Transactional Client-Server Cache Consistency: Alternatives and Performance

Michael J. Franklin
University of Maryland, College Park
and
Michael J. Carey
IBM Almaden Research Center
and
Miron Livny
University of Wisconsin-Madison

Client-server database systems based on a data shipping model can exploit client memory resources by caching copies of data items 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 transactional 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 present 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 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.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed Databases*; C.4 [**Computer Systems Organization**]: Performance of Systems; D.4.8 [**Operating Systems**]: Performance; H.2.4 [**Database Management**]: Systems—*Concurrency, Distributed Systems, Transaction Processing*

General Terms: Performance, Algorithms, Design

Additional Key Words and Phrases: Client-server databases, cache consistency, cache coherency, performance analysis, object-oriented databases

This work was supported in part by NSF grants IRI-9409575 and IRI-8657323, DARPA contract DAAB07-92-C-Q508, and by a research grant from IBM. Author's addresses: M. Franklin, Dept. of Computer Science, A. V. Williams Building, College Park, MD 20742; M. Carey, IBM Almaden Research Center, 650 Harry Road, K55/B1, San Jose, CA 95120; M. Livny, Computer Sciences Department, 1210 W. Dayton, UW-Madison, WI 53706.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works, requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept, ACM Inc., 1515 Broadway, New York, NY 10036 USA, fax +1 (212) 869-0481, or `permissions@acm.org`.

1. INTRODUCTION

1.1 Client-Server Database System Architectures

Advances in distributed computing and object-orientation have combined to bring about the development of a new class of database systems. These systems employ a client-server computing model to provide both responsiveness to users and support for complex, shared data in a distributed environment. Current relational DBMS products are based on a *query-shipping* approach in which most query processing is performed at servers; clients are primarily used to manage the user interface. In contrast, object-oriented database systems (OODBMS), which were initially developed to support computationally intensive applications such as computer aided design (CAD), typically support *data-shipping*, which allows data request processing (in addition to application processing) to be performed at the clients. With data-shipping, DBMS software running on the client machines determines which data items are needed to satisfy a given application request and obtains those items from the server if they can not be found locally.

The advantages of data-shipping for object-based DBMSs are two-fold: First, data-shipping moves the data closer to the applications, thus accelerating *navigation* through persistent data structures via the programmatic interfaces of object-based DBMSs. Second, data-shipping offloads much of the DBMS function from the server to the client workstations, providing both performance and scalability improvements. The performance advantages of data-shipping for navigation-oriented workloads are highlighted by benchmarks such as the 001 (or "Sun") engineering database benchmark [Cattell and Skeen 1992] and the more recent 007 benchmark [Carey et al. 1993]. As a result of these advantages, data-shipping is used in research prototypes such as ORION [Kim et al. 1990], Client-Server EXODUS [Franklin et al. 1992; Exodus Project Group 1993], SHORE [Carey et al. 1994], and THOR [Liskov et al. 1992], as well as in commercial products such as GemStone [Butterworth et al. 1991], O2 [O. Deux et al. 1991], ObjectStore [Lamb et al. 1991], Ontos [Ontos Inc. 1992], Objectivity [Objectivity Inc. 1991], and Versant [Versant Object Technology 1991].

While data-shipping can be beneficial in client-server object database systems, there is also a potential downside; data-shipping implementations are susceptible to network and/or server bottlenecks that can arise if a high volume of data is requested by clients. The key to avoiding these bottlenecks is to use local client storage resources for *data caching*. Client data caching enables clients to retain copies of data items that they have received from servers. In the presence of locality (i.e., the affinity of applications at certain workstations for certain subsets of the data items), such caching can significantly reduce the volume of data that clients must request from servers.

1.2 Transactional Cache Consistency

Client data caching is a dynamic form of data replication. As with any form of data replication, care must be taken to ensure that the presence of multiple copies in the distributed system does not jeopardize the correctness of programs. In a database system, correctness in the presence of concurrency, distribution, replication, and failures is tied to the concept of transactions. It is important, therefore, to dis-

tinguish between two types of caching: (1) *intra-transaction caching*, which refers to the caching of data within transaction boundaries; and (2) *inter-transaction caching*, which allows clients to retain locally cached data even across transaction boundaries. Intra-transaction caching can be implemented by purging any cached items that are not protected by an active transaction and relying on the normal concurrency control mechanism (e.g., two-phase locking, etc.) to ensure the validity the remaining cached data. In contrast, inter-transaction data caching allows data items to remain in client caches even outside of transaction boundaries. Such cached items are not protected by the regular (transaction-oriented) concurrency control mechanism, so an additional *cache consistency protocol* is required to ensure correctness.

Cache consistency protocols for client-server database systems have been the subject of much study in recent years and at least a dozen different algorithms have been proposed and studied in the literature [Wilkinson and Neimat 1990; Carey et al. 1991; Wang and Rowe 91; Franklin and Carey 1992; Adya et al. 1995]. In terms of semantics, *all of these proposed algorithms support the traditional notion of ACID transaction isolation*; that is, they all ensure that transactions always see a serializable, or "degree three" [Gray and Reuter 1993] view of the database.[1] The proposed algorithms, however, differ greatly in the details of their implementation; the developers of these algorithms have made widely varying choices along numerous design dimensions. Thus, despite the fact that all of these algorithms provide an identical level of protection, the performance of the various algorithms can be expected to differ significantly.

While the papers mentioned previously have all included performance comparisons of several of the proposed algorithms, none of these studies have attempted to unify and explain the large design space for transactional 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. In this paper we address this problem by proposing a taxonomy of transactional cache consistency maintenance algorithms that encompasses all of the algorithms that have been examined in the previously mentioned studies. The taxonomy outlines the numerous dimensions of the design space for transactional cache consistency algorithms and shows how proposed algorithms relate to one another. After describing this taxonomy, we then use it to drive an analysis of performance results for three families of algorithms across a range of workloads.

1.3 Relationship to Other Caching Systems

Caching has been used to reduce latency and improve scalability in many different computing environments, including multiprocessors [Archibald and Baer 1986; Stenstrom 1990; Lilja 1993; Adve and Gharachorloo 1995], distributed shared memory (DSM) systems [Li and Hudak 1989; Nitzberg and Lo 1991; Keleher et al. 1992], and distributed file systems [Howard et al. 1988; Nelson et al. 1988; Levy and Silberschatz 1990]. The maintenance of cache consistency has been addressed to varying

[1] Because all of these algorithms are intended to support navigation-oriented data access, they do not explicitly address the "phantom" problem [Gray and Reuter 1993] that arises in associative, predicate-based access.

degrees in all of these environments. While many of the underlying techniques used to enforce cache consistency are common across all of these environments, there are two aspects of modern object-oriented database systems that distinguish solutions in this environment from the others: (1) transactions, and (2) the client-server architecture.

The *transactional* nature of caching in client-server database systems both imposes constraints and provides additional implementation opportunities for cache consistency maintenance protocols. The constraints include the requirement to provide the ACID semantics, which combine correctness criteria for concurrent execution of *arbitrary* programs by multiple users with guarantees for fault tolerance. Opportunities arise from the well-defined points of synchronization provided by the transaction model and the ability to piggyback cache consistency information on the protocols used to support concurrency control and recovery. The client-server OODBMS architecture impacts the design of cache consistency maintenance protocols by enforcing a specific partitioning of responsibilities between clients and servers. The relationship between transactional caching in client-server object database systems and data caching in other environments is addressed in further detail in Section 3.

1.4 Scope of the Paper

This paper makes several contributions. One main contribution is the presentation of a taxonomy that provides a unified treatment of proposed cache-consistency algorithms for client-server object database systems. A key insight that arises from the taxonomy is the usefulness of classifying algorithms based on whether they *detect* or *avoid* access to stale cached data. This distinction is in contrast to the possibly more intuitive one between optimistic and pessimistic approaches, which does not accurately characterize the complex algorithms that have been proposed. While the taxonomy is complete in the sense that it encompasses the major algorithms that have been studied in the literature, it does not, of course, cover all *conceivable* algorithms. The body of work covered by this taxonomy consists of proposed algorithms that provide serializable transaction execution; the taxonomy does not address the many possible relaxations of these semantics. Instead, because all of the algorithms provide exactly the same semantics to client-based transactions, the emphasis of the taxonomy is on issues that impact *performance*.

