirdly, the structure of a shared disk system is peer-to-peer, as opposed to the client-server structure of a page server system, so many of the environmental considerations raised in Section 2.2.2 do not apply.

A number of papers on shared-disk caching performance have been written by a group at IBM Yorktown. One of their earlier papers examined cache consistency protocols that were integrated with the global lock manager of a shared-disk system [Dias87]. Later work has addressed the impact of data skew and contention for a range of possible algorithms [Dan90a, Dan90b], the interaction between private and shared buffering [Dan91] (similar to the interactions between client buffers and the server buffer), and extensions to callback-style algorithms [Dan92]. Other related work in this area includes the work of Mohan and Narang [Moha91], Rahm [Rahm93], and Lomet [Lome94]. An algorithm that could dynamically adjust the granularity at which locking and coherency were managed for a shared-disk DBMS was introduced in [Josh91]. This approach was later extended for use in page server environments in [Care94b].

5.2.2 Non-Database Environments

Of course, cache consistency issues arise in other types of distributed and/or parallel systems as well, such as multi-processor architectures, distributed file systems, and distributed shared memory systems. While there are many similarities between the solutions available for these environments and those available in client-server database systems, there are significant environmental differences that impact the design alternatives and their inherent tradeoffs. Important areas of difference, from the standpoint of cache consistency maintenance, include: correctness criteria, granularity of caching, cost tradeoffs, and workload characteristics.

Much of the early work in cache consistency maintenance was done in the context of shared-memory multiprocessors. A number of early protocols for such systems were studied in [Arch86]; a more recent survey appears in [Sten90]. These systems differ along all four of the dimensions listed above. For example, database systems support serializable transactions, while the consistency requirements for multiprocessors are at the level of individual memory accesses. Nevertheless, the basic consistency mechanisms, such as write-broadcast and write-invalidate [Arch86], or directories of cached copies (e.g., [Agar88]), are similar.

Client caching has been used in distributed file systems since some of the earliest work in the area (e.g., DFS [Stur80]). Many distributed file systems that support some form of client caching have been proposed and built. These include Andrew [Howa88] and Sprite [Nels88], which both used callback-style algorithms. A survey of distributed file systems can be found in [Levy90]. As with page server DBMSs, these systems use client caching to improve performance and scalability. However, they support much less stringent notions of correctness in terms of both concurrency and failure semantics. Furthermore, distributed file systems are typically designed for workloads in which read-write sharing is rare (e.g., [Bake91]). Caching is often done at a fairly coarse granularity, such as entire files or large portions of files.

Finally, Distributed Shared Memory (DSM) systems [Li89, Nitz91] provide the abstraction of a shared *virtual* memory address space that spans the nodes of a distributed system. DSM implementations allow multiple readers to access the same page, so DSM systems must deal with the problems raised by dynamic replication. Unlike multiprocessor caching, which relies heavily on hardware support, DSMs are typically implemented in software with only minimal hardware assists. Because pages are the unit of consistency, DSMs are similar to page servers with respect to granularity. In terms of cost tradeoffs, DSMs are again closer to page servers than are shared-memory multiprocessors, because messages are required for consistency (although if the DSM is built on a multiprocessor then messages may be less expensive). The main differences, therefore, lie in the area of correctness criteria. DSM systems typically do not provide transactional support for concurrency or for failure. Consistency is provided in terms of individual reads and writes, so any higher-level consistency model must be implemented on top of the DSM.

6 Conclusions

In this paper we began by describing the potential benefits of caching in client-server database systems based on the data-shipping approach. The introduction of caching raises the need for mechanisms to ensure that transaction semantics are not violated as a result of dynamic replication. We refer to such mechanisms as *transactional cache consistency* maintenance algorithms. The client-server DBMS server environment was shown to have certain characteristics that impose constraints on the design of such algorithms. We presented a taxonomy that describes the design space and showed how it encompasses the algorithms that have been proposed in the literature. Six algorithms, taken from three different families, were then described in more detail and analyzed. These algorithms were used to explore many of the tradeoffs inherent in the design choices of the taxonomy. The insight gained was then used to reflect upon the characteristics of other algorithms that appear in the taxonomy.

The choice of avoidance versus detection for preventing invalid access was seen to have a significant impact on the number of messages sent for maintaining consistency. Under pessimistic-style approaches, avoidance typically sends far fewer messages than detection. Concern about communication requirements has led designers of detection-based algorithms to employ optimistic techniques that defer consistency actions or perform them asynchronously. Such techniques reduce the number of messages sent at the possible expense of increasing the probability of transaction aborts. A secondary effect of the choice of invalid access prevention is that avoidance-based techniques are able to more efficiently use client caches, as they allow only valid data to reside in the caches. Efficient cache usage can reduce the number of pages that must be obtained from the server, saving messages, message volume, and possibly even server I/O. Several of the detection-based algorithms have been extended with notification hints that help remove invalid pages from client caches. These hints reduce the potential for aborts due to accessing invalid pages and help to ensure more efficient use of the caches.

