Empirical Performance Evaluation of Concurrency and Coherency Control Protocols for Database Sharing Systems

ERHARD RAHM
University of Kaiserslautern

Database Sharing (DB-sharing) refers to a general approach for building a distributed high performance transaction system. The nodes of a DB-sharing system are locally coupled via a high-speed interconnect and share a common database at the disk level. This is also known as a "shared disk" approach. We compare database sharing with the database partitioning (shared nothing) approach and discuss the functional DBMS components that require new and coordinated solutions for DB-sharing. The performance of DB-sharing systems critically depends on the protocols used for concurrency and coherency control. The frequency of communication required for these functions has to be kept as low as possible in order to achieve high transaction rates and short response times. A trace-driven simulation system for DB-sharing complexes has been developed that allows a realistic performance comparison of four different concurrency and coherency control protocols. We consider two locking and two optimistic schemes which operate either under central or distributed control. For coherency control, we investigate so-called on-request and broadcast invalidation schemes, and employ buffer-to-buffer communication to exchange modified pages directly between different nodes. The performance impact of random routing versus affinity-based load distribution and different communication costs is also examined. In addition, we analyze potential performance bottlenecks created by hot spot pages.

Categories and Subject Descriptors: C.2.4. [**Computer-Communication Networks**]: Distributed Systems; C.4 [**Computer Systems Organization**]: Performance of Systems; D.4.8 [**Operating Systems**]: Performance—*simulation*; H.2.4 [**Database Management**]: Systems—*distributed systems, transaction processing*

General Terms: Algorithms, Design, Performance

Additional Key Words and Phrases: Coherency control, concurrency control, database partitioning, database sharing, performance analysis, shared disk, shared nothing, trace-driven simulation

This work was financially supported by Siemens AG, Munich.
Author's address: University of Kaiserslautern, Department of Computer Science, 67678 Kaiserslautern, Germany.

1. INTRODUCTION

Large transaction processing applications, such as banking, flight reservation, or telecommunication networking, increasingly demand high performance transaction systems [24]. Such DB-based systems should not only provide high transaction rates (several thousands of transactions per second with response times acceptable for online applications), but also high availability, scalability (modular growth), and manageability and maintainability [24, 32].

Database Sharing (DB-sharing, "shared disk" [82]) systems, which have received considerable attention in recent years [2, 6, 15, 31, 36, 47, 53, 68, 72, 73, 79, 82, 84, 86], offer a promising solution to these problems. They consist of multiple autonomous processing nodes that share the common database at the disk level (see Figure 1a). A node in such a system may be either a uniprocessor or a tightly coupled multiprocessor with a local main memory and a separate copy of the operating system and the database management system (DBMS). In this paper, we usually assume that the nodes are loosely coupled (message-oriented communication), as opposed to closely coupled DB-sharing systems. A close coupling aims at a more efficient internode cooperation for certain functions by using common semiconductor memory [8, 17, 64] or by using special-purpose processors for global services such as concurrency control (e.g., by a "lock engine" [37, 69, 72]).

Existing DB-sharing systems and prototypes include the IMS Data Sharing product [56, 79], the Power System 5/55 of Computer Console [84], the Data Sharing Control System of NEC [72], the Amoeba prototype [73, 82], Fujitsu's Shared Resource Control Facility [2], and DEC's DBMS and Rdb/VMS products within a VaxCluster [38, 39, 42, 66]. Recently, Oracle has also introduced a version of its DBMS product, called "parallel server" which supports database sharing on different hardware platforms [53]. Furthermore, the IBM operating system TPF (transaction processing facility) [71, 81] also supports disk sharing for up to eight nodes and performs locking and data caching within the disk control units.

Comparison to Other Approaches

Database partitioning (DB-partitioning, "shared nothing" [78]) refers to another general approach for distributed transaction processing (Figure 1b). In contrast to DB-sharing, the database in shared-nothing systems is partitioned so that each node owns some fraction of the disk devices. This approach is adopted by Tandem's Encompass and NonStop SQL products [10, 80] and by several database machines, e.g., Teradata's DBC/1012 [50] and the Bubba and Gamma prototypes [9, 16]. Typically, *database machines* only process the database operations while application programs submitting the operations are executed on front-end processors, which are usually mainframe hosts or workstations. This approach is also used in other *client/server architectures*, and can effectively utilize the processing capacity of inexpensive microprocessors for transaction processing. A distributed server, needed for high availability and high transaction rates, may be based either on the

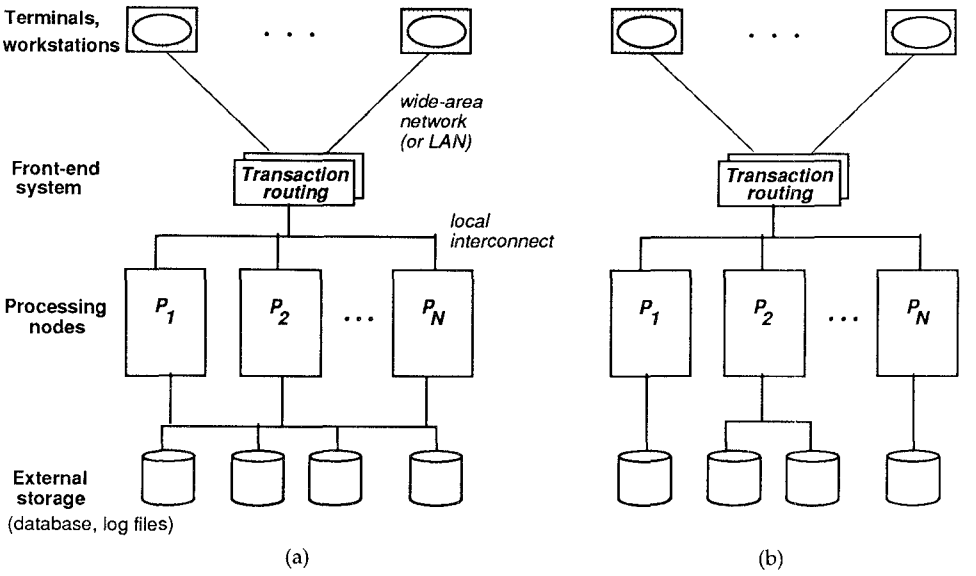

Fig. 1. DB-sharing versus DB-partitioning (a) DB-sharing (b) DB-partitioning.

DB-partitioning or DB-sharing approach. However, separating application and database processing across node boundaries can cause performance problems for online transaction processing (OLTP). This is because internode communication is required for every database operation in order to submit the request to the server and to return the result to the application program. For simple operations prevalent in typical OLTP applications, the associated communication overhead and response time delay dominates the execution cost and time of the operation itself. Off-loading database processing may be appropriate for complex queries by processing them in parallel on multiple back-end processors.

The controversy among DB-sharing and DB-partitioning systems has been discussed in the literature [32, 55, 74, 78]. Although research and system developments have concentrated on the DB-partitioning approach (including distributed database systems), we feel the DB-sharing approach offers a number of advantages that make it attractive for high performance transaction processing. In the following, we summarize the major differences between the two approaches:

(1) *Locally vs. geographically distributed systems.* DB-partitioning is applicable in locally and geographically distributed systems. DB-sharing, on the other hand, requires that all physical components are located in close proximity, due to the attachment of the disk drives to the nodes. New fiberoptic disk connections permit extending the distance from one room to several kilometers between processing nodes and disk drives.

A geographically distributed transaction processing system is desirable in order to reflect the organizational structure of large institutions. In addition,

it can offer better failure isolation than a locally coupled system. On the other hand, locally coupled systems allow a simpler administration and avoid problems of geographically distributed systems introduced by the autonomy requirements of individual sites, or unreliable communication paths or network partitions. In addition, locally coupled systems can utilize a higher communication bandwidth and streamlined communication protocols, thus improving performance for distributed transactions. Furthermore, local systems offer a greater flexibility for global load control and load balancing [13]. If for instance, transactions are routed to the nodes via a front-end system (Figure 1), a dynamic load distribution becomes feasible, which takes the current load situation into account. This promises a better utilization of the available resources, as in geographically distributed systems where workload allocation is typically static (predetermined assignment of terminals to nodes). Locally coupled systems can be protected against disasters by a limited form of geographic data distribution. As discussed in [11, 41, 46], this is possible by keeping a copy of the database at a passive remote data center. The database copy is not used for normal transaction processing, but only kept up to date by spooling log data from the primary to the remote data center. In the event of a disaster, the backup system takes over and continues transaction processing. Such an approach is applicable to DB-partitioning as well as DB-sharing.

(2) *Database design.* The key problem of DB-partitioning is the difficulty of finding a "good" fragmentation and allocation of the database [70]. This not only complicates system administration, but has far-reaching consequences on how transactions are processed and on the achievable performance. The problem is aggravated by the fact that database allocations tend to be rather static, since physical database relocations are expensive. Furthermore, a sufficient flexibility to adapt database partitioning is only provided if fine-grained fragmentation units are supported, e.g., by a horizontal partitioning of relational databases. For navigational databases, on the other hand, only coarse fragmentation units (segments, record types, etc.) are usually possible. This severely restricts the number of database partitions and processing nodes, and makes it difficult to achieve load balancing. Similar problems may be posed by complex objects in object-oriented database systems, e.g., for CAD applications. As pointed out in [34], partitioning of such objects is a major problem.

DB-sharing does not encounter these problems because there is no need to physically partition the database. As a result, no changes to the physical or logical database structure are necessary when migrating from a centralized system to DB-sharing or when the number of nodes in the DB-sharing system changes. Hence, DB-sharing can support nonrelational databases more easily than DB-partitioning.

(3) *Transaction execution model.* For DB-partitioning, the execution of a database operation is distributed if nonlocal data has to be accessed. Considerable extensions to the query optimizer are necessary in order to support the construction of distributed execution plans. If nonlocal data has been accessed

by a transaction, a distributed commit protocol has to be employed to guarantee the all-or-nothing property [33] of the transaction. Similarly, rollback of a transaction is distributed if the transaction has referenced external data. After a change in the data allocation, precompiled execution plans may become invalid, requiring a new compilation and optimization of affected application programs.

In DB-sharing systems, no distributed execution plans have to be determined by the query optimizer because each node has direct access to the entire database. Nevertheless, communication may be necessary to process a given operation, e.g., for concurrency and coherency control (see Section 2). These tasks, however, are performed by lower layers of the DBMS, and are thus transparent to the query optimizer. The first commit phase of a transaction is always local for DB-sharing; that is, a transaction is committed as soon as the commit record has been written to the local log file of the node where the transaction has been started [63]. Communication may be needed during the second commit phase to release externally managed locks.

Both approaches, DB-partitioning and DB-sharing, can employ parallel execution strategies for complex queries and process independent (nonconflicting) suboperations on different nodes in parallel. For DB-partitioning, the physical data distribution determines whether a parallel execution is applicable for a given operation. This limitation is removed for DB-sharing; even the same data can be concurrently read in different nodes, thus increasing the flexibility for parallel processing models. Obviously, the query optimizer needs to be enhanced for DB-sharing, too, in order to support parallel execution strategies.

(4) *Availability and scalability.* DB-sharing systems offer availability and extensibility advantages over the DB-partitioning approach because they can cope more easily with variations in the number of nodes. In DB-partitioning systems, a node crash makes the partition of the failed node unavailable unless another node is connected to the disk devices of the failed node. Even if the partition can be taken over, the node temporarily owning two partitions becomes easily overloaded. Even more difficult is the addition of a new node to a DB-partitioning complex, since it requires a physical redistribution of the database ($N \rightarrow N + 1$). This is generally a major reorganization and affects the availability of the data to be moved.

DB-sharing avoids these problems, since no physical data allocation needs to be adapted. On the other hand, depending on the I/O architecture, the number of nodes that can be connected to the same disk may be limited. For message-oriented storage devices that communicate with processing nodes by means of message passing, there is no inherent limit in the number of processing nodes. Such an approach has been chosen in the VaxClusters [42] which currently support up to 96 processing nodes and disk servers, as well as in parallel hypercube architectures (e.g., Intel's iPSC/860 or NCUBE).

(5) *Workload allocation and load balancing.* Although a dynamic workload allocation is possible for both approaches in a locally distributed system, DB-sharing offers a much higher flexibility for load balancing. The problem

of DB-partitioning is that at least simple database operations, which reference one or a few records, have to be processed where the data resides. That is, the physical data allocation determines where an operation is going to be processed, irrespective of the node to which the corresponding transaction has been assigned. If, for load balancing reasons, a transaction is assigned to a lightly loaded node which holds no or little of the data to be accessed, the database operations still have to be processed at the nodes owning the required data. Thus the load on these nodes is not significantly reduced and may even be increased, since additional communication overhead is introduced for receiving the remote requests, returning the results and participating in the commit protocol. As a consequence, resource utilization and message frequency are largely determined by data allocation, leaving only a small optimization potential for transaction routing. Variations in the load profile can thus easily lead to unbalanced CPU utilizations and overload situations at some nodes. In particular, partitioned systems do not have the flexibility to adapt to short-term workload fluctuations [74].

In DB-sharing systems, on the other hand, a database operation can largely be processed at the node to which the corresponding transaction has been assigned, since each node has direct access to the entire database. For instance, it is possible to allocate complex ad hoc queries and short online transactions to separate nodes so as to avoid resource contention on CPU and memory between these workload types. Furthermore, it is possible to free whole nodes from transaction processing when the current load can be handled by a smaller number of nodes [74]. For DB-sharing, the routing strategy is not determined by a static data allocation, and can thus be more easily adapted to short-term workload fluctuations and other changes in the system state.

(6) *Technical problems.* The discussion above already shows that various technical problems have to be solved for DB-partitioning and DB-sharing. Major problems for DB-partitioning include database design (database fragmentation and allocation), distributed query processing, and distributed commit algorithms. Geographically distributed systems pose a number of extra challenges: dealing with network partitions, authorization, catalog management, and replication control. DB-sharing systems have to address new problems in the areas of concurrency control, coherency control, logging, and crash recovery (see Section 2). Some problems are relevant for both DB-partitioning and DB-sharing: global deadlock management, disaster recovery, workload allocation/load balancing, and parallel query optimization.