After describing the taxonomy and placing the proposed algorithms in it, we then use the taxonomy to guide the analysis of the performance results that we have obtained in detailed simulation studies of a subset of the algorithms. This study includes the description of a suite of synthetic workloads that have been used to explore the performance of caching algorithms in a number of scenarios. Finally, the insight gained by analyzing the performance results in the context of the taxonomy is used to reflect upon the characteristics of the other proposed algorithms that have appeared in the literature. In this way, the taxonomy is used to shed light both on the nature of the design space for transactional cache consistency algorithms and on the performance tradeoffs implied by many of the choices that exist in the design space.

The remainder of the paper is organized as follows: Section 2 outlines a reference architecture for a data-shipping database system and describes the constraints on

the algorithm design space that are implied by the architecture. Section 3 discusses cache consistency issues that have been addressed in other computing environments. Section 4 presents our taxonomy of transactional cache consistency maintenance algorithms. Section 5 describes three families of transactional cache consistency maintenance algorithms in greater detail and examines their performance using a detailed simulation model. Section 6 comments on the tradeoffs made by the remaining algorithms in the taxonomy. Section 7 presents our conclusions.

2. CACHING IN CLIENT-SERVER DATABASE SYSTEMS

2.1 Reference Architecture

Figure 1 shows a 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.[2] 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. For simplicity, we assume that the database is statically partitioned across the servers. In general, data can be replicated among the servers in order to improve availability and/or performance; such replication, however, is beyond the scope of this paper.

In a data-shipping architecture, each Client DBMS process is responsible for translating local application requests into requests for specific database items and for bringing those items into memory at the client. As a result, all of the data items referenced by an application are ultimately brought from the server(s) to the client. Some or all of these items 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 items that it owns in response to client requests; of course, due to concurrency control conflicts, it may not be possible for the server to provide each requested item immediately.

Data-shipping systems can be structured either as *page servers*, in which clients and servers interact using fixed-length, physical units of data such as pages (typically on the order of four or eight Kbytes) or *object servers*, which interact using logical, possibly variable-length, units of data such as tuples or objects. Aspects of the tradeoffs between page servers and object servers have been studied for the

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

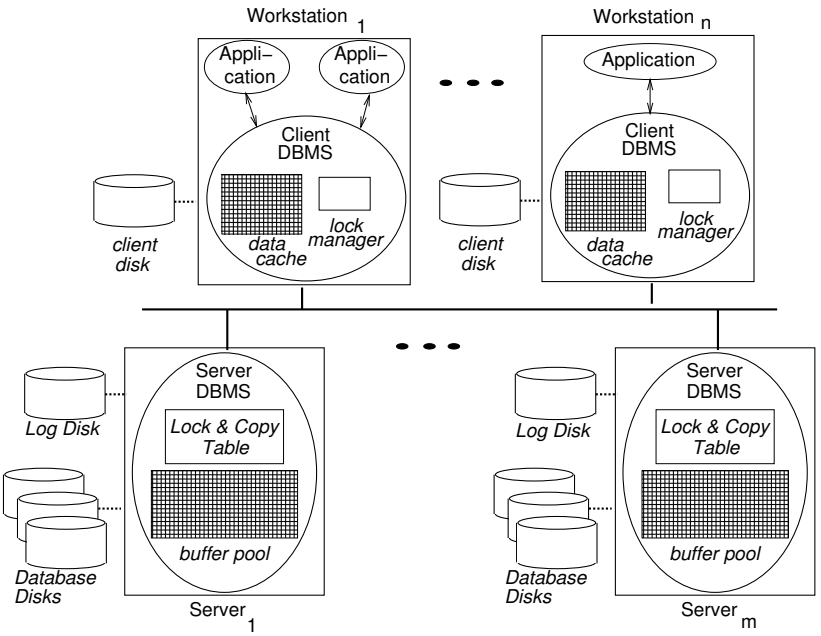

Fig. 1. Reference Architecture for a Data-Shipping DBMS

single-user case in [DeWitt et al. 1990; Kemper and Kossmann 1994], and for the multi-user case in [Carey et al. 1994; Chu and Winslett 1994; Adya et al. 1995]. For concreteness, this paper focuses on page-server architectures in which data transfer, concurrency control, and cache consistency are all performed at the granularity of a single page. The algorithms discussed here, however, are equally applicable to both page servers and object servers in that they assume that the same granularity (i.e., page or object) is used for concurrency control and cache consistency. The issues that arise when mixing levels of granularity for any of these functions are addressed in [Carey et al. 1994; Chu and Winslett 1994].

2.2 Architectural Implications for Caching

An important aspect of any data-shipping system is that all of the DBMS processes (both Client and Server) have storage resources (memory and/or disk) that they use for buffering data. 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. In fact, in a typical environment with fairly high-powered client machines (e.g., "fat" clients such as state-of-the-art PCs or workstations) the majority of the *aggregate* processing and storage resources available to the DBMS are likely to reside at the clients. As in other computing environments, client caching is a key technique for exploiting these resources.

The design of a client caching mechanism for a data-shipping database system must respect the correctness and availability constraints of that environment. Workstation-server database systems must be capable of providing the same level

of transaction support as more traditional database architectures, including serializability. Because client caching is essentially a form of 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* [Bernstein et al. 1987]. 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, a client caching mechanism must be designed in a way that the crash or disconnection of an individual client workstation does not impact the availability of data for applications running at other clients. Reducing the impact of client failures is crucial for several reasons. First, if individual client failures are allowed to inhibit availability, then the scalability of the system may be limited. Second, it is more cost-effective to fortify the (fewer) server machines (e.g., by adding duplexed log disk storage or non-volatile memory) than to bullet-proof all of the client machines. Finally, since client machines are typically located on users' desktops or in their homes or briefcases, they are typically less closely supported by an operations staff than servers. The combination of these correctness and availability concerns leads to the identification of two key properties of client data caching in a database environment: *dynamic replication* and *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, in which replication is determined as part of physical database design (e.g., [Stonebraker 1979]).

Second-class ownership refers to the fact that the cached copies of pages at clients are not considered to be the equals of the actual data pages, which are kept at the server. One problem with replication is that it can reduce data availability for updates in the presence of failures (e.g., network partition) in a distributed environment [Davidson et al. 1985]. Second-class ownership allows consistency to be preserved without sacrificing availability.[3] 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.

Client caching as defined above provides the ability to exploit client resources for performance and scalability improvements. The protocol used to maintain cache

[3]The term "second-class ownership" is derived from a similar concept called "second-class replication" used in the CODA distributed file system [Kistler and Satyanarayanan 1991]. 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, however, imposes its own costs on the database system. Depending on the protocol used, these costs can include communication with the server, additional transaction aborts, and reduced efficiency for client cache usage. Given that a requirement for any such protocol is that it support ACID transactions and respect the architectural constraints described above, the challenge is to design a protocol that incurs minimal overhead across a range of workloads. The remainder of this paper investigates the design space of the transactional cache consistency maintenance algorithms for client-server database systems that have been proposed in the literature, and analyzes the performance of a number of these algorithms.

3. RELATED WORK

As discussed in Section 1.3, cache consistency issues arise in many types of distributed and/or parallel systems including non-database systems such as multiprocessors, distributed shared memory systems, and distributed file systems, as well as other database architectures, such as shared-disk database systems. While there are many similarities between the basic consistency maintenance techniques available in client-server database systems and in these other environments, there are significant differences in the systems that impact the design alternatives and their inherent tradeoffs. In the following sections we first describe how transactional cache consistency differs from cache consistency in non-database environments, and then describe how client-server database systems differ from shared-disk database systems. Important areas of difference for each of the systems include some or all of the following: (1) correctness criteria, (2) granularity of caching, (3) inherent cost tradeoffs, and (4) workload characteristics.

3.1 Non-Database Environments

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 [Archibald and Baer 1986]; more recent surveys appear in [Stenstrom 1990; Lilja 1993]. The traditional notion of correctness in multiprocessor systems is *sequential consistency* [Lamport 1979], which aims to ensure that program execution on a multi-processor machine with distributed memory (i.e., processor caches) provides the same results as an interleaved multi-threaded execution on a uniprocessor machine with a single memory. Sequential consistency, therefore, deals with the ordering of individual memory accesses rather than compound units such as transactions.