The choice between synchronous and deferred declaration of write intentions was seen to be a tradeoff between the number of messages sent and the transaction abort rate. Deferring declarations introduces another form of optimism, which can reduce messages but may also increase aborts. A third design decision, the duration of write permissions, was examined using two variants of Callback Locking. The tradeoff lies in the number of messages sent, and is workload-dependent. In situations with high-locality and low data conflict rates, retaining write permissions across transaction boundaries was seen to save messages, while with low-locality and high data conflict rates retaining write permissions was shown to result in a net increase in messages sent. These observations indicate that a dynamic algorithm that can choose between these two strategies is likely to perform well. Finally, the choice between invalidating remote copies and propagating changes to them was investigated by comparing two variants of the Optimistic Two-Phase Locking approach. Invalidation was seen to be quite robust in the face of changes to a number of workload and configuration parameters. In contrast, propagation was shown to be dangerously sensitive to the level of sequential sharing and to the client cache sizes; however, it was also demonstrated to be beneficial in a workload meant to model an information dissemination environment. In the absence of a dynamic approach or detailed information about client access patterns, invalidation is clearly the safest choice for most situations.

The work reported here has been extended in several ways. The extension of these techniques to client disk caching was investigated in [Fran93b]. Issues that arise when clients are allowed to obtain data from each other (in addition to servers) were studied in [Fran92b]. More recently, callback-style approaches have been extended to support multiple granularities of concurrency control and cache consistency maintenance [Care94b, Chu94]. Current trends in client-server database systems raise additional challenges that must be addressed as well. In particular, the merging of Relational and Object technologies requires systems that can efficiently support both the navigational style of data access assumed in this study and the query-oriented access typically associated with relational systems. This is a major focus of our ongoing work. We are also investigating the extension of these techniques to systems in wider-area networks.

References

[Adya95] Adya, A., Gruber, R., Liskov, B., Maheshwari, U., "Efficient Optimistic Concurrency Control Using Loosely Synchronized Clocks" *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, San Jose, CA, June, 1995.

[Agar88] Agarawal, A., Simoni, R., Hennessy, J., Horowitz, M., "An Evaluation of Directory Schemes for Cache Coherence", *Proceedings of the 15th International Symposium on Computer Architecture*, Honolulu, June, 1988.

[Alsb76] Alsberg, P., Day, J., "Principles for Resilient Sharing of Distributed Resources", *Proceedings of the 2nd International Conference on Software Engineering*, IEEE, San Francisco, 1976.

[Amsa95] Amsaleg, L., Franklin, M., Gruber, O., "Efficient Incremental Garbage Collection for Client-Server Object Database Systems", Proc. of the 21st International Conference on Very Large Data Bases, Zurich, Switzerland, September, 1995.

[Arch86] Archibald, J., Baer, J., "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model", *ACM Transactions on Computer Systems*, 4(4), November, 1986.

[Bake91] Baker, M., Hartman, J., Kuper, M., Shirriff, K., Ousterhout, J., "Measurements of a Distributed File System", *Proceedings of the 13th International Symposium on Operating System Principles*, Pacific Grove, CA, October, 1991.

[Bern87] Bernstein, P., Hadzilacos, V., Goodman, N., *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.

[Bhid88] Bhide, A., Stonebraker, M., "An Analysis of Three Transaction Processing Architectures," *Proceedings of the 14th International Conference on Very Large Data Bases*, Los Angeles, August, 1988.

[Butt91] Butterworth, P., Otis, A., Stien, J., "The GemStone Object Database System", *Communications of the ACM*, 34(10), October, 1991.

[Care91a] Carey, M., Franklin, M., Livny, M., Shekita, E., "Data Caching Tradeoffs in Client-Server DBMS Architectures", *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, Denver, June, 1991.

[Care91b] Carey, M. and Livny, M., "Conflict Detection Tradeoffs for Replicated Data", *ACM Transactions on Database Systems*, 16(4), December, 1991.

[Care93] Carey, M., DeWitt, D., Naughton, J., "The 007 Benchmark", *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, Washington, D.C., May, 1993.

[Care94a] Carey, M., DeWitt, D., Franklin, M., Hall, N., McAuliffe, M., Naughton, J., Schuh, D., Solomon, M., Tan, C., Tsatalos, O., White, S., Zwilling, M.. "Shoring up Persistent Applications" *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, Minneapolis, MN, June 1994.