Organization

In this paper we investigate the concurrency and coherency control problems in DB-sharing systems. Both functions are critical to the performance of such systems because they largely determine the communication overhead for transaction processing. We have implemented four different protocols within a detailed trace-driven simulation system that allows a direct performance

comparison between the schemes. The algorithms have been designed to meet high performance requirements by utilizing various concepts to limit the message and I/O frequency. In particular, we employ a so-called NOFORCE strategy for update propagation to disk and to exchange modified pages directly across the interconnect rather than across the shared disks. In addition, affinity-based transaction routing, controlled by routing tables, is used to assign the workload. Such a routing aims at improving performance by supporting locality of reference within nodes.

The next section provides a discussion of the DB-sharing components, requiring new solutions, compared to centralized and DB-partitioning systems. In Section 3 we describe the four concurrency and coherency control protocols that have been implemented. Section 4 presents our trace-driven simulation system of loosely coupled DB-sharing configurations, together with the load characteristics and our parameter settings. Simulation results for the four protocols are then analyzed in Section 5. Section 6 deals with related performance studies for DB-sharing. Finally, we summarize our main conclusions and indicate areas for future research.

2. FUNCTIONAL COMPONENTS IN DB-SHARING SYSTEMS

In this section we discuss the four major components that require new solutions for DB-sharing. These functions are global concurrency control, coherency control, load control, and logging/recovery.

2.1 Concurrency Control

Since any data item of the shared database can be accessed by any node, DB-sharing requires global synchronization in order to preserve serializability of transaction processing. While concurrency control is basically a local function with DB-partitioning, where each node synchronizes accesses against its partition, loosely coupled DB-sharing systems require explicit message exchange for system-wide synchronization. The number of concurrency control messages has to be as low as possible in order to reduce the communication overhead and support high performance. Especially critical are so-called *synchronous* messages (for instance, global lock requests) which entail transaction deactivation, and hence process switching, until a response message is received. These messages not only increase overhead due to process switches, but also increase a transaction's response time, and thus data contention. Data contention can be a limiting factor for throughput, particularly in distributed environments where generally higher multiprogramming levels than in centralized DBMSs have to be dealt with. To limit the communication frequency it is also important (as we shall see) to treat concurrency and coherency control by an integrated protocol. A further requirement for a practical concurrency control protocol is robustness against failures in the system, in particular against node crashes.

A survey of concurrency control schemes for DB-sharing is given in [60, 65]. The most appropriate approaches are locking and optimistic concurrency control methods that operate either under central or distributed control. For

each of the resulting four categories, we examine one representative scheme (see Section 3).

2.2 Coherency Control

Coherency control is necessitated by the caching of database pages in main memory database buffers. Caching may in large database buffers substantially reduce the amount of expensive and slow disk accesses by utilizing locality of reference. Unfortunately, there is a *buffer invalidation* problem in DB-sharing systems, since a particular page (block) may simultaneously reside in the database buffers of different nodes. Thus, modification of the page in any buffer invalidates copies of that page in other nodes, as well as the page copy stored on disk. The basic task of coherency control is to ensure that transactions always see the most recent version of database objects despite buffer invalidations.

This buffer invalidation problem is analogous (although at a different level of the storage hierarchy) to the cache coherency problem in tightly coupled multiprocessors [18, 85] and to the replication control problem in distributed databases [5, 22, 76]. Coherence problems for pages cached in main memory buffers have recently also been studied in the context of network file systems [40, 51] and in so-called Distributed Shared Memory (DSM) systems [4, 7, 44, 52]. The main difference between the latter studies and DB-sharing comes from which correctness criterion for coherency control is supported. In network file systems, it is desirable that every access to a file block should return the most recently written version of that block [51]. In DSM systems, every machine instruction must be provided with the most recent page version. For DB-sharing, on the other hand, transactions are the basic execution units, and it must be guaranteed that every transaction sees all modifications of previously committed transactions. (Actually, the correct criterion of serializability demands only that a transaction sees a transaction-consistent state of the database, not necessarily the most recent one.) Since transactions typically encompass tens to hundreds of page accesses and tens to hundreds of thousands instructions, we can expect a significantly lower number of page transfers between nodes for DB-sharing than for network file systems or DSM systems. In addition, DB-sharing permits a combination of coherency and concurrency control and may utilize an affinity-based transaction routing (see below) to limit the number of buffer invalidations and page transfers.

Workstation/server DBMS with a caching of database pages in workstations require coherency controls similar to DB-sharing systems [12, 83]. In fact, the coherency control mechanisms for DB-sharing can be employed in this environment as well [65].

Solving the buffer invalidation problem for DB-sharing depends on two major factors, namely the data granularity for concurrency control and the strategy used for update propagation to disk.

(1) *Page versus record-level concurrency control.* Page-level concurrency control simplifies coherency control, since pages are the allocation units in the database buffers and transfer units between main memory and disk.

Furthermore, a smaller number of lock requests, and thus potentially less communication, is needed compared to record-level locking. On the other hand, page-level concurrency control may cause an unacceptable amount of data contention on frequently accessed "hot spot" pages. Record-level locking permits different records of the same page to be concurrently modified at different nodes. As a result, each node's page copy is only partially up-to-date and the page copy on disk contains none of the modifications (at first). Since writing out partially up-to-date pages could lead to lost updates, all modifications must be merged before the page can be written to disk. This merging of record modifications is cumbersome, in general, and may cause a substantial number of extra messages. Furthermore, even with record-level locking, short page locks/latches may be needed in order to serialize modifications to page control information (e.g., free space information in the page header). For DB-sharing, this requires additional messages and seems impractical.

To avoid the coherency control problems associated with record-level locking, existing DB-sharing systems only support page locking or a restricted form of record locking. A good compromise may be to restrict page-level concurrency control to concurrent write accesses of different systems, so that different records of a page can be concurrently modified within the same node. In addition, any record not being modified may be read in any system [47]. Full record-level locking, i.e., concurrent modifications of the same page in different nodes, may be tolerable for updates that leave the page structure unchanged (modification of existing records) [49, 63, 65]. Another approach to reduce data contention is to utilize the semantics of update operations, e.g., on index structures or for certain high-traffic data items (commutativity of increment/decrement operations) [28, 49].

(2) *FORCE versus NOFORCE strategy for update propagation to disk.* A FORCE strategy for update propagation requires that all pages modified by a transaction are forced (written) to the permanent database on disk before commit [33]. This approach is usually unacceptable for high performance transaction processing—since a high I/O overhead and significant response time delays for update transactions are introduced. For DB-sharing, this strategy implies further page-level concurrency control for updates between different nodes in order to serialize disk writes for the same page. On the other hand, coherency control is simplified because the most recent version of a page can always be obtained from disk. Furthermore, there is no need for redo recovery after a node failure. Despite the performance problems of the FORCE approach, most existing DB-sharing systems currently use the FORCE strategy.

The NOFORCE alternative permits a drastically reduced I/O overhead and avoids the response time increase due to synchronous disk writes at end of transaction (EOT). Only redo log data is written to a log file at EOT (possibly by utilizing group-commit [35]), and multiple modifications can be accumulated per page before it is written to disk. Since the permanent database on disk is not up-to-date, in general, coherency control has to keep track of where the most recent version of a modified page can be obtained.

Instead of reading a page from disk, a so-called *page request* may have to be sent to the node holding the current page copy in its buffer. The modified page can then be transferred to the requesting node either directly across the communication system through a "buffer-to-buffer" communication or across the shared disks. With a high-speed interconnect, the direct transmission is typically much faster, say 1 ms versus 50 ms for the two I/Os needed for a page exchange across shared disks. Nonvolatile disk caches could permit transfer delays of about 4–8 ms for the two I/Os. In closely coupled DB-sharing systems, modified pages may also be exchanged across shared semiconductor stores with faster access times than disk caches [64].

A disadvantage of buffer invalidations is that obsolete page copies cannot be reused, thus reducing buffer hit ratios. In addition, performance is degraded by extra messages that may be required for coherency control. On the other hand, data replication in main memory permits multiple nodes to concurrently read the same data. Furthermore, a buffer miss does not necessarily imply a need to read the page from disk. Rather, a page may be obtained much faster from another node.

A survey of coherency control methods for DB-sharing systems is presented in [65]. Buffer invalidations can either be detected or avoided. A simple but expensive scheme uses broadcast messages to indicate the pages which may have been invalidated by an update transaction (broadcast invalidation). A better approach avoids extra messages to detect buffer invalidations by checking the validity of cached pages during lock request processing (on-request invalidation). Another method is to avoid buffer invalidations by purging a page from the database buffer before its modification at another node. The schemes described in the next section are based either on the broadcast invalidation or on-request invalidation scheme.

2.3 Load Control and Load Balancing

The main task of load control is transaction routing, that is, the assignment of incoming transaction requests to the nodes of the DB-sharing complex. This workload allocation should not be determined statically by a fixed allocation of terminals and/or application programs to nodes, but should be automatic and adaptive with respect to changing conditions in the system such as overload situations, changed load profile, and node crashes. Effective routing schemes not only aim at achieving load balancing to primarily limit CPU resource contention, but also at supporting efficient transaction processing so that given response time and throughput requirements can be met. A general approach to achieve this goal is *affinity-based transaction routing* which uses information about the reference behavior of transaction types to assign transactions with an affinity to the same database portions to the same node [62]. In this way it strives to achieve what we call *node-specific locality of reference*, which requires that transactions running on different nodes should mainly access disjoint portions of the shared database. This is a promising approach because improved locality of reference supports better hit ratios and thus fewer disk I/Os. Similarly, node-specific locality helps to reduce the number of buffer invalidations and page transfers between nodes.

Furthermore, locality of reference can be utilized by some concurrency control schemes to limit the number of synchronization messages (see Section 3).

The achievable degree of node-specific locality of reference is not only determined by the routing strategy, but also depends heavily on the load characteristics and the number of nodes. Node-specific locality is hard to achieve if the references of a transaction type are spread over the entire database, if there are database areas that are referenced by most transactions, or in the case of transaction types which cannot be processed on a single node without overloading it. Additionally, the more nodes are to be utilized, the less node-specific locality can generally be achieved, unless new transaction types and/or database partitions are also added to the system. Our analysis of a number of traces from different DBMS and customers shows that these characteristics are quite common in real database applications. None of the workloads we examined was nearly as "delightful" and "scalable" [78] as the well-known debit-credit transaction load [3, 25]. This load is completely homogeneous (one transaction type with every transaction updating four record types) and permits an almost ideal partitioning of the workload and database for different numbers of nodes. The amount of achievable node-specific locality is thus very high and almost independent of the number of nodes. (According to the benchmark definition in [3, 25], up to 15% of the account accesses may require communication. Since typically the other three record types can be locally accessed, at most 3.75% of all accesses are remote.)

A more detailed discussion of transaction routing and a framework for classifying different approaches can be found in [62]. In [61], a hierarchical model for a comprehensive workload management for transaction processing is proposed where load control takes place at various levels. It is based on feedback loops, for local and global system control, that periodically analyze monitor data to detect performance problems and initiate corrective actions if necessary. Apart from improving performance, the main objective is to simplify system administration by automatically controlling the most important control parameters (routing strategy, multiprogramming level, transaction priorities, etc.). Only those problems are reported, together with hints on possible reasons, for which automatic corrections could not be applied or did not prove effective.

2.4 Logging and Recovery

Each node of the DB-sharing system maintains a local journal where the modifications of locally executed transactions are logged. This information is used for transaction abort and crash recovery. For media recovery, a global journal may be constructed by merging the local log data [48, 73]. Preferably, the global log is constructed on-the-fly to support quick recovery after a disk failure. Existing DB-sharing systems either use mirrored disks to handle disk failures or provide a tool for merging the local log files offline. The latter approach is much easier to realize than an online construction of the global log, but prevents a fast recovery from disk failures, thus limiting availability. A discussion of the problems for creating global logs can be found in [48].

Apart from media and disaster recovery, crash recovery is the major issue that requires new solutions for DB-sharing. Crash recovery has to be performed by the surviving nodes which use the local journal of the failed node in order to provide high availability. The realization of this recovery form depends on many factors, including the underlying protocol for concurrency and coherency control, the strategy used for update propagation (FORCE versus NOFORCE), and the log and concurrency control granularities [63]. In general, lost effects of transactions committed at the failed node have to be redone while modifications of in-progress, hence failed, transactions may have to be undone. If modified pages are directly exchanged between nodes across the communication system, crash recovery may require use of a global log in order to redo the modifications of affected pages in correct order. Special recovery actions may be necessary for the reconstruction of lost control information in order to properly continue concurrency and coherency control.

Recovery in DB-sharing systems is discussed in more detail in [47] and [63].

3. CONCURRENCY AND COHERENCY CONTROL PROTOCOLS

For this study we have chosen to examine two locking and two optimistic concurrency control protocols, with one centralized and one distributed algorithm for each class. For coherency control, we apply either the broadcast invalidation or on-request invalidation approach. To support high performance, our coherency protocols are designed for a NOFORCE environment and use buffer-to-buffer communication to propagate modified pages to other nodes. All algorithms assume page-level concurrency control to facilitate the integration of concurrency and coherency control. Record-level concurrency control could not be evaluated because our traces provide only reference information at the page level.

3.1 Central Lock Manager (CLM)

In the simplest form, every lock request and release is forwarded to the CLM node. This results in two messages per lock request and one message at EOT to release all locks. Such an approach can be considered as a worst-case protocol. Batching of messages reduces the communication overhead, but at the expense of increased delays for the synchronous lock requests and thus increased response times. We have incorporated two other techniques into the CLM scheme which utilize locality of reference and are able to reduce both the communication overhead and response times.

—A so-called *read optimization* [57, 60, 65] is applied that allows multiple nodes at the same time to grant and release read locks for a page locally, without contacting the CLM. The first read access to a page B in a node has to be granted by the CLM. If no write lock request is known at the CLM at this point in time, the CLM assigns a so-called *read authorization* for B to the requesting node. This read authorization gives the node the permission to process all further read lock requests and releases for B locally, thus

reducing the number of synchronization messages and response time delays. The effectiveness of this technique increases with increasing locality of read accesses.