More recent work in the multiprocessor area has focused on developing alternative models that provide improved performance at the expense of requiring programmers (or compilers) to correctly place explicit synchronization primitives in their code [Adve and Gharachorloo 1995]. Many of these models, such as *release consistency* [Gharachorloo et al. 1990], allow programmers to combine multiple memory accesses into units that are protected by synchronization. For correctness, however, the composition of these units must be known and agreed upon *a priori* by all concurrent processes that may possibly access affected memory locations. Such *a priori* knowledge is a reasonable assumption in this environment, as the goal is typically to provide concurrent execution of a single, multi-threaded program. Sharing of data across separate programs is not directly supported by these protocols. Also,

all of these models have as their goal the provision of semantics that approach those of (volatile) memory in a uniprocessor, so they do not include support for fault tolerance.

Database systems have a different set of requirements than those of the multiprocessor memory algorithms. ACID semantics provide correctness for concurrent execution of transactions containing *arbitrary* groups of operations and ensure correct execution, even in the presence of failures. This is important because database systems must support correct access to a shared database in the presence of a constantly changing workload. Database workloads are typically imposed by many different users who concurrently present a mix of *ad hoc* queries and updates to the system. As a result of these differences in focus, the basic techniques that have been developed for database systems, such as two-phase locking or optimistic concurrency control, do not exist in the multiprocessor domain. As will be seen in Sections 4 and 5, the cache consistency algorithms that have been proposed for client-server database systems are based closely on these fundamental techniques.

Nevertheless, despite the differences in emphasis, there are basic common issues that must be addressed by all cache consistency maintenance algorithms. For example, stale cached copies can be dealt with in two ways: by invalidating them or by propagating new values to the affected caches (i.e., write-invalidate and write-broadcast [Archibald and Baer 1986]). Also, consistency actions can be distributed through the use of a broadcast medium (e.g., snooping caches [Goodman 1983]) or can be maintained in directories (e.g., [Agarawal et al. 1988]). Even at this level, however, the substantial architectural differences (e.g., message cost, cache consistency granularity, data access granularity, peer-to-peer vs. client-server) between data-shipping database systems and shared-memory multiprocessors limit the extent to which results from one area can be applied in the other.

Distributed Shared Memory (DSM) systems [Li and Hudak 1989; Nitzberg and Lo 1991] have cost tradeoffs that are closer to those in a data-shipping database environment. DSM systems provide the abstraction of a shared *virtual* memory address space that spans the nodes of a distributed system. Unlike multiprocessor caching, which can rely heavily on hardware support, DSMs are typically implemented in software with only minimal hardware assists. Because pages (or at least, relatively large cache lines) are the unit of consistency, DSMs are similar to client-server databases 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, messages may be less expensive). The main differences, therefore, lie in the area of correctness criteria for DSM systems, which are typically the same as those for multiprocessors.

Because of the high cost of messages in a distributed environment, recent improvements to DSM cache consistency protocols such as *Lazy Release Consistency* [Keleher et al. 1992] exploit communication patterns among processors to reduce the message overhead. Many of the protocols for data-shipping database systems also try to exploit existing communication patterns. In the database environment, however, these patterns are determined to a large extent by the protocols used between clients and servers to support ACID transactions (e.g, two-phase locking, write-ahead-logging, and optimistic concurrency control).

Client caching has been used in distributed file systems since some of the earliest work in the area (e.g., DFS [Sturgis et al. 1980]). Many distributed file systems that support some form of client caching have been proposed and built. A survey of distributed file systems can be found in [Levy and Silberschatz 1990]. As with data-shipping 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., [Baker et al. 1991]). Caching is often done at a fairly coarse granularity, such as entire files or large portions of files. Even so, the algorithms used in distributed file systems (e.g., Andrew [Howard et al. 1988] and Sprite [Nelson et al. 1988]) have served as the inspiration for at least one important class of cache consistency algorithms for client-server database systems (i.e., callback locking).

3.2 Shared-Disk Database Systems

In addition to work outside of the database area, cache-consistency issues have also been addressed in multiprocessor database architectures other than client-server 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 [Bhide and Stonebraker 1988]. 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, there is likely to be less locality of access at the individual nodes. 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 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 [Dias et al. 1987]. Later work has addressed the impact of data skew and contention for a range of possible algorithms [Dan et al. 1990; Dan and Yu 1991], the interaction between private and shared buffering [Dan et al. 1991] (similar to the interactions between client buffers and the server buffer), and extensions to callback-style algorithms [Dan and Yu 1992]. Other related work in this area includes the work of Mohan and Narang [Mohan and Narang 1991], Rahm [Rahm 1993], and Lomet [Lomet 1994]. An algorithm that dynamically adjusts the granularity at which locking and coherency are managed for a shared-disk DBMS was introduced in [Joshi 1991]. This approach was later extended for use in hybrid page server environments in [Carey et al. 1994].

4. A TAXONOMY OF ALGORITHMS

In this section we provide a taxonomy of transactional cache consistency algorithms that encompasses the major algorithms that have appeared in the literature, including [Wilkinson and Neimat 1990; Carey et al. 1991; Wang and Rowe 91; Franklin

and Carey 1992; Adya et al. 1995]. Recall that all of these algorithms provide strict one-copy serializability and are applicable to page server DBMSs (although some were originally proposed for object servers). The taxonomy is shown in Figures 2 and 3. 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. A key aspect of this taxonomy, is the choice of *Invalid Access Prevention* as the main criterion for differentiating algorithms. 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 are allowed to commit only if it can be verified that they 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 [Rahm 1991]. 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 possible approach would be 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.

4.1 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. A data item is considered to be stale if its value is older than the item's latest *committed* value.[4] 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. Some avoidance-based

[4]Some concurrency control techniques based on multiple versions allow read-only transactions to access stale data provided that they see a consistent snapshot of the database [Bernstein et al. 1987]. While such algorithms are beyond the scope of this paper, several of the consistency algorithms do allow multiple active transactions to simultaneously access different values of the same page, provided that strict (i.e., commit order) serializability is not violated.