[Care94b] Carey, M., Franklin, M., Zaharioudakis, M., "Fine-grained Locking in Page Server Database Systems", *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, Minneapolis, MN, June, 1994.

[Catt91] Cattell, R., *Object Data Management*, Addison Wesley, Reading, MA, 1991.

[Catt92] Cattell, R., Skeen, J., "Object Operations Benchmark", *ACM Transactions on Database Systems*, 17,1, March, 1992.

[Chu94] Chu, S., Winslett, M., "Minipage Locking Support for Page-Server Database Management Systems", *Proc. 3rd Intl. Conference on Information and Knowledge Management*, Gaithersburg, MD, November, 1994.

[Dan90a] Dan, A., Dias, D., Yu, P., "The Effect of Skewed Data Access on Buffer Hits and Data Contention in a Data Sharing Environment", *Proceedings of the 16th International Conference on Very Large Data Bases*, Brisbane, Australia, August, 1990.

[Dan90b] Dan, A., Yu, P., "Performance Comparisons of Buffer Coherency Policies", *IBM Research Report RC16361*, November, 1990.

[Dan91] Dan, A., Dias, D., Yu, P., "Analytical Modelling of a Hierarchical Buffer for a Data Sharing Environment", *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modelling of Computer Systems*, San Diego, May, 1991.

[Dan92] Dan, A., Yu, P., "Performance Analysis of Coherency Control Policies through Lock Retention", *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, San Diego, June, 1992.

[Davi85] Davidson, S., Garcia-Molina, H., Skeen, D., "Consistency in Partitioned Networks", *ACM Computing Surveys*, 17(3), September, 1985.

[Deux91] O. Deux *et al.*, "The O2 System", *Communications of the ACM*, 34(10), October, 1991.

[DeWi90] DeWitt, D., Futtersack, P., Maier, D., Velez, F., "A Study of Three Alternative Workstation-Server Architectures for Object-Oriented Database Systems", *Proceedings of the 16th International Conference on Very Large Data Bases*, Brisbane, Australia, August, 1990.

[Dias87] Dias, D., Iyer, B., Robinson, J., Yu., P., "Design and Analysis of Integrated Concurrency-Controls", *Proceedings of the 13th International Conference on Very Large Data Bases*, Brighton, England, 1987.

[Exod93] EXODUS Project Group, "EXODUS Storage Manager Architectural Overview", *EXODUS Project Document*, Computer Sciences Department, University of Wisconsin-Madison, (available by ftp from ftp.cs.wisc.edu), 1993.

[Fran92a] Franklin, M., Carey, M.,"Client-Server Caching Revisited", in *Proceedings of the International Workshop on Distributed Object Management*, Edmonton, Canada, August, 1992, (Published as *Distributed Object Management*, Ozsu, Dayal, Vaduriez, eds., Morgan Kaufmann, San Mateo, CA, 1994).

[Fran92b] Franklin, M. Carey, M., and Livny, M., "Global Memory Management in Client-Server DBMS Architectures", *Proceedings of the 18th International Conference on Very Large Data Bases*, Vancouver, B.C., Canada, August, 1992.

[Fran92c] Franklin, M., Zwilling, M., Tan, C., Carey, M., DeWitt, D., "Crash Recovery in Client-Server EXODUS", *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, San Diego, June, 1992.

[Fran93a] Franklin, M., *Caching and Memory Mangement in Client-Server Databse Systems*, Ph.D. Thesis, Dept. of Computer Science, University of Wisconsin, July, 1993.

[Fran93b] Franklin, M. Carey, M., and Livny, M., "Local Disk Caching in Client-Server Database Systems", *Proceedings of the 19th International Conference on Very Large Data Bases*, Dublin, Ireland, August, 1993.

[Good83] Goodman, J., "Using Cache Memory to Reduce Processor Memory Traffic", *Proceedings of the 10th International Symposium on Computer Architecture*, Stockholm, Sweden, June, 1983.

[Gray93] Gray, J., Reuter, A., *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, San Mateo, CA, 1993.

[Henn90] Hennesy, J., Patterson, D., *Computer Architecture, A Quantitative Approach*, Morgan Kaufmann, San Mateo, CA., 1990.

[Howa88] Howard, J., Kazar, M., Menees, S., Nichols, D., Satyanarayanan, M., Sidebotham, R., West, M., "Scale and Performance in a Distributed File System", *ACM Transactions on Computer Systems*, 6(1), February, 1988.

[Josh91] Joshi, A., "Adaptive Locking Strategies in a Multi-Node Data Sharing System", *Proceedings of the 17th International Conference on Very Large Data Bases*, Barcelona, 1991.