Since multiple nodes can simultaneously hold a read authorization for the same page, the read optimization supports a high concurrency and fast read accesses, which are also supported by the data replication in the database buffers. Write accesses, however, may suffer from this technique since a write lock cannot be granted until the CLM has revoked all read authorizations. Thus the read optimization seems less attractive for applications where update accesses are more frequent than read accesses.

—A similar concept, called *sole interest* [68], is applied to grant an authorization for a local synchronization of read *and* write requests. Such a *write authorization* is assigned to a node, when it requests a lock at the CLM and no other node has issued a lock request for the same page ("sole interest"). In contrast to read authorizations, a write authorization can be assigned only to one node at a time, and has to be revoked by the CLM as soon as any other node requests a read or write lock for the same page. If a read request causes the sole interest revocation, the write authorization is degraded into a read authorization. Otherwise the write authorization of the current owner is given up and assigned to the requesting node (if there are no waiting requests from other nodes).

The sole interest concept pays off only if more lock requests can be locally satisfied than sole interest revocations occur. This is because four messages are required for a lock request causing a sole interest revocation, compared to two messages without sole interest concept. In contrast to the read optimization, the effectiveness of sole interest depends on the amount of *node-specific* locality of reference requiring that different nodes should reference different portions of the database.

Both techniques could also be applied for coarser granularities than pages, e.g., record types or segments. For instance, if a node holds a write (read) authorization for an entire segment, all (read) lock requests against this segment can be locally synchronized. Such a hierarchical scheme has not been implemented, mainly because of complexity reasons. Also, for "important" segments or record types to which a substantial share of the database references is directed, it is generally unlikely that only one node has interest or that only read references are issued for longer periods of time. Rather, thrashing-like situations with only short-lived assignments and frequent revocations of read/write authorizations could occur, which causes more messages than are saved.

For coherency control, a simple *broadcast invalidation* scheme is used. At the end of every update transaction, a broadcast message is sent to all nodes indicating which pages have been modified. Invalidated page copies can thus be removed from the buffers and access to them is avoided. With NOFORCE, additional provisions are required in order to provide a transaction with the most recent page copies (remember that the page versions on disk may be obsolete). For this purpose, every buffer manager maintains a so-called

modified-blocks table (MBT). The MBT indicates for all (recently) updated pages the node where the latest modification has been performed, and thus where the current page version can be requested. The MBT is maintained without additional communication overhead by using information of the broadcast messages. By periodically broadcasting modified pages which have been written to disk, the number of MBT entries can be limited and page requests that can no longer be satisfied by buffer-to-buffer communication are largely avoided. These notifications can be piggy-backed on the broadcast invalidation messages.

A CLM scheme is used in the DB-sharing systems of Computer Console and NEC as well as in the Amoeba prototype. They rely on the sole interest concept (for coarser granules than pages, however) for reducing the communication overhead, while a read optimization is unknown in existing data sharing systems. The broadcast invalidation scheme is used for coherency control, but in combination with FORCE, and thus with an exchange of modified pages across shared disks. In the Amoeba project [73] the existence of a nonvolatile shared semiconductor store has been assumed for speeding up the exchange of modified data.

3.2 Primary Copy Locking (*PCL*)

In this distributed scheme, the database is divided into logical partitions, and each node is assigned the synchronization responsibility, or *primary copy authority* (*PCA*), for one partition [68]. Lock requests against the local partition can be handled without communication overhead and delay, while other requests have to be directed to the authorized node holding the PCA for the respective partition.

A simple PCL scheme results if the PCA allocation is determined by a hash function such that each node controls the same number of hash classes. If all hash classes are referenced with similar probability, we yield an average of $(2 - 2/N)$ messages per lock request, where N stands for the number of nodes. The difference from the straightforward CLM scheme (two messages per lock request) shrinks as the number of nodes grows (1.5 messages for $N = 4$, but already 1.9 for $N = 20$). On the other hand, the CLM node is likely to become the system's bottleneck with growing N, at least in the simple approach, while with PCL the concurrency control overhead is distributed among all nodes.

We implemented a PCL protocol with two major enhancements to reduce the number of synchronization messages:

—A *read optimization* is employed for the primary copy scheme where the read authorizations are assigned and revoked by the PCA lock managers. This permits a local read synchronization of pages belonging to the partition of another node.

—We *coordinate the allocation of PCAs and the workload distribution* such that transaction types are generally allocated to the node where most references can be locally synchronized. This kind of affinity-based transaction routing is accomplished by using a predetermined routing table indi-

cating the best node(s) for every transaction type [67]. The computation of the routing table (see Section 4) assumes some knowledge of the reference pattern of the most important transaction types, as could be obtained from a DBMS-internal monitor.

Coordinating PCA and load allocation aims at achieving node-specific locality of reference to reduce communication frequency. Contrary to sole interest assignments, PCA allocations are stable and independent of whether one or several nodes reference a partition. Also, there is no analogous disadvantage to the expensive revocations of write authorizations in the CLM scheme. In contrast to the static data allocation in DB-partitioning systems, PCA distribution can be dynamically adapted together with the routing strategy (table), since it is only represented by internal control structures. This would typically be done when the load profile changes significantly, or less frequently, after a node has failed or been added.

For coherency control, an *on-request invalidation* (check-on-access) scheme is applied. It uses extended information in the lock table, which allows the PCA lock manager to decide on the validity of a buffer page together with the lock request processing. Thus, buffer invalidations are detected without any additional communication—a big advantage compared to broadcast invalidation schemes. The on-request invalidation approach is used in DEC's VaxClusters [42], but in conjunction with a FORCE strategy. In [57], we describe two realization strategies for PCL that use either page version (sequence) numbers or so-called invalidation vectors to detect buffer invalidations. Both schemes incur no communication overhead, but the invalidation vector solution used in the simulation does not depend on version numbers stored within pages.

The coherency protocol for NOFORCE has been designed such that transmissions of modified pages can also be combined with regular concurrency control messages. First, modified pages belonging to the partition of another node are transmitted to the responsible PCA site, together with the message required for releasing the write lock at EOT. This has the effect that the PCA node always gets the most recent page versions for its partition. Buffer invalidations are now limited to pages belonging to another node's partition. Moreover, when a lock is granted to an external transaction, the PCA node can send the most recent page version directly to the requesting transaction, together with the lock response message.[1] In this case, the requesting transaction does not need to be deactivated again for requesting the page from another node or reading it from disk. Therefore, our coherency control scheme requires neither extra messages for detecting buffer invalidations nor for exchanging modified pages between different sites; more details can be found in [57].

[1]This would be the case if the requesting node holds no copy of the page in its buffer or only an obsolete one (detected by the PCA lock manager). If the PCA node does not hold a copy of the page in its buffer, it indicates in the lock response message that the page can be read from disk.

Note that not only a local PCA but also a read authorization guarantees the validity of a buffer page. This is because the read authorization indicates that the page has not been modified since the authorization was obtained from the PCA lock manager. Read authorization can be given back as soon as the page is replaced from the buffer, since for the next reference the PCA node has to be contacted anyway in order to get a current page copy, because the copy on disk may still be obsolete. Such a voluntary return of read authorizations reduces the number of lock table entries and can largely avoid delays for write locks due to revocations of read authorizations. Read authorizations for frequently referenced pages will not be given up voluntarily because these pages are not removed from the buffer unless they become invalidated. On the other hand, if a read authorization is explicitly revoked, the corresponding page can also be purged from the buffer, since it is going to become invalidated.

Recovery protocols for PCL are presented in [63]. It turns out that an explicit construction of a global log can be avoided with this approach if the PCA nodes log all modifications against their partition. This is possible without extra communication because all modifications are transferred to the PCA nodes at EOT in commit phase 2.

3.3 Central Validation Scheme (CV-OCC)

In this optimistic concurrency control (OCC) [43] scheme, all validations are performed at a designated site. A main reason to investigate such a scheme is that only one synchronous concurrency control request per transaction is required, namely the validation request sent at EOT. For performing the validations at the central site, we implemented a simple and efficient scheme using timestamp comparisons for conflict detection (for details, see [59]). Validation verifies whether or not the page copies referenced by the validating transaction are still up to date. If not, validation fails and the validating transaction is aborted.

An inherent problem of OCC is the danger of a high abort rate and starvation (i.e., a transaction may never succeed due to permanent restart). To address this problem we adopted a special combination with locking techniques similar to that suggested in [68]. For unsuccessful transactions, we perform a "preclaiming" at the central site, right after the failed validation. Thus, before reexecution of the transaction is started, locks are acquired for all pages referenced during the first execution. These locks prevent the invalidation of the respective pages again and guarantee a successful second execution, at least if no additional objects are referenced. Note that no extra communication is required for setting the locks and that deadlocks are also avoided. However, lock conflicts with other failed transactions can occur.

Though access to invalidated pages is detected during validation, coherency control is still required to reduce the number of aborts by removing obsolete pages early from the buffers and providing the current page copies. For this purpose, a *broadcast invalidation* scheme is applied which also relies on replicated MBTs to indicate the nodes where modified pages can be requested (as in the CLM scheme). Here, however, the broadcast invalidation messages

are sent by the central site, after the successful validation of an update transaction. For the site where the successful transaction has been executed, the broadcast message also serves as notification that validation has been successful, so that a separate validation response message is saved.

3.4 Optimistic Token Ring Protocol (TR-OCC)

In this distributed OCC scheme, a node performs validations only while it is holding the token [30]. After a transaction has reached its EOT, it first has to wait until the token arrives in order to validate itself against local transactions. For validation against external transactions, the validation request is sent around the ring, together with the token. The final outcome of a transaction is determined after the token arrives once more at the transaction's site of execution.

We implemented a "forward-oriented" validation scheme where validations are performed against running transactions and where only update transactions have to validate [27]. Since all conflicting transactions are not yet committed, a conflict can be resolved either by aborting the validating transaction (abort policy) or by restarting the running transactions (kill policy). In the simulations, we adopted a hybrid scheme for conflict resolution. Transactions start using the abort policy until their number of aborts has reached a certain restart limit, RL, which is a simulation parameter. The executions following then apply the kill strategy against conflicting transactions. If two "killer transactions" conflict with each other, the transaction with the lower priority, as determined by the number of restarts, is aborted. Starvation is thus avoided, since a frequently restarted transaction will eventually hold the highest priority. In our simulation system, a different restart limit can be selected for every transaction type. If RL = 0 is chosen, transactions start immediately with the kill policy.

For coherency control, a broadcast invalidation scheme with a MBT in any node is employed. The scheme has been enhanced by several features described in [58]. Blocking of pages that belong to the write set of a validating transaction, until it is known whether the transaction has been successful, is one such feature.

4. SIMULATION MODEL

We invested major effort in implementing a detailed simulation system for message-based DB-sharing complexes. The system is structured in a modular way such that different algorithms and realization strategies for the main components can easily be incorporated. The primary objective for developing such a system was to identify critical performance factors in DB-sharing systems and to quantify the performance impacts of different realization strategies. More specifically, we are interested in comparing the performance of the concurrency and coherency control algorithms described above and in assessing the effectiveness of the various optimizations applied therein. Scalability is also of major interest (i.e., how performance is affected if we change the number of nodes). Another aspect covered in our studies and

described in this paper is the impact of different strategies for load distribution and the influence of hot spot pages on performance.

Though we could have used synthetic workloads for our simulations, we chose to apply a *trace-driven approach*. This was motivated by experiences with performance evaluations of centralized DBMS [29], indicating the importance of a realistic workload model for assessing the merits or shortcomings of different algorithms, specifically for concurrency control or buffer management. Traces of real-life OLTP applications provide load profiles which typically consist of many individual transactions of different types with nonuniform reference pattern, hot spot objects, and locality of reference. It is difficult to capture these important aspects adequately by synthetic workloads. Nonuniform reference patterns are especially important for the evaluation of buffer management schemes, and thus for DB-sharing where buffer invalidations and coherency control are expected to play a major role. In fact, trace-driven simulation is applied in most performance studies of caching schemes [1, 19, 40, 51, 75]. Knowledge of reference distribution can also be used for a meaningful, affinity-based, load distribution.

A valid criticism of trace-driven simulations is that the results apply primarily only to the specific application from which the data has been collected. On the other hand, one could equally argue that synthetic workloads do not represent *any* application well. Also, the generality of an empirical performance study can be improved by using traces from different environments. A main reason that trace-driven simulation is less frequently used in database performance studies lies in the difficulty of obtaining traces of commercial OLTP applications.

The next section describes traces, for which simulation results are presented. In Section 4.2 we describe the structure and realization of our simulation system, together with the parameters.

4.1 Trace Characteristics

Our simulation system does not use the original traces as input, but a more compact representation, called *reference string*, containing only the relevant record types from the trace. Four different record types are essential for our purposes: (1) a begin of transaction (BOT); (2) an EOT record for every transaction; (3) a FIX and (4) UNFIX record for every page reference. A page reference is actually represented by the FIX record, while the UNFIX record merely indicates to the buffer manager that the page need no longer be "fixed" in the buffer on behalf of the respective transaction, but may be considered for replacement. The BOT record indicates the transaction type and the access mode (update or read-only). A page reference specifies the transaction and page identifiers, the page type (e.g., regular database page or administration data for free space management, etc.) and the access mode (read or write).

Simulation runs were conducted for six different transaction loads originating from real applications with a nonrelational DBMS. The largest reference string contains over one million page references and 17,500 transactions. However, simulation execution times turned out to be extremely long for

reference strings of this size, so that most runs had to be conducted for smaller loads. Since the main findings of our extensive performance study [60] could already be observed for these loads, in Section 5 we present the simulation results for two shorter transaction mixes (Table I). Both loads originate from OLTP environments with interactive processing only. Mix 1 consists of very short transactions with a comparatively high share of update accesses. Over 90% of the page accesses are performed by two of the four transaction types that operate on disjoint database partitions. For a DB-sharing system with two nodes, a high degree of node-specific locality of reference can therefore be expected when the two major transaction types are processed on different systems.