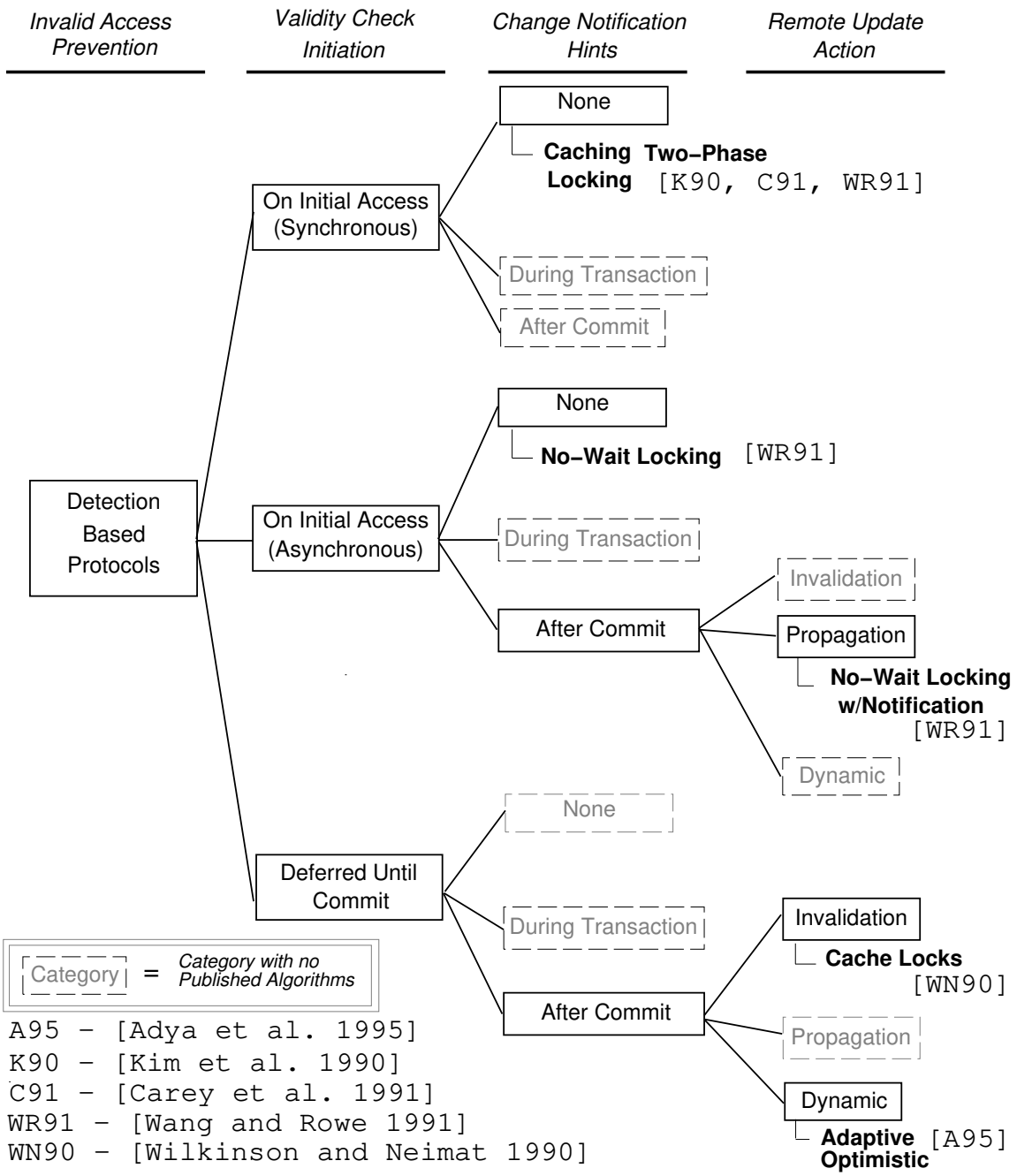

Fig. 2. Taxonomy of Detection-Based Algorithms

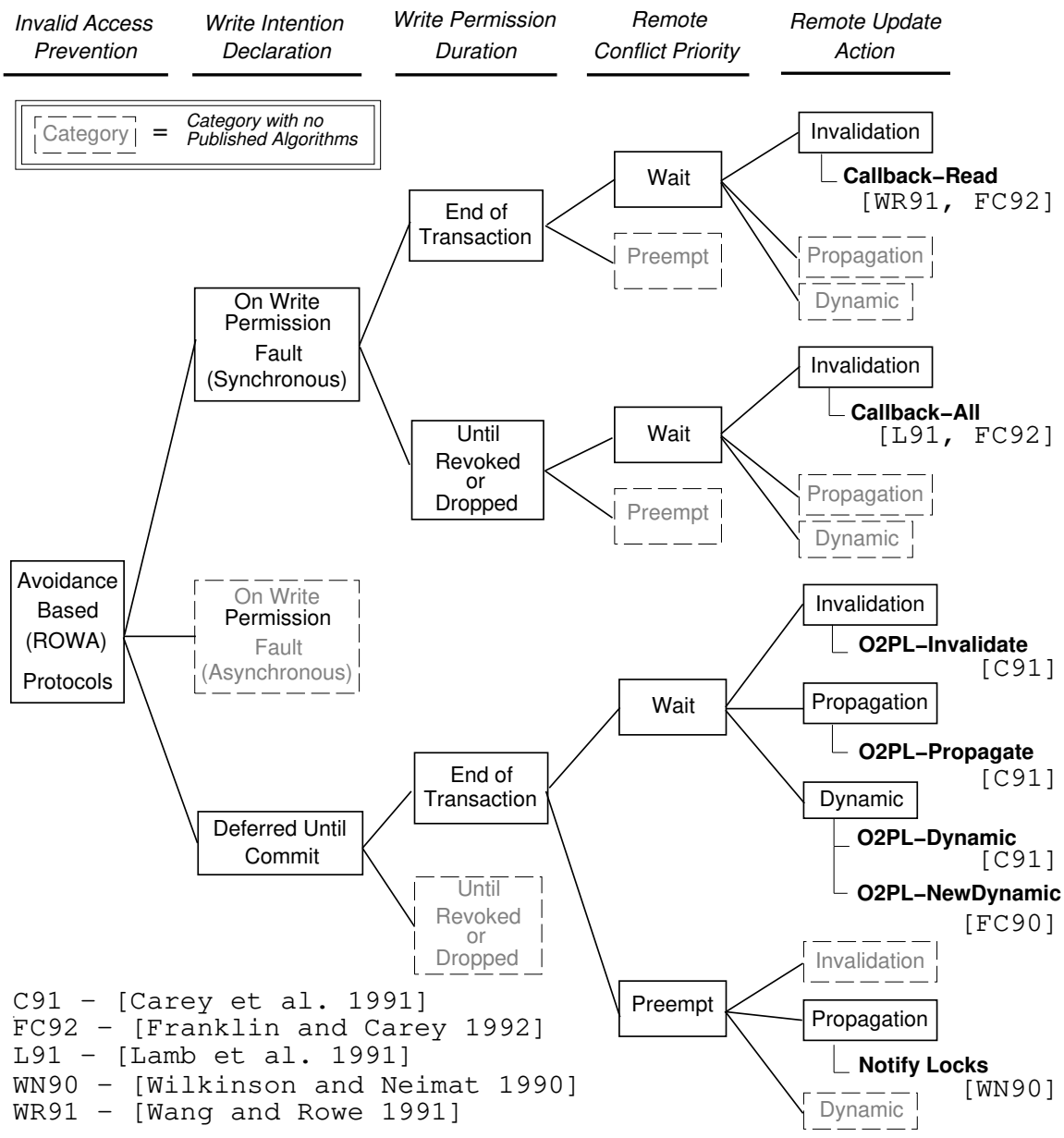

Fig. 3. Taxonomy of Avoidance-Based Algorithms

schemes also retain certain access permissions across transaction boundaries in order to protect cached data that is not accessed by an active transaction.

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.[5]

Before proceeding, it should be noted that detection-based algorithms can be augmented with techniques similar to those employed to enforce consistency in avoidance-based algorithms. 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. A key point, however, is that these three algorithms lie strictly in the detection-based camp, as the notifications are sent only as "hints". That is, despite the use of hints, these algorithms still allow stale cache contents to be present and potentially accessed by transactions and thus, must ultimately depend on detection to ensure that committing transactions have not accessed any stale data. In contrast, the use of avoidance obviates any possible need for detection, so there is no augmentation in the opposite direction within the taxonomy.

4.2 Detection-based Algorithms

A number of detection-based algorithms (shown in Figure 2) have been proposed and studied in the literature. An advantage of 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 [Exodus Project Group 1993] chose a detection-based approach largely for this reason. 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 notification hints, and remote update action.

[5] As described in Section 2.2, the use of second-class replication allows the server to unilaterally eliminate any unreachable copies from the protocol so that transaction processing can continue.

4.2.1 *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.[6]

Synchronous validity checking is the simplest of the three classes. On the first access that a transaction makes to a cached data item, 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.

[6] 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.

The main disadvantage is that deferral can result in the late detection of data conflicts, which will cause the abort of one or more transactions. The asynchronous approach is 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.

4.2.2 *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 use optimism 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 [Cattell 1991]. 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 [Ramakrishnan et al. 1992; Sandhu and Zhou 1992] indicate that sequential sharing may, in fact, be quite common in the types of situations in which page servers are intended to be used.

Two approaches to reducing the potential for aborts in optimistic techniques have been proposed. One is to treat "hot spot" data differently, e.g., by switching to a more pessimistic protocol for such data (e.g., [Adya et al. 1995]). The other is to use techniques from the avoidance-based (ROWA) algorithms 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 (or an impending update) that may impact the validity of an item cached at that client. Purging or updating a stale copy removes the risk that a subsequent transaction will be forced to abort as a result of accessing it.

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.

4.2.3 *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.

4.3 Avoidance-based Algorithms

Avoidance-based algorithms form the other half of our taxonomy. 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).

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 [Goodman 1983]. 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 [Agarawal et al. 1988] must be used. As discussed in Section 2, 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.

4.3.1 *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[7]. The duration of write permissions is addressed in Section 4.3.2.

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 or asynchronously. 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 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 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. Furthermore, in avoidance-based algorithms the remote consistency operations are in the critical path of transaction commit; an update transaction cannot commit until all of the necessary consistency operations have been successfully completed at remote clients. These considerations tend to strengthen the case for deferral of consistency actions for avoidance-based algorithms. Of course, the cost of such deferral is a potential increase in the number of aborted transactions.

[7] 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 (i.e., they can be released and reacquired).

4.3.2 *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 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 or until the server asks a client to drop its write permission (in conjunction with the performance of a consistency action).

4.3.3 *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.

4.3.4 *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, all 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 that provide the basis for the correctness of avoidance-based algorithms. 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.

5. A PERFORMANCE COMPARISON OF THREE ALGORITHM FAMILIES

The taxonomy presented in the previous section illuminates the wide range of options that have been explored by designers of transactional cache consistency maintenance algorithms. Recall that because the algorithms that have been proposed

all provide the same functionality (i.e., they support one-copy serializability in the presence of dynamic caching), 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.