[Kim90] Kim, W., Garza, J., Ballou, N., Woelk, D., "The Architecture of the ORION Next-Generation Database System," *IEEE Transactions on Knowledge and Data Engineering*, 2(1), March, 1990.

[Kist91] Kistler, J., Satyanarayanan, M., "Disconnected Operation in the Coda File System", *Proceedings of the 13th International Symposium on Operating System Principles*, Pacific Grove, CA, October 1991.

[Lamb91] Lamb, C., Landis, G., Orenstein, J., Weinreb, D., "The ObjectStore Database System", *Communications of the ACM*, 34(10), October, 1991.

[Levy90] Levy, E., Silbershatz, A., "Distributed File Systems: Concepts and Examples", *ACM Computing Surveys*, 22(4), December, 1990.

[Li89] Li, K., Hudak, P., "Memory Coherence in Shared Virtual Memory Systems", *ACM Transactions on Computer Systems*,7(4) November, 1989.

[Lisk92] Liskov, B., Day, M., Shrira, L., "Distributed Object Management in Thor", *Proceedings of the International Workshop on Distributed Object Management*, Edmonton, Canada, August, 1992, (Published as *Distributed Object Management*, Ozsu, Dayal, Vaduriez, eds., Morgan Kaufmann, San Mateo, CA, 1994).

[Livn90] Livny, M., *DeNet User's Guide*, Version 1.5, Computer Sciences Dept., University of Wisconsin-Madison, 1990.

[Lome94] Lomet, D., "Private Locking and Distributed Cache Management", *Proc. of the 3rd Intl. Conference on Parallel and Distributed Information Systems*, Austin, TX, September, 1994.

[Moha91] Mohan, C., Narang, I., "Recovery and Coherency Control Protocols for Fast Intersystem Page Transfer and Fine-Granularity Locking in a Shared Disks Transaction Environment", *Proceedings of the 17th International Conference on Very Large Data Bases*, Barcelona, Spain, September, 1991.

[Moha92] Mohan, C., Haderle, D. Lindsay, B., Pirahesh, H., Schwarz, P., "ARIES: A Transaction Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging", *ACM Transactions on Database Systems*, 17(1), March, 1992.

[Nels88] Nelson, M., Welch, B., Ousterhout, J., "Caching in the Sprite Network File System", *ACM Transactions on Computer Systems*, 6(1), February, 1988.

[Nitz91] Nitzberg, B., Lo, V., "Distributed Shared Memory: A Survey of Issues and Algorithms", *IEEE Computer*, 24(8), August, 1991.

[Obje91] Objectivity Inc., *Objectivity/DB Documentation Vol. 1*, 1991.

[Onto92] ONTOS Inc., *ONTOS DB 2.2 Reference Manual*, 1992.

[Rahm91] Rahm, E., "Concurrency and Coherency Control in Database Sharing Systems", *Technical Report 3/91*, Computer Science Dept., University of Kaiserslautern, Germany, November 1991.

[Rahm93] Rahm, E., "Emperical Performance Evaluation of Concurrency and Coherency Control Protocols for Database Sharing Systems", *ACM Transactions on Database Systems*, 18,2, June, 1993.

[Rama92] Ramakrishnan, K., Biswas, P., Karedla, R., "Analysis of File I/O Traces in Commercial Computing Environments", *Proceedings of the ACM SIGMETRICS and Performance '92 Conference*, May, 1992.

[Sand92] Sandhu, H., Zhou, S., "Cluster-Based File Replication in Large-Scale Distributed Systems", *Proceedings of the ACM SIGMETRICS and Performance '92 Conference*, May, 1992.

[Sten90] Stenstrom, P., "A Survey of Cache Coherence Schemes for Multiprocessors", *IEEE Computer*, 23(6), June, 1990.

[Ston79] Stonebraker, M., "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES", *IEEE Transactions on Software Engineering*, SE-5(3), May, 1979.

[Stur80] Sturgis, H., Mitchell, J., Israel, J., "Issues in the Design and use of a Distributed File System", *Operating Systems Review*, 14(3), July, 1980.

[Vers91] Versant Object Technology, *VERSANT System Reference Manual, Release 1.6*, Menlo Park, CA, 1991.

[Wang91] Wang, Y., Rowe, L., "Cache Consistency and Concurrency Control in a Client/Server DBMS Architecture", *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, Denver, June, 1991

[Wilk90] Wilkinson, W., Neimat, M., "Maintaining Consistency of Client Cached Data", *Proceedings of the 16th International Conference on Very Large Data Bases*, Brisbane, Australia, August, 1990.