Mix 2 is about four times as long as Mix 1 in terms of page accesses, and is dominated by read references. Though most transactions are also short in Mix 2, 70% of the page references are due to 13% of the transactions that perform more than 100 page accesses. These long (read-only) transactions are also responsible for a high degree of locality of reference for Mix 2 (on average, a page is referenced 10 times by the same transaction). The reference matrix in Figure 2 depicts the access distribution of the 10 transaction types against 9 database areas (files). The matrix shows that there is a dominant transaction type (TT1) that encompasses about 65% of all page accesses. Similarly, 58% of all accesses are directed to a single file (area 1). In order to utilize all nodes, it is not possible to assign the dominating transaction type to a single node for configurations with more than two nodes. Assigning a transaction type to multiple nodes reduces the amount of node-specific locality of reference, and can therefore deteriorate performance.

In both workloads there are several hot spot pages that contain database address translation tables (DBTT) and data for free space administration (FPA). DBTT and FPA pages have a much higher access frequency than "normal" database pages. Particularly critical are FPA pages that are always accessed with the intent to update (these pages are used by insert operations to determine a database page with sufficient free space). So in Mix 2 there is a single FPA page that is accessed by 28% of all transactions; in Mix 1 there are two such pages accessed by 17 to 20% of the transactions with intent to update. Page-level locking with long write locks on these pages would result in disastrous performance. In the underlying DBMS, lock conflicts on these pages are largely avoided by locking only the respective table entry (DBTT) or holding only short locks/latches (FPA).

In our simulations the cost for transaction processing is modeled by requesting a certain number of instructions for every "unit of processing" (UP) which is either a page reference, a BOT, or an EOT. The values for "#instructions per UP" (Table I) are based on path length measurements and differ from load to load. For Mix 1 the value is considerably higher than for Mix 2, since the average number of page accesses per database operation was smaller, causing a higher overhead for process switching. In addition, the higher share of update operations resulted in an increased pathlength per page request. Note that the overhead for I/O and communication is not included in the UP cost, but is modeled separately (see below).

Table I. Workload Characteristics

	Mix 1	Mix 2
# transactions	2288	669
share of update transactions	45.7%	46.5%
# transaction types	4	10
DB size	565 MB	330 MB
# page references	9862	40751
# pages referenced	2188	3025
share of write accesses	48.9%	6.7%
# units of processing (UP)	14438	42089
# instruction per UP	8100	2850
# UPs per transaction (avg.)	6	62

Transaction type	Area (partition) 1	2	3	4	5	6	7	8	9	Total
TT1	17960	5975	1324	918	118	211	15	18	5	26544
TT2	2707	7	619	167	1616	76	130	102	1	5425
TT3	1567	32	1765	299	2	137	8			3811
TT4	1444		727	263	521	45	207			3207
TT5	34		51	1132						1217
TT6				363						363
TT7	16		1	21	14	5	49			106
TT8						52				52
TT9			18							18
TT10	1						7			8
Total	23729	6014	4505	3163	2271	526	417	120	6	40751

Fig. 2. Reference matrix for Mix 2.

4.2 Structure and Realization of the Simulation System

The simulation system has been implemented in PL/1 and employs discrete event simulation. It models DB-sharing systems with an arbitrary number of nodes, and considers CPU, I/O, and communication costs. The gross structure of the simulation system is shown in Figure 3.

—The scheduler is the central component requesting services from the other four modules. It manages the CPUs and models transaction processing for the entire DB-sharing system.

—The reference manager manages the reference string and delivers transactions and their reference records to the scheduler. It also performs transaction routing controlled by a routing table, as could be done by a front-end system in a real DB-sharing complex.

—The protocols for concurrency and coherency control as well as buffer management and (local) logging have been implemented within the respective components. Buffer management and logging is based on the DB-cache

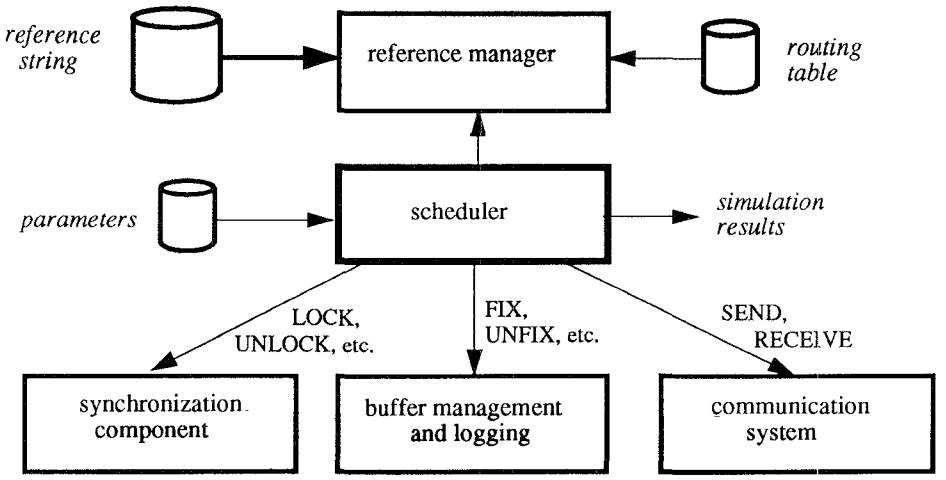

Fig. 3. Gross structure of the simulation system.

approach [20] which employs a NOFORCE strategy and uses a sequential log file to speed logging. For logging, a separate log buffer is maintained for each node to hold the afterimages of modified pages. A log buffer is written out in one sequential I/O when it is full or at commit time of an update transaction. Global LRU is employed for page replacement.

—The communication system handles the message transfers between the nodes. We modeled a point-to-point connection between any two nodes and an additional bus for the schemes employing broadcast invalidation. Batching of messages is possible, and is controlled by two parameters, the blocking factor B and the maximal buffering delay BMAX (only relevant if $B > 1$).

Tables II–V show the main parameters of our simulation system together with their default settings. Most of the parameters had to be fixed in order to limit the number of simulation runs. Parameters varied for each mix include the concurrency/coherency control protocol, the number of nodes, and the multiprogramming level. For most of the other parameters, the effect of different values has also been studied, but was found to be less relevant for the *relative* performance of the different schemes. Still, in Section 5 we also analyze the influence of page-level locking on FPA/DBTT pages and the effect of different communication costs and routing strategies.

CPU, communication and I/O costs. The scheduler simulates a single CPU server for every node and distinguishes between three types of CPU requests with different priorities and different costs (#instructions). CPU requests for communication (send or receive operation, message processing) have highest priority, followed by CPU requests for I/O (disk read or write, log I/O). The remaining CPU requests are for transaction processing and have lowest priority; these requests are issued for every BOT, page

Table II. General Parameters

Parameter	*Settings*
reference string	Mix 1, Mix2
number of nodes N	1, 2, 3, 4
multiprogramming level P	4, 8, 16
CPU capacity per processor	3 MIPS
# instructions per UP	see Table 1
routing table	dependent on load and N

Table III. Concurrency Control Parameters

Parameter	*Settings*
concurrency/coherency control protocol	CLM, PCL, CV-OCC, TR-OCC
FPA/DBTT locking	no
consistency level	2
PCA distribution (PCL)	dependent on load and N
token delay (TR-OCC)	0
restart limit RL (TR-OCC)	2
MAX-WAIT (timeout)	1 - 30 s

Table IV. Communication Parameters

Parameter	*Settings*
transmission rate	
point-to-point	3 MB/s
bus	3 MB/s
bundling factor	1
max. bundling delay BMAX	-
message length	100 B (+ page size)
# instructions per send	5000
# instructions per receive	5000
# instructions for processing one message	1000

Table V. Parameters for Buffer Management and I/O

Parameter	*Settings*
buffer size	600 page frames
page size	2 KB
log buffer size	16 page frames
I/O time	30 - 60 ms (equally distributed)
log buffer write time	9 - 20 ms (dependent on number of pages)
# instructions per I/O	2500

reference, or EOT. The average number of instructions per request type is specified by parameters. The actual number of instructions per request is exponentially distributed over the specified mean.

Communication costs are represented by CPU overhead for sending, receiving, and processing messages, as well as communication delays for their transmission over the network. Every point-to-point connection, as well as the bus, have been modeled as separate servers in order to capture possible bottleneck situations—for instance, due to page transfers. The net transmission time is calculated from the message length and bandwidth parameters.

I/O costs are represented by CPU overhead and I/O delay for every I/O operation. Disk servers have not been explicitly modeled, assuming that bottleneck situations can be prevented by a sufficiently large number of disk drives or by using a disk array [54].

Modeling of transaction processing. For each of the N nodes, a fixed multiprogramming level P is applied indicting the number of concurrently active transactions. The total degree of parallelism is thus $N*P$. The execution of a transaction is modeled by processing all its records from the reference string in chronological order. The processing of a reference record, in turn, depends on the concurrency and coherency control protocol and the current system state. So different actions are needed, depending on whether or not a lock conflict occurred or a page was found in the local buffer. In general, multiple events like CPU, I/O or communication requests are involved until a reference record is processed. The execution of an EOT record triggers commit processing, consisting of logging and protocol-specific steps like validation or release of locks. After completion of a transaction, the scheduler requests the next transaction from the reference manager. The simulation stops as soon as there are no more transactions to be executed for any of the nodes (according to the routing strategy, see below).

Transaction routing. We employed a static strategy for load distribution by using a predetermined routing table that remains unchanged during a simulation run. The routing table specifies, for every node, which transaction types it may process. Random routing can easily be achieved by letting every node execute transactions of any type.

For calculation of the routing tables, it was necessary at first to determine the workload's reference matrix from the trace, indicating for every transaction type the relative frequency and distribution of database accesses. The reference matrix and the number of nodes then served as input parameters for an iterative heuristic that was developed for the primary copy protocol, which determines the routing table as well as the PCA distribution in a coordinated way. In each step of this heuristic, a transaction type, or some part of it, is assigned to one node such that the node is not overloaded. This assignment starts with the largest transaction type and is continued until the entire workload is allocated. The PCA allocation is adapted in each step of the assignment procedure such that the load distributed so far can be processed with a minimum of internode communication. The routing tables

obtained by this procedure were also applied to the other protocols in order to facilitate a comparison of the schemes. These routing tables also help to achieve node-specific locality, which is generally useful.

Although we have perfect knowledge of the reference behavior of the transactions, it would not have been realistic to utilize this fact fully to determine an "optimal" routing policy. So we only considered the reference information at the type level, rather than for individual transactions. In order to support load balancing, we used a simple approach by requiring that about the same number of page accesses ($\pm 5\%$) should be assigned to every node. For more than two nodes, the dominating transaction types in our loads had therefore to be assigned to multiple nodes.

Specific concurrency control aspects. Our simulation system offers a choice between consistency level 2 and serializability (level 3)—an option that is also provided by some commercial DBMS (DB2, Tandem NonStop SQL a.o.). The simulation results (presented in Section 5) were achieved for *consistency level* 2, which was also used in the applications reflected in the traces. Consistency level 2 improves performance by tolerating the possibility of "unrepeatable reads"; that is, a transaction may see different versions of a database object [26]. For locking schemes, consistency level 2 means that read locks are generally released before EOT ("short" read locks), giving rise to a reduced conflict probability. There will, however, be more lock requests, since a transaction may now request multiple read locks for the same object. OCC can also utilize a restriction to consistency level 2. As pointed out in [60], in this case only update transactions have to validate against other update transactions (to avoid lost updates), while read-only transactions are always successful. Note that access to uncommitted objects is always avoided with OCC, since all modifications are performed on private object copies.

For the locking schemes, *deadlocks* are handled by a hybrid strategy. Deadlocks between local transactions are explicitly detected and resolved by aborting the transaction causing the deadlock. Global deadlocks are resolved by a simple timeout mechanism (parameter MAX-WAIT).

For the optimistic token ring scheme (TR-OCC), the token is usually sent to the next node as soon as all local validations have been performed. The parameter "token delay" can be used to delay the token transmission in order to control the communication overhead. For the CLM and central validation (CV-OCC) schemes, the central concurrency controller is located on a separate node with the same CPU capacity as the transaction processing nodes, giving a total of $N + 1$ nodes for $N > 1$.

Another important factor is concurrency control on hot spot pages. As pointed out above, in our traces so-called FPA/DBTT pages represent hot spot pages for which page-level concurrency control is expected to result in unacceptable performance. We could not implement record (entry)-level locking for these pages, since our traces only specify the page identifiers and page type (FPA/DBTT or regular pages), but not the accessed entries and records in a page. Instead, we either completely ignore lock conflicts on FPA/DBTT pages or perform page-level locking. The first option was chosen in most

simulation runs, assuming that a sufficiently low level of lock contention can be achieved by specialized locking techniques. To illustrate the effect of page-level locking, we also ran experiments where we assumed no lock conflicts among local transactions only, but applied page-level locking between transactions of different nodes. The latter option only assumes a special treatment of FPA/DBTT pages within a node and also provides coherency control for FPA/DBTT pages.

Performance measures. The simulation system determines throughput and response time as the main performance measures. Throughput is not expressed by "transactions per second," since transactions of different types differ significantly in size. Instead, we use the number of UPs per second (*UPS*) as the throughput measure. The UPS value is calculated from the number of processed UPs of successful transactions divided by the total processing time. To explain the throughput and response time results, a large number of detailed statistics is produced by every simulation run, providing extensive information on resource utilization, response time composition, buffer behavior (hit ratio, buffer invalidations, page requests) and concurrency control aspects (frequency of aborts, lock waits, external lock requests). Some of the results are reported in the next section.

5. SIMULATION RESULTS

The main part of this section is devoted to comparing the simulation results for the four concurrency and coherency control protocols described in Section 3. In Sections 5.2 and 5.4, we discuss some additional experiments illustrating how performance is influenced by routing strategy, varying communication costs, and page-level locking on hot spot pages. The results refer to the two transaction loads introduced in Section 4 and the parameter settings from Tables II–V.