Our own work has focused primarily on algorithms from the avoidance-based half of the taxonomy [Carey et al. 1991; Franklin and Carey 1992]. 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 6 to reflect on the performance characteristics of the remaining design choices and algorithms in the taxonomy.

5.1 Algorithms

The algorithms that we have focused on come from three families: Server-based two-phase locking (S2PL), Callback Locking (CBL), and Optimistic 2PL (O2PL). The study of these three families was initially undertaken to develop alternatives for the SHORE object manager. We focused on these three families for the following reasons: S2PL algorithms have been used in a number of early systems including ORION and EXODUS. CBL is a logical extension of two-phase locking and has been used in the ObjectStore OODBMS; a variant of CBL was shown to have good performance in [Wang and Rowe 91]. Finally, the O2PL family is based on an algorithm that performed well in an earlier study of distributed database systems [Carey and Livny 1991]. While the algorithms in these three families 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 [Franklin 1993] 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 4.

5.1.1 *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 [Alsberg and Day 1976; Stonebraker 1979]. 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" pol-

icy. All page copies are tagged with a version number that uniquely identifies the state of the page.[8] 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 [Kim et al. 1990] and the EXODUS storage manager [Exodus Project Group 1993]. An algorithm similar to C2PL has also been studied in [Wang and Rowe 91].

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 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.

5.1.2 *Callback Locking (CBL).* 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.

[8] 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 [Gray and Reuter 1993].

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 acknowledgment 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 a dirty page back to the server until the write

permission on the page is downgraded or the page is dropped [Franklin et al. 1993].

Callback-style algorithms originated in the operating systems community for maintaining cache consistency in distributed file systems such as Andrew [Howard et al. 1988] and Sprite [Nelson et al. 1988], 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 [Lamb et al. 1991]. An algorithm similar to CB-R was also studied in [Wang and Rowe 91].

5.1.3 *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 [Carey and Livny 1991]. 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.[9] 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 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 acknowledgment message to the server, a two-phase commit protocol is not necessary in this case. In contrast, under O2PL-P, remote clients must enter a two-phase commit protocol with the server in order to sure that the updates to all remote copies happen atomically. First, each client sends a message to the server indicating that it has obtained the necessary local locks. This message

[9] 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.

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.[10]

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.[11] As in the previously described algorithms, deadlocks are resolved by aborting the youngest transaction involved.

5.1.4 *Evaluating the Tradeoffs.* The three families of cache consistency maintenance algorithms described in the preceding sections cover a number of the design alternatives identified in the taxonomy presented in Section 4. As stated previously, the focus of our work 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 6. Table 1 summarizes the portion of the design space covered in this study and shows which algorithms can be compared in order to examine the performance tradeoffs implied by a given decision point in the taxonomy.

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 4, 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

[10] 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.

[11] 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 [Carey and Livny 1991].

prior to transaction commit, while *Avoidance* ensures that 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 the CBL algorithms can provide the clearest insights into this tradeoff. These algorithms all allow inter-transaction caching, do not propagate updated pages, and initiate their consistency actions synchronously. Of the two CBL algorithms, CB-A provides the strongest contrast with C2PL because it retains both read and write permissions. Comparing C2PL with CB-R is also useful because CB-R is avoidance-based but requires obtaining write permissions from the server in the same manner as C2PL.

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 4.3.1, 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 4.3.1, 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 4.3.4, 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.

5.2 A Client-Server Performance Model

5.2.1 *The System Model.* Figure 4 shows the structure of our simulation model, which was constructed using the DeNet discrete event simulation language [Livny 1990]. 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.

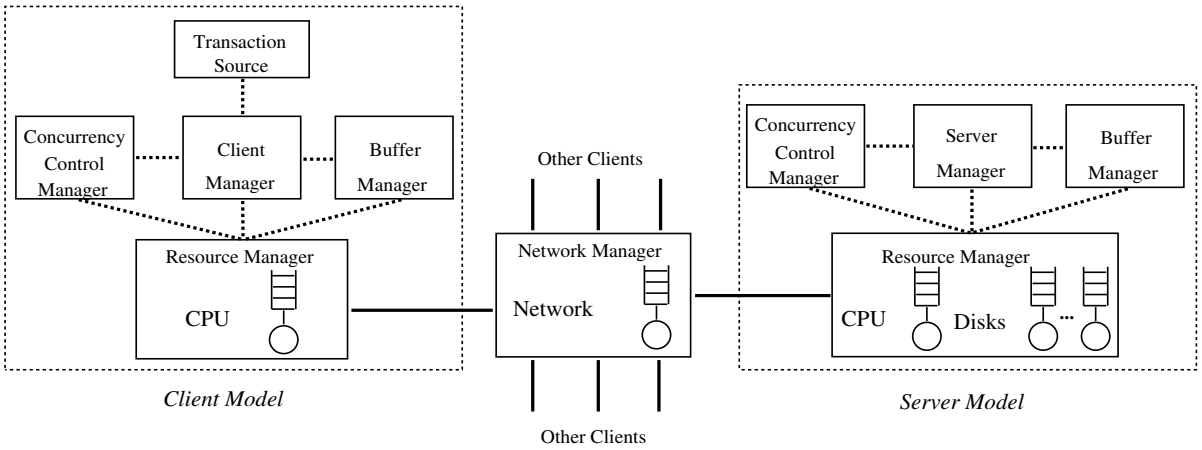

Fig. 4. Performance Model of a Client-Server DBMS

Table 2 describes the parameters that are used to specify the system resources and overheads and shows the settings used in this study. We used a relatively small database in order to make simulations involving fractionally large buffer pools feasible in terms of simulation time. The most important factor here is the ratio of the transaction and client-server buffer pool sizes to the database size, not the absolute database size itself.[12] 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 queuing 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

[12] Results demonstrating the scalability of the simulator when the database size and buffer sizes are increased by an order of magnitude and the transaction length is increased accordingly are described in [Carey et al. 1994].

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.

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

5.2.2 *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: hot and cold. The probability of a page access being to a hot range page 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, where each user has a portion of the design on which they work while accessing additional

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

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 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.

5.3 Experiments and Results

In this section we present results from performance experiments involving the algorithms described in Section 5.1. The main performance metric presented is system throughput (measured in transactions per second).[13] 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.

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

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 [Carey et al. 1991; Franklin and Carey 1992; Franklin 1993].

5.3.1 *The PRIVATE Workload.* We first examine performance results for the PRIVATE workload. As described in Section 5.2.2, 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 because it is unable to exploit client memory for storing data pages across transaction boundaries.

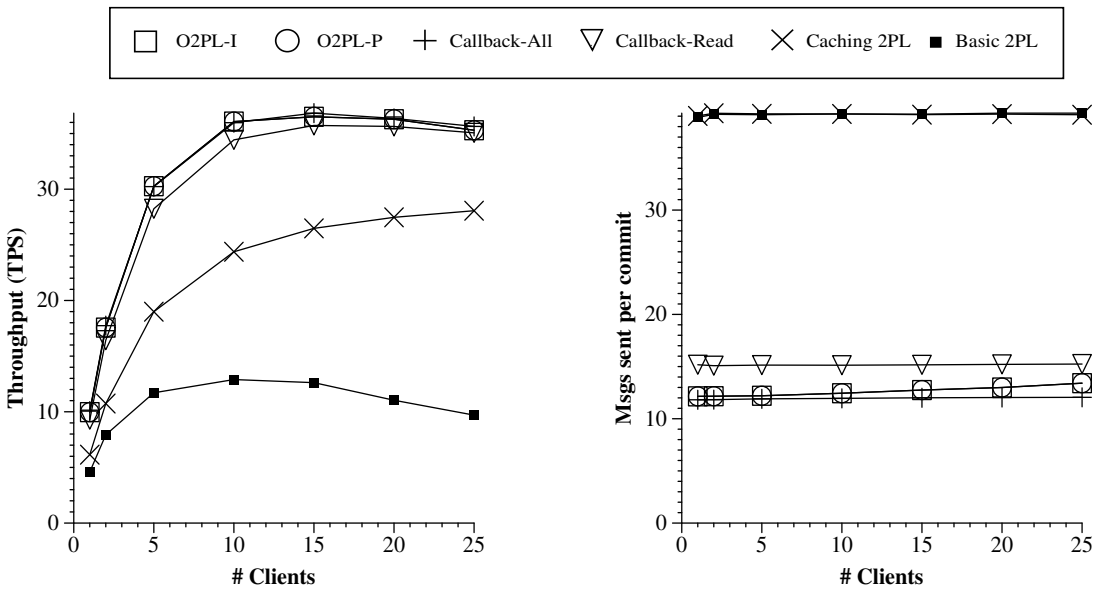

Fig. 5. Throughput
(PRIVATE, 25% Client Cache, Slow Net)

Fig. 6. Messages Sent/Commit
(PRIVATE, 25% Client Cache, Slow Net)

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 5.1.4. 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 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.

5.3.2 *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 5.2.2, 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.

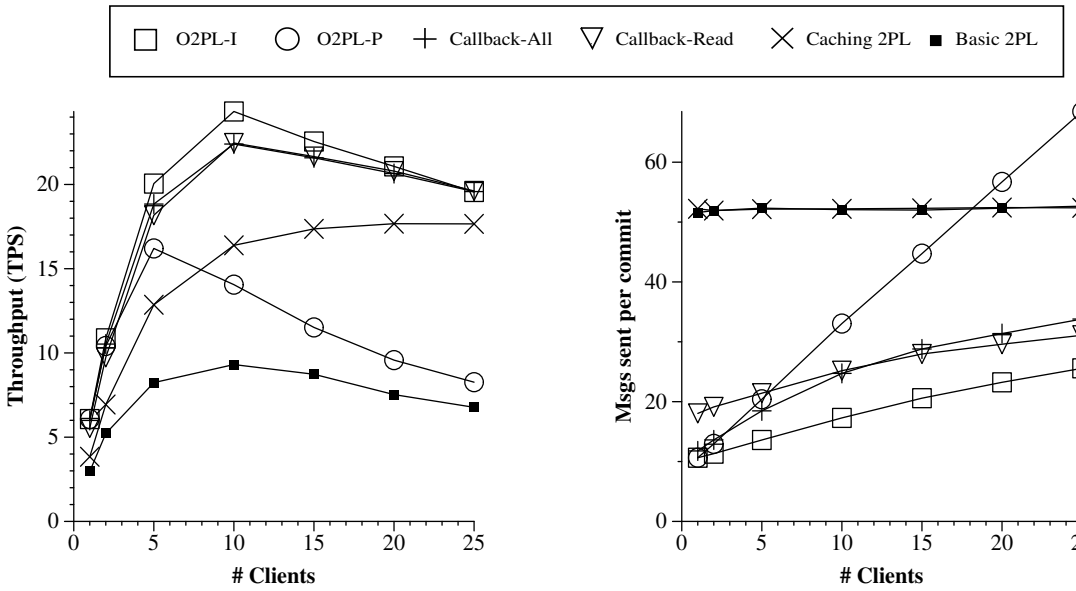

Fig. 7. Throughput
(HOTCOLD, 25% Client Cache, Slow Net)

Fig. 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, which in turns sends at most one request for Remote Update Actions to each remote client. This difference has only a small impact on throughput here, and that impact 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 5.1.4), 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 can more than nullify the performance benefits of caching in some cases.

5.3.3 *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 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.[14] 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 the re-execution of 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

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

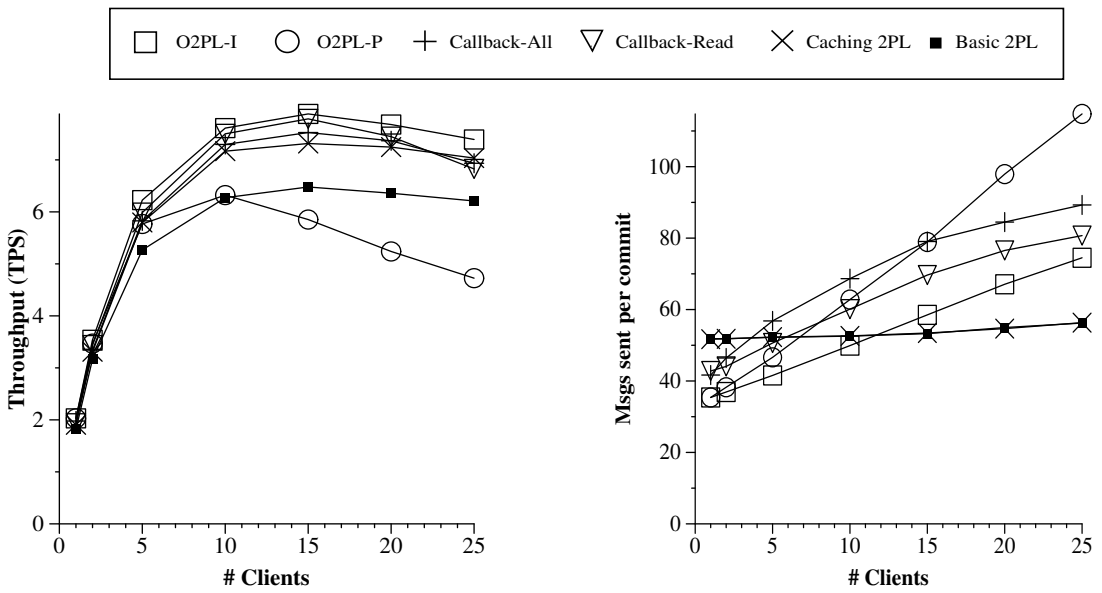

Fig. 9. Throughput
(UNIFORM, 25% Client Cache, Slow Net)

Fig. 10. Messages Sent/Commit
(UNIFORM, 25% Client Cache, Slow Net)

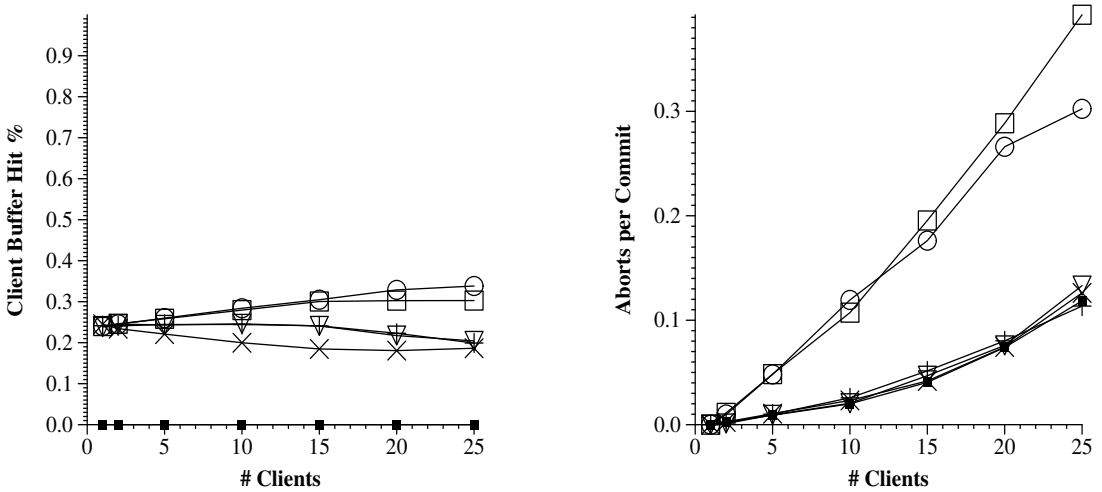

Fig. 11. Client Hit Rate
(UNIFORM, 25% Client Cache, Slow Net)

Fig. 12. Aborts/Commit
(UNIFORM, 25% Client Cache, Slow Net)

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 [Franklin and Carey 1992; Franklin 1993], the high abort rate of O2PL-I can cause it to have significantly worse performance than CB-R if data contention is further increased.

In terms of the Write Permission Duration, the high degree of sharing 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 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 HOTCOLD 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.

5.3.4 *The FEED Workload.* The last workload to be examined here is the FEED workload. As discussed in Section 5.2.2, 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.