5.1 Performance Comparison of the Implemented Protocols

Figures 4 and 5 show the throughput results for Mix 1 and Mix 2, respectively, for our four protocols. The results are given for one to four nodes (N) and three different values of P (parallelism per node). In general, throughput increases as the multiprogramming level P grows due to the increased CPU utilization; deviations from this behavior are caused by increased data contention and/or communication overhead. Response time results correspond to the throughput values according to Little's result [45]. Response times deteriorate as the degree of multiprogramming grows due to increased CPU and data contention. Figures 4 and 5 show that in most cases the *primary copy locking protocol achieved the best throughput results*, followed in second place by the central validation scheme, CV-OCC, with preclaiming for failed transactions. Significantly lower performance was observed for the two other protocols, the token ring scheme, TR-OCC, as well as the CLM scheme. In the following, simulation results are discussed separately for each protocol.

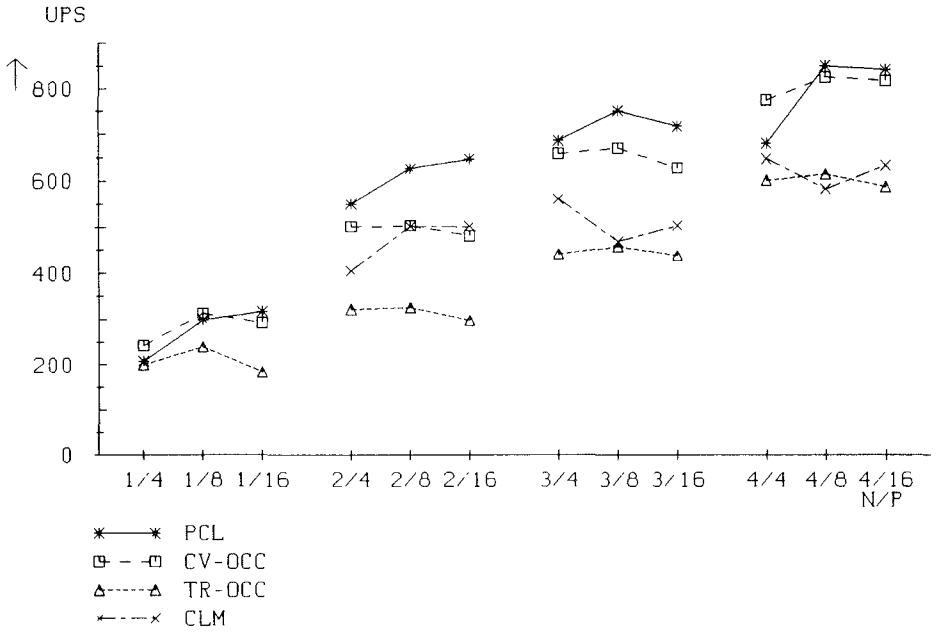

Fig. 4. Throughput results for Mix 1.

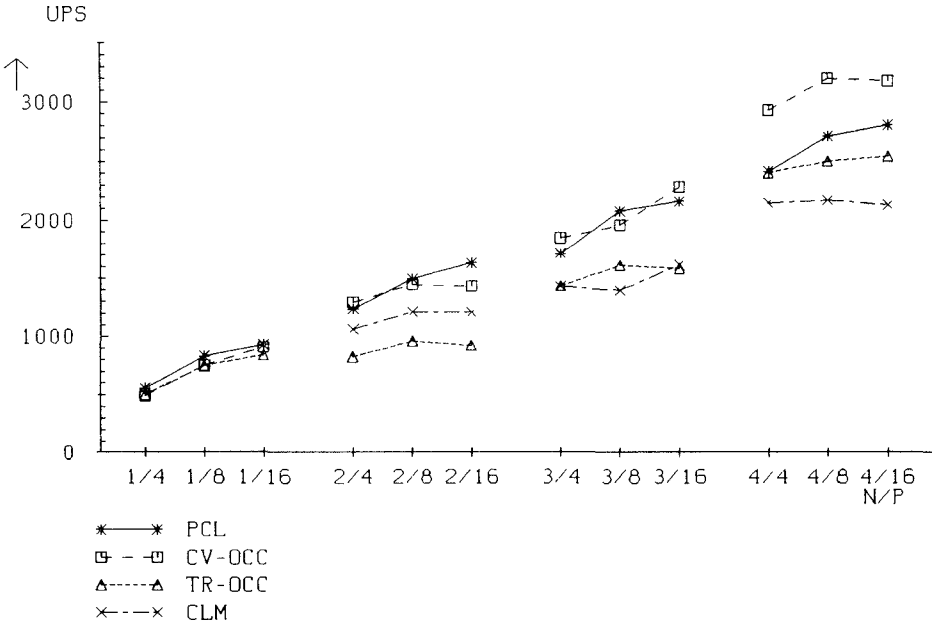

Fig. 5. Throughput results for Mix 2.

Results for primary copy locking (PCL). Simulation results for Mix 1 are influenced by the fact that this load is dominated by two transaction types that mainly operate on disjoint database portions. Thus for $N = 2$, an almost optimal load and PCA allocation was feasible, so that nearly all ($> 97\%$) lock requests could be granted locally. This helped to achieve optimal performance results for two nodes (more than twice the throughput and better response times than in the centralized case). The superlinear throughput increase for two nodes was possible because the aggregate buffer size was doubled compared to the central case, resulting in a reduced I/O delay and overhead. This permitted a higher effective CPU utilization per node, despite the (small) communication overhead for remote lock requests (see below). However, for more than two nodes, throughput could not be significantly improved further due to the fact that each of the dominating transaction types had to be processed by more than one node. This led to reduced node-specific locality and a higher number of global lock requests (for N = 4, "only" 76% of the locks could be locally granted). Lock contention also increased considerably for three and four nodes, so that the average CPU utilization decreased with growing N despite the increasing communication overhead.

For Mix 2, lock conflicts were not a throughput-limiting factor due to the high proportion of read accesses. Here throughput could be improved almost linearly, even for three or four nodes, in spite of the fact that in these cases the dominating transaction type was processed at more than one node. Figure 6 shows the composition of the average CPU utilization for Mix 2, which helps to explain the throughput results. Total utilization is composed of the effective CPU utilization ("transaction processing") and the overhead for I/O and communication. The effective CPU utilization, which directly determines throughput, was (mostly) lower for DB-sharing than for the central case, owing to the comparatively high communication overhead for Mix 2. The communication overhead grows with the multiprogramming level P, since more transactions can then issue lock requests concurrently. Communication overhead also grows as the number of nodes is increased because fewer lock requests may then be granted locally (see below). The negative effect of communication overhead may be partially compensated for by the fact that I/O overhead is substantially lower for DB-sharing than in the central case, due to the increased aggregate buffer size. For $P = 4$, the reduced I/O delays in the DB-sharing configurations even permitted a higher effective CPU utilization than in the central case for Mix 2.

Communication overhead is mostly determined by the number of remote lock requests. Figure 7 illustrates the average number of messages per lock request for Mix 1 and Mix 2. The curves labeled "4" correspond to the results shown in Figures 4 and 5, and were achieved with both improvements mentioned in Section 3.2 (read optimization, coordinated PCA and load allocation). The graphs labeled "3" show the average number of messages that would have occurred if only the coordinated PCA and load distribution had been applied, but not read optimization. The curves labeled "1" and "2" refer to the simple CLM and simple PCL schemes (see Section 3) requiring 2 and $(2 - 2/N)$ messages per lock request, respectively, which brings our

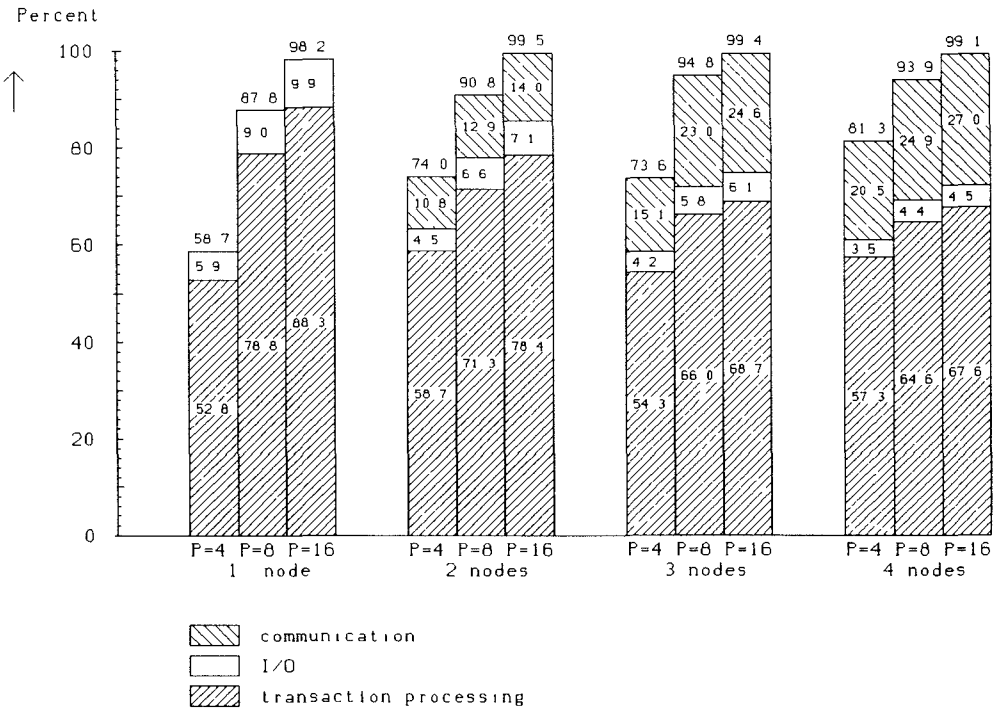

Fig. 6. Average CPU utilization for Mix 2 (PCL).

optimizations into proper perspective. The number of messages per lock request refer to the average for all multiprogramming levels. The graphs show that most messages could be saved, due to coordination between load and PCA distribution, though the effectiveness of this optimization decreases with more nodes (similar to the simple PCL scheme). This is because the average partition size decreases with growing N if the database size is kept constant, and because we had to split the dominating transaction types across multiple sites (the PCA allocation can support a local synchronization for a transaction type on only one node). Such an increase in message frequency may however be avoided for "scalable" and "delightful" applications [78] (e.g., debit–credit) where the database grows proportionally with the transaction rate (number of nodes) and the same degree of node-specific locality can be sustained. Although these prerequisites are not given for our loads, most lock requests could be granted locally. The coordinated load and PCA allocation was particularly effective for Mix 1. Even with four nodes, this optimization alone allowed us to grant 68% (compared to 25% with the simple PCL scheme) of the locks locally for Mix 1—which translates into less than one global lock request per transaction for this load.

Read optimization was also very effective, and is the main reason for good performance results for Mix 2. As can be seen from Figure 7, read optimization is of greater help to more nodes and when fewer locks are granted due to

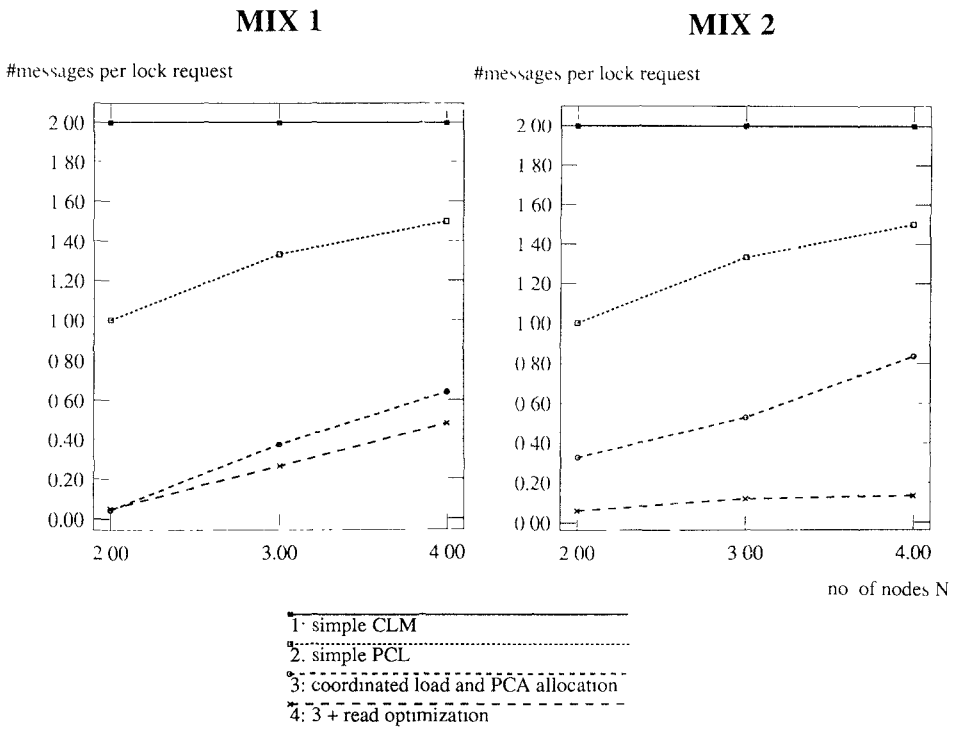

Fig. 7. Average number of messages per lock request.

local PCA ownership. This is because read authorizations are assigned only for objects not belonging to the local partition and because with more nodes more objects of external partitions have to be referenced. Our results confirm that read optimization is capable of improving scalability of the primary copy scheme by reducing dependencies on how far load and PCA distribution can be effectively coordinated.

This was, of course, particularly useful for Mix 2, due to the high share of read accesses, and helped that more than 93% of the locks could be acquired and released without communication even for four nodes. This high share was, however, favored by the use of short read locks (consistency level 2), since the long transactions in Mix 2 frequently requested multiple read locks for the same page. With read optimization, communication was typically required only for the first read lock request for a page, while subsequent requests could take advantage of a local read authorization. With short read locks and *without* read optimization, the number of global lock requests per transaction would have increased from 3.9 to 24 for Mix 2 and $N = 4$. We thus consider read optimization indispensable for consistency level 2, though its usefulness is not restricted to intratransaction locality of read references. Revocations of read authorizations were very rare and did not cause any noticeable performance degradation.

Table VI. Average Hit Ratios and Number of Physical Reads per Transaction

	Mix 1				Mix 2			
	N=1	N=2	N=3	N=4	N=1	N=2	N=3	N=4
average hit ratio (%)	67.4	79.8	83.5	83.7	88.0	91.7	92.6	95.1
average number of disk reads per tx	1.4	0.9	0.7	0.7	7.7	5.2	4.7	3.1

The *I/O behavior* was another important factor that determined the performance results. As mentioned, the increased aggregate buffer size allowed, in some DB-sharing configurations, for a superlinear throughput increase and better response times, compared to the centralized case ($N = 1$). This is confirmed by Table VI, showing that DB-sharing configurations achieved significantly better hit ratios than for $N = 1$. The savings in I/O delay were generally more significant that the communication delays for external lock requests, thus supporting good response times. High response times for DB-sharing were mostly due to increased lock contention.

Of course, hit ratios for the central case can also be improved by using a larger database buffer. Thus we observed a hit ratio of 92.5% for Mix 2 in the central case, when we doubled the buffer size, and 95.7% when we increased the buffer size by a factor of four. These hit ratios are only slightly better than the corresponding hit ratios for DB-sharing and the same aggregate buffer size ($N = 2$ and $N = 4$), despite the fact that some pages are replicated in multiple buffers for DB-sharing. Good hit ratios for DB-sharing are due mainly to the affinity-based routing strategy that supports node-specific locality of reference by coordinating PCA and load distribution. The direct exchange of modified pages between nodes also helped to improve I/O behavior for DB-sharing. We found that considerably more pages were received from the PCA node (together with the lock response message) than buffer invalidations occurred. This helped save disk I/O, compared to the centralized case where a buffer miss always causes a disk read. In addition, delay for getting a page from another node is significantly shorter than for a disk access (factor 50 for our parameters).

Efficient coherency control without any additional messages was a main advantage of PCL over the schemes using broadcast invalidation. Apart from avoidance of broadcast messages at the end of update transactions, it was especially important that no additional requests for pages modified at other nodes were necessary. Furthermore, buffer invalidations are only possible for pages for which no read authorization is held and which belong to the partition of another node. This resulted in a very low frequency of buffer invalidations, with a maximum of 2% of all lock requests (Mix 1, $N = 4$). Page transmissions did not cause any bottleneck situations in the communication system; its utilization was always less than 2%.

The coordinated load and PCA allocation was, of course, mainly responsible for the low amount of buffer invalidations and page transmissions (most pages were referenced and modified at the PCA node). In general, buffer invalidations and page transmissions are the more frequent the higher the share of update accesses and the lower the amount of node-specific locality.

Simulation results for CLM scheme. Figures 4 and 5 reveal that in all simulation runs the central lock manager approach was clearly inferior to the PCL protocol, in spite of the fact that an additional node was used for concurrency control (of course, for $N = 1$, the CLM and PCL results are identical). Even for two nodes, throughput could not be substantially improved compared to the centralized case.

Although the sole interest concept, as well as read optimization, helped to reduce the number of global lock requests, we found that by far fewer locks could be locally synchronized due to sole interest than by a local PCA in the primary copy scheme. One reason is that in order to get a write authorization (sole interest), communication with the CLM is necessary, while PCA ownerships are not requested from other nodes but are assigned a priori. Furthermore, sole interest is unstable and frequently revoked if transactions of the same type (or of other types referencing the same database portions) are running on different sites. So, generally, more than twice the number of global lock requests was observed for the CLM scheme than for PCL. Additional delays and messages were caused by sole interest revocations, in particular for Mix 1, where up to 35% of these authorizations had to be released involuntarily.

Read optimization was also important for the CLM scheme, in view of the use of short read locks, but worked less smoothly than in the primary copy protocol due to conversions between read and write authorizations. Also, in the PCL scheme, a read authorization can always be assigned immediately, unless the PCA lock manager has learned of a write request. In the CLM scheme, however, sole interest assignments delay the assignment of read authorizations, even if in the node holding sole interest no lock requests or only read lock requests have been issued.

The comparatively high frequency of synchronization messages caused a high utilization of the CLM node. For $N = 4/P = 16$, its CPU utilization was over 80%, indicating that the CLM node may easily become a throughput bottleneck.

The broadcast invalidation scheme for coherency control contributed to a lesser degree to the unacceptable performance of the CLM scheme. Only for Mix 1, where many update operations occur, and for more than two nodes was there noticeable communication overhead and delays to the broadcast messages and page requests. So even an on-request invalidation scheme to detect buffer invalidations does not seem likely to substantially improve the overall performance of the CLM scheme.

Results for central validation scheme with preclaiming. After PCL, this optimistic scheme achieved the best performance results—in some cases, particularly for Mix 2 and $N = 4$, even better throughput than with PCL. In this comparison, however, note that in the CV-OCC scheme an additional node was used for concurrency control and also for sending the broadcast messages for successfully validated transactions. This helped to reduce communication overhead on the N transaction processing nodes. In contrast to

the CLM scheme, the central synchronization node was never highly utilized (< 30%), despite higher transaction rates.

A basic observation is that the *optimistic protocol scaled up better than PCL*, mainly because of the two following reasons. First, communication for concurrency control is limited to one validation request per update transaction (and one additional request for every reexecution). Thus message frequency for concurrency control—but not for coherency control—was less dependent on the number of nodes than the locking schemes that depend on node-specific locality of reference. So for N > 2, communication overhead on transaction processing nodes was smaller than with the primary copy protocol. The second key factor was more efficient processing of reader transactions with OCC than with the locking protocols, due to the relaxation to consistency level 2.

In the PCL and CLM locking schemes, read-only transactions suffered from communication delays until a short read lock was granted—though read optimization reduced the number of global read lock requests considerably—as well as from lock conflicts with update transactions. With OCC and consistency level 2, however, read-only transactions are freed from validation, thus never aborted, and do not cause any validation overhead. Instead, reader transactions can always get the current version of a page, since modifications are performed on private page copies (pages are only blocked for a short time when they are to be replaced by a new version during the write phase of a local update transaction [60]). Thus communication for read-only transaction was restricted to requesting pages from other nodes having performed the most recent modification of the respective pages. The number of these page requests was only significant for Mix 1, and increased with the number of nodes. For read-dominated Mix 2, on the other hand, CV-OCC outperformed PCL for four nodes. For more than four nodes, CV-OCC would have been even more superior to PCL for Mix 2, due to the increasing difficulty of finding an adequate PCA and load allocation.

With the OCC scheme, on the other hand, update transactions were less efficiently processed by far than with PCL. With the central validation scheme, update transactions failed frequently in their first validation (for $N = 4$ more than 50%) and had to perform a preclaiming before the second execution. In the simulation, this preclaiming helped insure that the second execution was always successful (the same pages were referenced than in the first execution). Thus, starvation was avoided, but at the expense of a delayed reexecution due to "lock" conflicts with other failed update transactions in their preclaiming phase. Typically, the second execution was faster than the first, since most of the pages to be accessed still resided in the database buffer. With a lesser degree of *access invariance* [21]; that is, if different pages were referenced during reexecution of failed transactions, the number of restarts and I/Os would be higher, thus degrading performance.

Simulation results for token ring protocol. As may be seen from Figures 4 and 5, throughput for the optimistic token ring scheme was always significantly worse than for PCL or the central validation scheme. The unacceptable

Table VII. Average CPU Utilization for Communication (TR-OCC, P = 8)

	Mix1		Mix2	
	N=2	N=4	N=2	N=4
broadcast messages	4.0 %	6.8 %	1.4 %	3.5 %
page requests	0.7 %	7.4 %	0.7 %	4.3 %
token (+ cvalidation requests)	33.3 %	13.8 %	42.3 %	18.9 %
Total	38.0 %	28.0 %	44.4 %	26.7 %

results are largely due to the fact that update transactions were aborted even more often than in the central OCC scheme, where preclaiming at least guaranteed a successful second execution. Here, however, even transactions that reached their restart limit (2) were still aborted by concurrently validating "killer" transactions (as described in Section 3.4).

Another critical factor was the communication overhead for validation and coherency control (Table VII) The table shows that the communication overhead associated with coherency control (broadcast messages, page requests) increases with the number of nodes and already accounts for a considerable portion of CPU utilization (14.2% for Mix 1 and $N = 4/P = 8$). More important, however, was the communication overhead for validation—determined by the average token-holding time (circulation time). Table VII shows that without token delay, in addition to the time required for validations, a very high communication overhead is introduced which decreases with more nodes due to increased circulation times. So the highest communication overhead was observed for two nodes, accounting for the low throughput figures in this case.

In order to reduce communication overhead, many simulation runs with varying settings for the parameter "token delay" have been conducted. Although this allowed for improved performance results, particularly for Mix 2, the results of PCL or the central OCC scheme could not be reached. The increased token-waiting times had a negative effect on throughput for smaller multiprogramming levels only (e.g., $P = 4$). Even response times often gained more from reduced CPU waiting times, due to smaller communication overhead, than they suffered from the increased token-waiting times (at least for short token delays). Thus token-holding times must be carefully controlled in order to limit the communication overhead without causing overly long waiting times for (update) transactions ready to validate.

5.2 Performance Impact of the Routing Strategy

For the primary copy protocol, which showed the best performance results, additional experiments have been conducted and are analyzed in this and the next two sections. The results of these experiments are presented for the larger workload, Mix 2.

In the simulation runs discussed above, we always applied a coordinated load and PCA allocation which aimed at supporting node-specific locality.

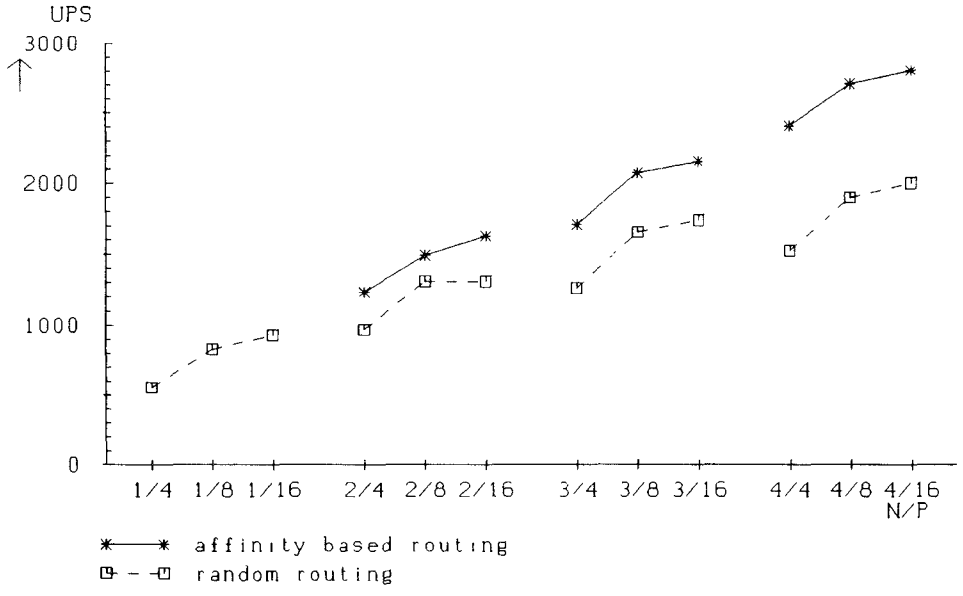

Fig. 8. Throughput for Mix 2 and PCL under different routing strategies.

These results are now compared to those achieved with a random routing of the transactions. In the runs with a random routing, the database was partitioned in a way such that each node controlled about equally important database portions, that is, each node had to process about the same number of lock requests.

Figure 8 shows the throughput results for Mix 2 with such a random routing, as well as the results obtained with a coordinated load and PCA distribution (repeated from Figure 5). Observe that throughput is always worse with random routing and that the differences from the results with the affinity-based load distribution increase significantly as the number of nodes grows. Compared to the centralized case, random routing allowed only modest throughput improvements, by a factor 1.41 (instead of 1.76 for affinity-based routing) with two nodes, by a factor of merely 2.16 (3.02) with four nodes (for $P = 16$). The same trends apply for response times, which were always worse for DB-sharing with random routing than in the centralized case, despite the increased buffer capacity.

We found that random routing had a negative effect on the frequency of global lock requests, on hit ratios, on data contention (fewer local but many more global lock conflicts), and on the number of buffer invalidations and page transmissions. Most important for Mix 2 were the number of physical reads and global lock requests, which were on average about twice as high as for the affinity-based load distribution. Table VIII summarizes the fractions of local and global lock requests for random as well as for affinity-based routing.

Table VIII. PCL Lock Behavior under Random Routing (and Affinity-Based Routing)

	N=2	N=3	N=4
locally granted locks due to local PCA (%)	48.2 (83.6)	33.1 (73.5)	24.6 (58.2)
locally granted due to read optimization (%)	45.3 (13.4)	57.0 (20.5)	63.4 (35.1)
global lock requests (%)	6.5 (3.0)	9.9 (6.0)	12.0 (6.7)
number of global lock requests per tx	3.7 (1.7)	5.7 (3.4)	6.9 (3.9)

The table shows that with random routing far fewer lock requests could be granted locally due to PCA ownership than with affinity-based load distribution; this share is about the same as with the simple PCL scheme (3.2), allowing a local synchronization for $100/N\%$ of the locks. For Mix 2, however, this low share could be greatly improved by read optimization, which turned out to be even more helpful here. This confirms even more strongly that the read optimization is able to reduce the dependencies of the PCL scheme on node-specific locality (the effectiveness of coordinating load and PCA distribution). So only a comparatively small share of the locks had to be acquired remotely ($\leq 12\%$), although the absolute number of global lock requests per transaction is significantly higher than with affinity-based routing. In the case of four nodes, up to 40% of the CPU capacity was required for communication overhead with random routing, compared to "only" 27% with affinity-based routing and its higher throughput.

5.3 Influence of Communication Costs

Though the primary copy scheme generally caused very few global lock messages, the communication overhead was considerable, mainly because of the choice of CPU capacity (3 MIPS) and the comparatively expensive communication primitives, causing 22,000 instructions per global lock request/ response (5000 instructions per send or receive operation and another 2000 instructions for processing the two messages). In order to study the performance effects of reduced communication costs, we conducted simulation runs with 500 instead of 5000 instructions per send or receive operation, resulting in a total of 4000 instructions per global lock request/ response.