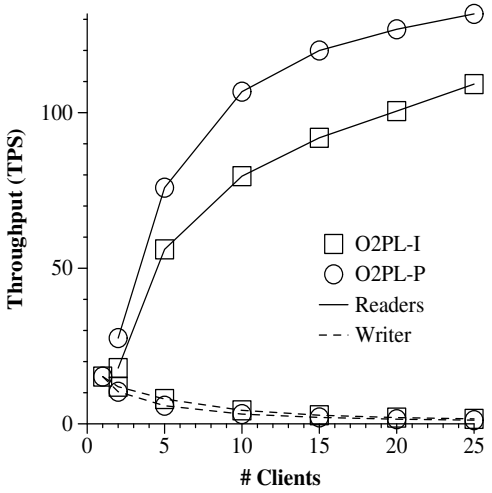

Fig. 13. Throughput
(FEED, 25% Client Cache, Slow Net)

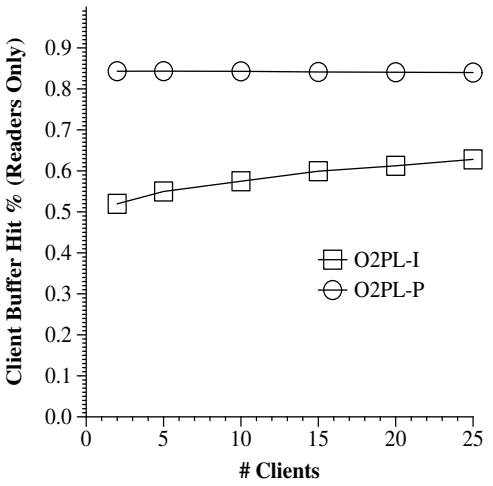

Fig. 14. Client Cache Hit % (Readers only)
(FEED, 25% Client Cache, Slow Net)

5.4 Summarizing The Results

In the preceding sections, we compared the performance of six different cache consistency maintenance algorithms. 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 pages prior to accessing them, 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. CB-A, however, 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. These results show that if detection is to be used, it must be done in an *optimistic* manner. Optimistic detection-based algorithms are discussed in Section 6.1. 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. Detection-based approaches can be extended with notification "hints" to help reduce those costs.

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. If,

however, sharing increases to the point where data contention arises, deferring declarations can lead to significantly higher abort rates; transaction aborts can result in higher resource utilization due to lost work [Franklin and Carey 1992; Franklin 1993], though this effect did not significantly hurt throughput in the workloads examined here. Furthermore, 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. Based on these results, we have developed and investigated *dynamic* algorithms that can adaptively choose between invalidation and propagation on a page-by-page basis at each client. These algorithms are discussed briefly in Section 6.4.

6. OTHER PROPOSED ALGORITHMS

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 4. 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 the other published algorithms that appear in the taxonomy.

6.1 Optimistic Detection-Based Algorithms

The first published paper to analyze transactional cache consistency algorithms for client-server OODBMSs was [Wilkinson and Neimat 1990]. 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 [Adya et al. 1995]. 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 [Wilkinson and Neimat 1990] 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 [Adya et al. 1995], 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 acknowledgment 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 [Adya et al. 1995], 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 5.3 may be incurred by this approach as well.

6.2 Notify Locks

The second algorithm proposed in [Wilkinson and Neimat 1990], *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 [Wilkinson and Neimat 1990] because that study used a probabilistic cache model that assumed that cache hit probabilities were independent of cache size.

6.3 No-wait Locking

No-wait locking algorithms were studied in [Wang and Rowe 91]. 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 5.1.4, 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 [Wang and Rowe 91] 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 [Wang and Rowe 91] 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 [Carey et al. 1991]) 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 [Wang and Rowe 91].

6.4 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 [Carey et al. 1991]. 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 [Franklin and Carey 1992; Franklin 1993]. 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 one case where that algorithm provided the best performance).

7. 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. We presented a taxonomy that describes the design space for such algorithms 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. As a result, most detection-based schemes that have been proposed employ optimistic techniques that defer consistency actions or perform them asynchronously. Such techniques reduce the number of messages sent at the 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 performance results to client disk caching was investigated in [Franklin et al. 1993]. Client disk caching raises additional problems, such as the relatively large size of disk-based caches (compared to memory caches) and the tradeoffs of accessing data from the local disk vs. obtaining it from a server. Issues that arise when clients are allowed to obtain data from each other (in addition to servers) were studied in [Franklin et al. 1992]. More recently, callback-style approaches have been extended to support multiple granularities of concurrency control and cache consistency [Carey et al. 1994; Chu and Winslett 1994]. 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. The development of distributed database architectures that efficiently support both associative and navigational access is a major focus of our ongoing work.

REFERENCES

Adve, S. and Gharachorloo, K. 1995. Shared memory consistency models: A tutorial. Technical Report 95/7 (Sept.), Digital Western Research Laboratory.

Adya, A., Gruber, R., Liskov, B., and Maheshwari, U. 1995. Efficient optimistic concurrency control using loosely synchronized clocks. In *Proc. of the ACM SIGMOD Conf. on Management of Data* (San Jose, CA, USA, May 1995), pp. 23–34.

Agarawal, A., Simoni, R., Hennessy, J., and Horowitz, M. 1988. An evaluation of directory schemes for cache coherence. In *Proc. 15th International Symposium on Computer Architecture* (Honolulu, HI, June 1988), pp. 208–289.

Alsberg, P. A. and Day, J. D. 1976. A principle for resilient sharing of distributed resources. In *2nd International Conf. on Software Eng., San Francisco* (1976).

Archibald, J. and Baer, J. 1986. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems 4*, 4 (Nov.), 273–298.

Baker, M. G., Hartman, J. H., Kupfer, M. D., Shirriff, K. W., and Ousterhout, J. K. 1991. Measurements of a distributed file system. In *Proc. Thirteenth ACM Symp. on Operating System Principles* (Pacific Grove, CA, Oct. 1991), pp. 198–212.

Bernstein, P. A., Hadzilacos, V., and Goodman, N. 1987. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, MA, USA.

Bhide, A. and Stonebraker, M. 1988. An analysis of three transaction processing architectures. In *Proc. of the Conf. on Very Large Data Bases (VLDB)* (Los Angeles, USA, 1988), pp. 339–350.

Butterworth, P., Otis, A., and Stein, J. 1991. The gemstone object database management system. *CACM 34*, 10 (Oct.), 64–77.

Carey, M., DeWitt, D., and Naughton, J. 1993. The 007 benchmark. In *Proc. of the ACM SIGMOD Conf. on Management of Data* (Washington, DC, USA, May 1993), pp. 12–21.

Carey, M., Franklin, M., and Zaharioudakis, M. 1994. Fine-grained sharing in a page server OODBMS. In *Proc. of the ACM SIGMOD Conf. on Management of Data* (Minneapolis, MI, USA, May 1994), pp. 359–370.

Carey, M. and Livny, M. 1991. Conflict detection tradeoffs for replicated data. *ACM Transactions on Database Systems 16*, 4 (Dec.), 703–746.

Carey, M. J., DeWitt, D. J., Franklin, M. J., Hall, N. E., McAuliffe, M. L., Naughton, J. F., Schuh, D. T., Solomon, M. H., Tan, C. K., Tsatalos, O. G., White, S. J., and Zwilling, M. J. 1994. Shoring up persistent applications. In *Proc. of the ACM SIGMOD Conf. on Management of Data* (Minneapolis, MI, USA, May 1994), pp. 383–394.

Carey, M. J., Franklin, M. J., Livny, M., and Shekita, E. J. 1991. Data caching tradeoffs in client-server DBMS architectures. In *Proc. of the ACM SIGMOD Conf. on Management of Data* (Denver, USA, May 1991), pp. 357–366.

Cattell, R. G. G. 1991. *Object Data Management: Object-Oriented and Extended Relational Database Systems*. Addison-Wesley, Reading, MA, USA.

Cattell, R. G. G. and Skeen, J. 1992. Object operations benchmark. *ACM Transactions on Database Systems 17*, 1 (March), 1–31.

Chu, S. and Winslett, M. 1994. Minipage locking support for page-server database management systems. In *Proc. 3rd International Conference on Information and Knowledge Management* (Gaithersburg, MD, Nov. 1994).

Dan, A., Dias, D. M., and Yu, P. S. 1990. The effect of skewed data access on buffer hits and data contention in a data sharing environment. In *Proc. Int'l. Conf. on Very Large Data Bases* (Brisbane, Australia, Aug. 1990), pp. 419–431.

Dan, A., Dias, D. M., and Yu, P. S. 1991. Analytical modelling of a hierarchical buffer for a data sharing environment. In *Proc. 1991 ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems* (San Diego, California, USA, May 21-24 1991), pp. 156–167. IBM.

Dan, A. and Yu, P. S. 1991. Performance comparisons of buffer coherency policies. In *Proc. 11th Int'l. Conf. on Distr. Computing Sys.* (Arlington, TX, May 1991).

Dan, A. and Yu, P. S. 1992. Performance analysis of coherency control policies through lock retention. In *Proc. of the ACM SIGMOD Conf. on Management of Data* (San Diego, USA, June 1992), pp. 114–123.