Figure 9 shows the throughput results for Mix 2 (and affinity-based routing) obtained with communication costs of 500 and 5000 instructions per send or receive operation. The cheaper communication primitives allow for significant throughput improvements, which grow with the number of nodes, as well as with the multiprogramming level. This allowed for an almost N-fold throughput, with N nodes compared to the centralized case (or even a superlinear throughput improvement for lower multiprogramming levels). For $N = 4/P = 16$, throughput was 24% higher than with the expensive communication operations, and only 8% instead of 27% of the CPU capacity was needed for communication overhead. The cheaper send and receive operations also improved response times by up to 20%.

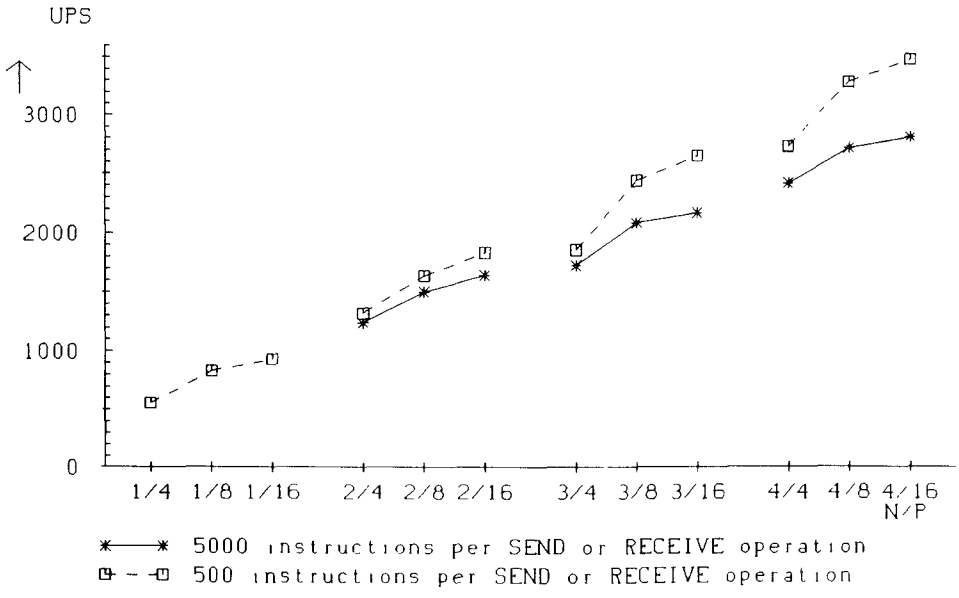

Fig. 9 Influence of communication costs on throughput (Mix 2, PCL).

In order to reduce communication overhead, not only should the number of messages be small, but it is equally important that the underlying hardware and operating system provide efficient communication primitives. Faster processors only reduce the absolute time for message processing, but do not improve the relative costs for communication. *Message bundling* is also no general solution. In experiments with different bundling factors and delays, we found that the reduction of communication overhead pays off only when CPUs are highly utilized. Delaying remote requests when sufficient CPU capacity for an immediate send is available only increases the message delay and often decreases throughput. For message batching to be effective, a more flexible approach that automatically adapts the bundling parameters according to the available CPU capacity would therefore be required. Such an approach, however, is difficult to implement and control. Message bundling is more effective when more messages are directed to the same destination, namely a central lock manager. This makes message bundling less attractive for distributed protocols designed for few remote requests, as in PCL.

5.4 Influence of Page-Level Locking on Hot Spots

As discussed in Section 4, our workloads reference several hot spot pages containing free space information and database translation tables. The simulation results presented so far refer to the case where lock conflicts on FPA/DBTT pages have been ignored, assuming a similarly effective synchronization scheme as possible for the central case (i.e., the use of latches or record-level locking). To illustrate the consequences of a page-oriented

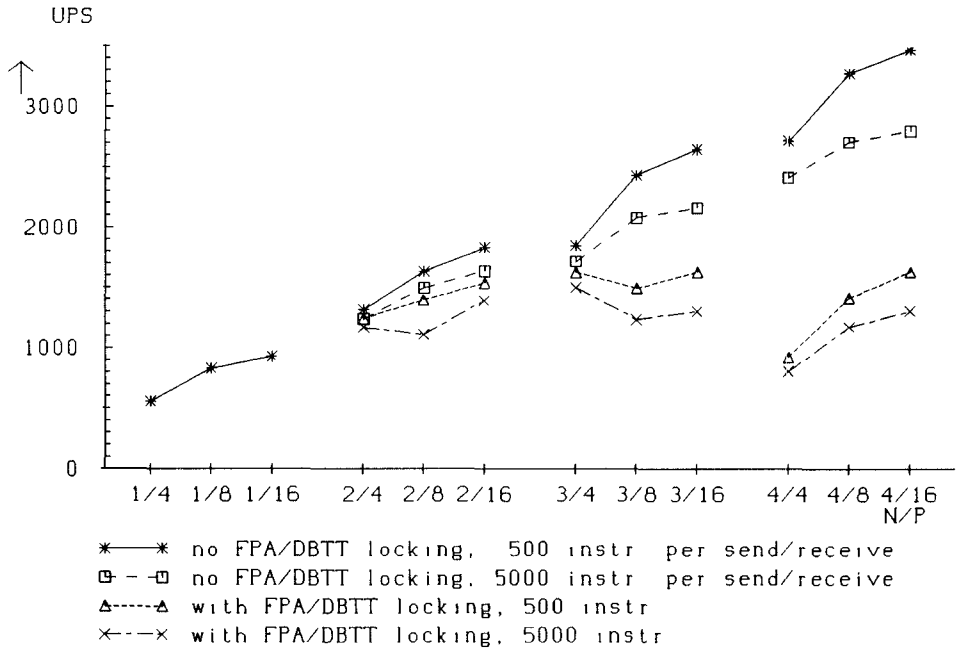

Fig. 10. Influence of FPA/DBTT locking on throughput (Mix 2, PCL).

protocol for concurrency/coherency control on such pages, we also performed simulation runs with locking on FPA/DBTT pages. However, since there was no problem in using latches or record-level locking within a node, we used page-level locking on FPA/ DBTT pages for synchronizing accesses between transactions of different nodes only. Lock contention between local transactions on FPA/ DBTT pages was still assumed to be negligible.

Figure 10 shows the throughput results for Mix 2 with and without such a locking on FPA/DBTT pages, as well as for the two different communication costs from Section 5.3. The graphs show that throughput is dramatically worse in the case of page locking on FPA/DBTT pages, in particular for $N = 3$ and $N = 4$. For two nodes, affinity-based transaction routing could achieve a situation where FPA/DBTT pages were mostly referenced by transactions of one node. Consequently, FPA/DBTT lock conflicts between different nodes were comparatively rare, and affected throughput to a lesser degree. However, adding a third and fourth node actually resulted in a decline of throughput compared to $N = 2$, because the hot spot pages were then accessed concurrently on different nodes, causing a high lock contention, which prevented the utilization of the added capacity. Typically, a majority of transactions were blocked because of lock conflicts on a few FPA/DBTT pages. Long waiting lines formed for these pages, resulting in long deactivations until the requested locks could be granted. The waiting times were also prolonged by message delays for assigning and releasing locks between different nodes and exchanging the modified pages. In addition, the timeout

mechanism to resolve global deadlocks proved to be unsuitable in the case of FPA/ DBTT locking, since the long waiting times caused many transactions to be aborted unnecessarily. For four nodes, lock contention and transaction aborts limited effective CPU utilization to only 31%, on average. The actual CPU utilization differed significantly from node to node since different degrees of lock contention were experienced.

As Figure 10 shows, even the use of cheap communication primitives was of comparatively little help, since lock contention was the performance bottleneck. This indicates that a low communication overhead, either because of a small message frequency and/or cheap communication primitives, enables acceptable or good performance only if lock contention is sufficiently low. Page-level locking on frequently modified pages, even if restricted to inter-node concurrency control, was not sufficient to keep lock contention at an acceptable level for our workloads. Thus, special synchronization protocols on these page types are required for DB-sharing, too. Realization of such protocols has to be tailored to the specific use of data, which may depend on the underlying DBMS.

6. RELATED PERFORMANCE STUDIES FOR DB-SHARING

The simulation study in [6] also compares the performance of four concurrency control schemes for DB-sharing. The protocols considered are a disk controller locking, a simple CLM scheme (without a sole interest concept or read optimization) and two basic primary copy protocols with synchronous or asynchronous lock requests; optimistic protocols have not been studied. The major technique to reduce communication overhead was batching of messages. For the primary copy schemes, the share of local lock requests was provided as a parameter (mostly 10%).

The main weakness of the study is that coherency control has not been investigated. Instead, a simple buffer purge scheme was assumed where no pages are retained in the database buffers, but all pages are purged out at EOT in order to avoid their invalidation. This scheme is even worse than FORCE where only modified pages are written out of the buffer. In addition to the write overhead and delays, the purge scheme also leads to a drastic increase in the number of physical reads, and must therefore be considered as unacceptable for real database systems. Simulations were driven by synthetic workloads and showed the best performance for disk controller locking. This was because there were almost no buffer hits in the simulated system, so that locking could be combined with the disk accesses without extra overhead. Similar throughput results as for disk controller locking were predicted for the simple CLM scheme and asynchronous primary copy locking if they were to apply high batching degrees.

The performance of DB-sharing with a CLM and message batching was also compared to DB-partitioning systems. Despite the simple approaches for concurrency control and buffer management (buffer purge), significantly better throughput results were generally predicted for DB-sharing than for DB-partitioning with a DB-cache approach for buffer management. This was

probably influenced by the workload model, which allowed only for small hit ratios ($\leq 8\%$) and negligible lock contention. Also, apparently, no CPU overhead was considered for I/O operations, thus favoring the buffer purge scheme. There was no mention of whether or not I/O delays for purging (writing) modified pages out of the buffer (FORCE) were modeled for the DB-sharing configurations. If these I/O delays are ignored, the results for DB-sharing are invalid.

A combination of analytical modeling and simulation was used in [86] to investigate the performance of a pass-the-buck protocol (used in IMS Data Sharing) and DB-sharing configurations using a central "lock engine" for concurrency control. The study assumed a FORCE strategy for update propagation to disk and a broadcast invalidation scheme for coherency control. The performance evaluation further assumed a completely homogeneous workload, with all transactions and nodes experiencing the same amount of lock contention and I/O frequency. I/O frequency was further assumed to be independent of the number of sites. Database objects were not explicitly modeled in the DB-sharing simulations, but the average lock conflict probabilities and waiting times were calculated analytically. The lock engine scheme (assumed to perform concurrency control without communication overhead) was found to allow for more nodes to be effectively coupled than the pass-the-buck scheme. The authors conclude that lock contention is the critical factor that determines the maximal achievable transaction rate. Performance studies in [17] and [36] show that on-request invalidation outperforms broadcast invalidation for coherency control, since it avoids extra messages for detecting buffer invalidations.

In [31] the performance of a largely improved pass-the-buck scheme was evaluated by means of empirical simulations. The original scheme was enhanced by an integrated coherency control for NOFORCE. So-called retention locks are used for all pages retained in the buffers after EOT in order to avoid their invalidation. These locks also helped reduce the frequency of global lock requests, similar to read and write authorizations in the CLM scheme. As in the optimistic token ring protocol, studied in this paper, the token (buck) holding time was a critical factor for the pass-the-buck scheme. Short token holding times result in high communication overhead and CPU contention, while longer token delays increase the waiting times for global lock requests, and thus response times and lock contention.

We discuss additional performance studies on DB-sharing in [65].

7. CONCLUSIONS AND OUTLOOK

DB-sharing represents a locally distributed architecture for high-volume transaction processing. In contrast to DB-partitioning systems, there is no need to physically partition the database because all processing nodes can directly access all (shared) disks. This property results in an increased flexibility for dynamic load balancing and for dealing with variations in the number of nodes. Since finding an acceptable database partitioning, and

adapting it to changing demands, is a major problem in "shared-nothing" systems, administration is also simplified. Technical problems to be solved for DB-sharing include concurrency control, coherency control, workload allocation, logging, and recovery. Critical to the performance of a DB-sharing system is the protocol used for concurrency and coherency control since these functions mainly determine the amount of communication for transaction processing. Both tasks should be solved in an integrated way in order to limit the number of extra messages.

We presented a performance evaluation of four concurrency and coherency control protocols for DB-sharing. Unlike other studies, we applied a trace-driven simulation approach and assumed a NOFORCE strategy for update propagation to disk, together with buffer-to-buffer communication to exchange modified pages directly between different nodes. Two optimistic protocols and two locking schemes were examined for concurrency control, with the latter relying on various improvements to reduce the communication frequency by utilizing locality of reference. On-request invalidation or broadcast invalidation schemes were applied for coherency control.

The primary copy-locking (PCL) protocol generally showed the best performance (from the implemented protocols). Key factors for the good results were a coordinated load and PCA distribution with affinity-based transaction routing, the employment of a so-called read optimization, and an efficient coherency control. Affinity-based transaction routing does not only help to limit the communication overhead for concurrency control, but also reduces the number of physical reads, buffer invalidations, page transfers, and global lock conflicts. Read optimization limits the dependency of locking schemes on the number of achievable node-specific localities of reference. Reduction in the number of global lock requests is greater if locality of read references is higher and the share of update accesses is smaller. The coherency control scheme applied to PCL requires neither additional messages for the detection of buffer invalidations (on-request invalidation) nor for the exchange of modified pages between different nodes. Affinity-based transaction routing and buffer-to-buffer communication contributed to improved I/O behavior for DB-sharing compared to the centralized case. Thus the response time impact of communication delays and increased lock contention could at least partially be compensated for.

The central lock manager (CLM) scheme required significantly more global lock requests that PCL, and showed unacceptable performance results. This was mainly due to the weaknesses of the sole interest concept, which helped to save far fewer messages than the use of PCAs in the primary copy scheme. Batching of messages reduces the communication overhead, but at the expense of increased response times and lock contention. Also, the CLM is a potential bottleneck for growing transaction rates and requires special recovery provisions.