Davidson, S. B., Garcia-Molina, H., and Skeen, D. 1985. Consistency in partitioned networks. *ACM Computing Surveys 17*, 3 (Sept.), 341–370.

DeWitt, D. J., Futtersack, P., Maier, D., and Velez, F. 1990. A study of three alternative workstation-server architectures for object-oriented database systems. In *Proc. Int'l. Conf. on Very Large Data Bases* (Brisbane, Australia, Aug. 1990), pp. 107–121.

Dias, D., Iyer, B., Robinson, J., and Yu, P. 1987. Design and analysis of integrated concurrency-coherency controls. In *Proc. Int'l. Conf. on Very Large Data Bases* (Brighton, England, Aug. 1987), pp. 463–471.

Exodus Project Group. 1993. Exodus storage manager architectural overview. Computer Sciences Dept., University of Wisconsin-Madison.

Franklin, M. 1993. *Caching and Memory Management in Client-Server Database Systems*. Ph. D. thesis, University of Wisconsin, Madison. (Published as *Client Data Caching: A Foundation for High Performance Object Database Systems*, Kluwer Academic Publishers, Boston, MA, 1996.).

Franklin, M. and Carey, M. 1992. Client-server caching revisited. In *Proc. International Workshop on Distributed Object Management* (Edmonton, Canada, May 1992), pp. 57–78. (Published as *Distributed Object Management*, Ozsu, Dayal, Vaduriez, eds., Morgan Kaufmann, San Mateo, CA, 1994.).

Franklin, M. J., Carey, M. J., and Livny, M. 1992. Global memory management in client-server DBMS architectures. In *Proc. of the Conf. on Very Large Data Bases (VLDB)* (Vancouver, Canada, 1992), pp. 596–609.

Franklin, M. J., Carey, M. J., and Livny, M. 1993. Local disk caching in client-server database systems. In *Proc. of the Conf. on Very Large Data Bases (VLDB)* (Dublin, Ireland, 1993), pp. 543–554.

Franklin, M. J., Zwilling, M. J., Tan, C. K., Carey, M. J., and DeWitt, D. J. 1992. Crash recovery in client-server EXODUS. In *Proc. of the ACM SIGMOD Conf. on Man-*

agement of Data (San Diego, USA, June 1992), pp. 165–174.

Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P., Gupta, A., and Henessy, J. L. 1990. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proc. 17th International Symposium on Computer Architecture* (June 1990), pp. 15–26.

Goodman, J. R. 1983. Using cache memory to reduce processor-memory traffic. In *Proc. 10th ACM Symp. on Computer Architecture* (Stockholm, Sweden, June 1983), pp. 124.

Gray, J. and Reuter, A. 1993. *Transaction Processing: Concepts and Techniques.* Morgan-Kaufmann Publishers, San Mateo, CA, USA.

Howard, J. H., Kazar, M. L., Menees, S. G., Nichols, D. A., Satyanarayanan, M., Sidebotham, R. N., and West, M. J. 1988. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems 6*, 1 (Feb.), 51–81.

Joshi, A. M. 1991. Adaptive locking strategies in a multi-node data sharing model environment. In *Proc. Int'l. Conf. on Very Large Data Bases* (Barcelona, Spain, Sept. 1991), pp. 181–191.

Keleher, P., Cox, A., and Zwaenepoel, W. 1992. Lazy release consistency for software distributed memory. In *Proc. 19th Annual Int'l Symp. on Computer Architecture* (May 1992), pp. 13–21.

Kemper, A. and Kossmann, D. 1994. Dual-buffering strategies in object bases. In *Proc. of the Conf. on Very Large Data Bases (VLDB)* (Santiago, Chile, Sept. 1994), pp. 427–438.

Kim, W., Garza, J. F., Ballou, N., and Woelk, D. 1990. Architecture of the ORION next-generation database system. *IEEE Transactions on Knowledge and Data Engineering 2*, 1 (March), 109–124.

Kistler, J. J. and Satyanarayanan, M. 1991. Disconnected operation in the coda file system. In *Proc. Thirteenth ACM Symp. on Operating System Principles* (Pacific Grove, CA, Oct. 1991), pp. 213–225.

Lamb, C., Landis, G., Orenstein, J., and Weinreb, D. 1991. The ObjectStore database system. *Communications of the ACM 34*, 10, 50–63.

Lamport, L. 1979. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers C-28*, 9, 690–691.

Levy, E. and Silberschatz, A. 1990. Distributed file systems: Concepts and examples. *ACM Computing Surveys 22*, 3 (Dec.), 321–374.

Li, K. and Hudak, P. 1989. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems 7*, 4 (Nov.), 321–359.

Lilja, D. J. 1993. Cache coherence in large-scale shared-memory multiprocessors: Issues and comparisons. *ACM Computing Surveys 25*, 3 (Sept.), 303–338.

Liskov, B., Day, M., and Shrira, L. 1992. Distributed object management in Thor. In *Proc. International Workshop on Distributed Object Management* (Edmonton, Canada, May 1992), pp. 79–91. (Published as *Distributed Object Management*, Ozsu, Dayal, Vaduriez, eds., Morgan Kaufmann, San Mateo, CA, 1994.).

Livny, M. 1990. Denet user's guide, version 1.5. Computer Sciences Dept., University of Wisconsin-Madison.

Lomet, D. 1994. Private locking and distributed cache management. In *Proc. of the 3rd Intl. Conference on Parallel and Distributed Information Systems* (Austin, TX, Sept. 1994), pp. 151–159.

Mohan, C. and Narang, I. 1991. Recovery and coherency-control protocols for fast intersystem transfer and fine-granularity locking in a shared disks transaction environment. In *Proc. Int'l. Conf. on Very Large Data Bases* (Barcelona, Spain, Sept. 1991), pp. 193–207.

Nelson, M. N., Welch, B. B., and Ousterhout, J. K. 1988. Caching in the sprite network file system. *ACM Trans. on Computer Sys. 6*, 1 (Feb.), 134–154.

Nitzberg, B. and Lo, V. 1991. Distributed shared memory: A survey of issues and algorithms. *IEEE Computer 24*, 8 (Aug.), 52–60.

O. Deux et al. 1991. The O2 system. *Comm. of the ACM, Special Section on Next-Generation Database Systems 34*, 10 (Oct.), 34–49.

Objectivity Inc. 1991. Objectivity/DB Documentation V 1.

ONTOS INC. 1992. Ontos DB 2.2 Reference Manual.

RAHM, E. 1991. Concurrency and coherency control in database sharing systems. Technical Report 3/91 (Nov.), Computer Science Dept., University of Kaiserslautern, Germany.

RAHM, E. 1993. Empirical performance evaluation of concurrency and coherency control protocols for database sharing systems. *ACM Transactions on Database Systems 18*, 2 (June), 333–377.

RAMAKRISHNAN, K. K., BISWAS, P., AND KAREDLA, R. 1992. Analysis of file I/O traces in commercial computing environments. In *Proc. 1992 ACM SIGMETRICS and PERFORMANCE '92 Int'l. Conf. on Measurement and Modeling of Computer Systems* (Newport, Rhode Island, USA, June 1-5 1992), pp. 78–90.

SANDHU, H. AND ZHOU, S. 1992. Cluster-based file replication in large-scale distributed systems. In *Proc. 1992 ACM SIGMETRICS and PERFORMANCE '92 Int'l. Conf. on Measurement and Modeling of Computer Systems* (Newport, Rhode Island, USA, June 1-5 1992), pp. 91–102.

STENSTROM, P. 1990. A survey of cache coherence protocols for multiprocessors. *IEEE Computer 23*, 6 (June), 12–24.

STONEBRAKER, M. 1979. Concurrency control and consistency of multiple copies of data in distributed INGRES. *IEEE Trans. on Softw. Eng. 5*, 3 (May), 188–194.

STURGIS, H., MITCHELL, J., AND ISRAEL, J. 1980. Issues in the design and use of a distributed file system. *Operating Systems Review 14*, 3 (July).

VERSANT OBJECT TECHNOLOGY. 1991. VERSANT system reference manual, release 1.6.

WANG, Y. AND ROWE, L. A. 91. Cache consistency and concurrency control in a client/server DBMS architecture. In *Proc. of the ACM SIGMOD Conf. on Management of Data* (Denver, CO, May 91), pp. 367–377.

WILKINSON, K. AND NEIMAT, M.-A. 1990. Maintaining consistency of client-cached data. In *Proc. of the Conf. on Very Large Data Bases (VLDB)* (Brisbane, Australia, 1990), pp. 122–134.