The optimistic schemes generally allowed for the lowest communication overhead for concurrency control (validation), and were less dependent on the amount of node-specific locality of reference than the locking schemes. However, they achieved good performance results only for loads with a high share

of read-only transactions, because update transactions suffered from an intolerably high number of restarts, particularly in the token-ring scheme. Also, optimistic schemes cannot use on-request invalidation for coherency control, but are bound to the more expensive broadcast invalidation alternative. The best optimistic protocol was a central validation scheme that employed a "preclaiming" for failed transactions, where it generally helps that their second execution ends successfully.

DB-sharing with primary copy locking has similarities to DB-partitioning systems. There is a partitioning of the database in both approaches which determines, together with the load distribution, the frequency of internode communication. However, PCL uses only logical partitioning, which can be more easily and automatically adapted than can a physical data allocation, say, on an hourly basis, when significant changes in the load profile are observed. With DB-partitioning, a transaction is processed for the most part where the referenced data resides, independent of which node a transaction is assigned to. With DB-sharing and PCL, however, a transaction can be largely executed where it has been routed to, while only global lock requests have to be processed by the responsible nodes. Thus, PCL preserves a significantly higher potential for load balancing compared to DB-partioning. A third advantage of PCL is that the same objects can be read and processed concurrently at different sites. Read optimization, in many cases, allows a local synchronization of read accesses for objects belonging to another node's partition, and reduces dependencies on how far transactions can be routed to the node controlling most of the data they need. This optimization promises a better scalability for DB-sharing and PCL than with DB-partitioning, at least for loads with a higher share of read accesses.

It is interesting to note that our primary copy scheme could also be applied in a *distributed main memory database system*. In this case, each node would hold its entire partition in main memory and would also cache pages from remote partitions. As outlined in Section 3.2, page and lock requests could be combined, as well as release of write locks and propagation of the modified pages to the primary copy node. A coordinated load and PCA allocation and read optimization for locally buffered pages of remote partitions would reduce the number of global lock requests and page transmissions in this environment as well.

There are several areas in the analysis of DB-sharing systems that deserve further investigation. First, as our simulation results have shown, there is a need to support concurrency and coherency control below the page level for some types of database objects, namely free space administration, index structures, or application-specific high traffic objects. Proposals for dealing with such objects [28, 47, 49, 63] still need to be implemented and evaluated quantitatively. Second, adaptive load distribution strategies which take the current load and system state into account for transaction assignment should be investigated. Third, parallel query processing strategies tailored for DB-sharing should be studied and compared to query processing in DB-partitioning systems. Finally, the use of shared, and possibly nonvolatile, semiconductor stores [64] needs further evaluation.

ACKNOWLEDGMENTS

Many discussions with Theo Härder helped in the design and evaluation of the algorithms. The detailed comments and recommendations by Gio Wiederhold and the referees are also gratefully acknowledged. The simulation system was implemented by M. Luzcak, G. Petry, and P. Scheug with the help of U. Lanzer and V. Bohn.

REFERENCES

1. AGARWAL, A., HENNESSY, J , AND HOROWITZ, M. Cache performance and operating system and multiprogramming workloads *ACM Trans. Comput Syst 6*, 4 (1988), 393–431.
2. *AIM / SRCF Functions and Facilities.* Facom OS Tech. Manual 78SP4900E, Fujitsu, 1986
3. ANON ET AL. A measure of transaction processing power. *Datamation* (April 1985), 112–118.
4 BELLEW, M., HSU, M., AND TAM, V. Update propagation in distributed memory hierarchies. In *Proceedings of the IEEE 6th International Conference on Data Engineering*, (1990), 521–528.
5. BERNSTEIN, P. A., HADZILACOS, V , AND GOODMAN, N. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.
6. BHIDE, A. An analysis of three transaction processing architectures. In *Proceedings of the 14th International Conference on Very Large Data Bases* (1988), 339–350.
7. BLACK, D. L., GUPTA, A., AND WEBER, W. Competitive management of distributed shared memory. In *Proceedings of the IEEE Spring CompCon* (1989), 184–190.
8. BOHN, V., HÄRDER, T., AND RAHM, E. Extended memory support for high performance transaction systems. In *Proceedings of the 6th (German) Conference on Measurement, Modelling and Evaluation of Computer Systems* (Munich, Sept. 1991), Informatik-Fachberichte, vol. 286, Springer-Verlag, 92–108.
9. BORAL, H., ET AL. Prototyping Bubba: A highly parallel database system. *IEEE Trans Knowl. Data Eng. 2*, 1 (1990), 4–24.
10. BORR, A. Transaction monitoring in Encompass: A non shared-memory multi-processor approach. In *Proceedings of the 7th International Conference on Very Large Data Bases* (1981), 155–165.
11. BURKES, D. L., AND TREIBER, R. K. Design approaches for real-time transaction processing remote site recovery In *Proceedings of the IEEE Spring CompCon* (1990), 568–572.
12. CAREY, M. J., FRANKLIN, M. J., LIVNY, M., AND SHEKITA, E. J. Data caching tradeoffs in client-server DBMS architectures. In *Proceedings of the ACM SIGMOD Conference* (Boulder, Colo., May 1991), 357–366.
13. CAREY, M. J., AND LU, H. Load balancing in a locally distributed database system. In *Proceedings of the ACM SIGMOD Conference* (1986) 108–119.
14. CHENG, J. M., LOOSLEY, C R. SHIBAMIYA, A., AND WORTHINGTON, P S. IBM Database 2 performance: design, implementation, and tuning. *IBM Syst. J. 23*, 2 (1984), 189–210.
15. DAN, A., DIAS, D. M., AND YU, P. S. The effect of skewed data access on buffer hits and data contention in a data sharing environment. In *Proceedings of the 16th International Conference on Very Large Data Bases* (Brisbane, Aug. 1990), 419–431.
16. DEWITT, D. J., ET AL. The Gamma database machine project. *IEEE Trans Knowl. Data Eng. 2*, 1 (1990), 44–62.
17. DIAS, D. M., IYER, B. R., ROBINSON, J. T., AND YU, P. S. Integrated concurrency-coherency controls for multisystem data sharing. *IEEE Trans. Softw. Eng. 15*, 4 (1989), 437–448.
18. DUBOIS, M., SCHEURICH, C., AND BRIGGS, F. A. Synchronization, coherence, and event ordering in multiprocessors. *IEEE Computer 21*, 2 (Feb. 1988), 9–21.
19. EFFELSBERG, W., AND HÄRDER, T. Principles of database buffer management. *ACM Trans. Database Syst. 9*, 4 (1984), 560–595.

20. ELHARDT, K., AND BAYER, R. A database cache for high performance and fast restart in database systems. *ACM Trans. Database Syst. 9*, 4 (1984), 503–525.
21. FRANASZEK, P. A., ROBINSON, J. T., AND THOMASIAN, A. Access invariance and its use in high contention environments. In *Proceedings of the 6th IEEE International Conference on Data Engineering* (1990), 47–55.
22. GARCIA-MOLINA, H., AND ABBOTT, R. K. Reliable distributed database management. *Proc. IEEE 75*, 5 (1987), 601–620.
23. GOLDSTEIN, A. C. The design and implementation of a distributed file system. *Digital Tech. J. 5* (Sept. 1987), 45–55.
24. GRAY, J., ET AL. One thousand transactions per second. In *Proceedings of the IEEE Spring CompCon* (1985), 96–101.
25. GRAY, J., ED. *The Benchmark Handbook for Database and Transaction Processing Systems.* Morgan Kaufmann, 1991.
26. GRAY, J. N., LORIE, R. A. PUTZOLU, G. R., AND TRAIGER, I. Granularity of locks and degrees of consistency in a shared data base. In *Proceedings of the IFIP Working Conference on Modelling in Data Base Management Systems*, North-Holland, 1976, 365–394.
27. HÄRDER, T. Observations on optimistic concurrency control. *Inf. Syst. 9*, 2 (1984), 111–120.
28. HÄRDER, T. Handling hot spot data in DB-sharing systems. *Inf. Syst. 13*, 2 (1988), 155–166.
29. HARDER, T., PEINL, P., AND REUTER, A. Performance analysis of synchronization and recovery schemes. *IEEE Database Eng. 8*, 2 (1985), 50–57.
30. HÄRDER, T., PEINL, P., AND REUTER, A. Optimistic concurrency control in a shared database environment. Tech. Rep., Computer Science Dept., Univ. of Kaiserslautern, 1985.
31. HÄRDER, T., AND RAHM, E. Quantitative analysis of a synchronization protocol for DB-sharing. In *Proceedings 3. GI/NTG Conference on Measurement, Modelling and Evaluation of Computer Systems*, Informatik-Fachberichte, vol. 110, Springer-Verlag, 1985, 186–201 (in German).
32. HÄRDER, T., AND RAHM, E. Multiprocessor database systems for high performance transaction systems. *Informationstechnik 28*, 4 (1986), 214–225 (in German).
33. HÄRDER, T., AND REUTER, A. Principles of transaction-oriented database recovery. *ACM Comput. Surv. 15*, 4 (1983), 287–317.
34. HÄRDER, T., SCHÖNING, H., AND SIKELER, A. Evaluation of hardware architectures for parallel execution of complex database operations. In *Proceedings of the 3rd Annual Parallel Processing Symposium* (1989), 564–578.
35. HELLAND, P., ET AL. Group commit timers and high volume transaction systems. In *Proceedings of the 2nd International Workshop on High Performance Transaction Systems* (Asilomar, 1987). Also in *Lecture Notes in Computer Science, vol. 359*, Springer-Verlag, 1989, 301–328.
36. HSU, Y.-P. Performance evaluation of data sharing transaction processing systems. Master's Thesis, Dept. of Electrical and Computer Eng., Univ. of Mass. at Amherst, 1988.
37. IYER, B. R., YU, P. S., AND DONATIELLO, L. Analysis of fault-tolerant multiprocessor architectures for lock engine design. *Comput. Syst. Sci. Eng. 2*, 2 (1987), 59–75.
38. JOSHI, A. M. Adaptive locking strategies in a multi-node data sharing environment. In *Proceedings of the 17th International Conference on Very Large Data Bases* (Barcelona, Sept. 1991), 181–191.
39. JOSHI, A. M., AND RODWELL, K. E. A relational database management system for production applications. *Digital Tech. J., 8* (Feb. 1989), 99–109.
40. KENT, C. A. Cache coherence in distributed systems. Res. Rep. 87/4, DEC Western Research Lab., 1987.
41. KING. R. P., HALIM, N., GARCIA-MOLINA, H., AND POLYZOIS, C. A. Management of a remote backup copy for disaster recovery. *ACM Trans. Database Syst. 16*, 2 (June 1991), 338–368.
42. KRONENBERG, N. P., LEVY, H. M., AND STRECKER, W. D. VAX clusters: A closely coupled distributed system. *ACM Trans. Comput. Syst. 4*, 2 (1986), 130–146.
43. KUNG, H. T., AND ROBINSON, J. T. On optimistic methods for concurrency control. *ACM Trans. Database Syst. 6*, 2 (1981), 213–226.

44. LI, K., AND HUDAK, P. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst 7*, 4 (1989), 321–359.
45. LITTLE, J. D. A proof of the queuing formulate L = λW. *Oper. Res. 9* (1961), 383–387.
46. LYON, J. Tandem's remote data facility. In *Proceedings of the IEEE Spring CompCon* (1990), 562–567.
47 MOHAN, C., AND NARANG, I. Recovery and coherency-control protocols for fast intersystem page transfer and fine-granularity locking in a shared disks transaction environment. In *Proceedings of the 17th International Conference on Very Large Data Bases* (Barcelona, Sept 1991), 193–207.
48. MOHAN, C., NARANG, I., AND PALMER, J. A case study of problems in migrating to distributed computing: data base recovery using multiple logs in the shared disks environment. IBM Res. Rep. RJ 7343, San Jose, Calif., 1990.
49. MOHAN, C., NARANG, I., AND SILEN, S. Solutions to hot spot problems in a shared disks transaction environment. In *Proceedings of the 4th International Workshop on High Performance Transaction Systems* (Asilomar, 1991).
50. NECHES, P. M. The anatomy of a database computer—revisited. In *Proceedings of the IEEE CompCon Spring Conference* (1986), 374–377.
51 NELSON, M. N., WELCH, B. B , AND OUSTERHOUT, J. K. Caching in the Sprite network file system. *ACM Trans. Comput. Syst. 6*, 1 (1988), 134–154.
52. NITZBERG, B., AND LO, V. Distributed shared memory: a survey of issues and algorithms. *IEEE Computer* (Aug. 1991), 52–60.
53. *Oracle for massively parallel systems—technology overview.* Oracle Corp., part 50577-0490, 1990.
54. PATTERSON, D. A., GIBSON, G., AND KATZ, R. H. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the ACM SIGMOD Conference* (1988), 109–116.
55. PIRAHESH, H., ET AL. Parallelism in relational data base systems: Architectural issues and design approaches. In *Proceedings of the 2nd International Symposium on Databases in Parallel and Distributed Systems* (Dublin, 1990).
56. QUENON, M., AND VORIS, D. IMS VS data sharing guidelines. IBM Tech. Bull. G320-0590, 1987.
57. RAHM, E. Primary copy synchronization for DB-sharing. *Inf. Syst 11*, 4 (1986), 275–286.
58. RAHM, E. Integrated solutions to concurrency control and buffer invalidation in database sharing systems. In *Proceedings of the 2nd IEEE International Conference on Computers and Applications* (1987), 410–417.
59. RAHM, E. Design of optimistic methods for concurrency control in database sharing systems. In *Proceedings of the 7th IEEE International Conference on Distributed Computing Systems* (1987), 154–161
60 RAHM, E Concurrency control in multiprocessor database systems—Concepts, realization strategies and quantitative evaluation. Doctoral dissertation (in German), Informatik-Fachberichte, vol. 186, Springer-Verlag, 1988.
61. RAHM, E., ET AL. Goal-oriented workload management in locally distributed transaction systems. IBM Res. Rep. RC 14712, Yorktown Heights, N.Y., 1989.
62. RAHM, E. A framework for workload allocation in distributed transaction systems. *J. Syst. Softw. 18*, 3 (1992), 171–190.
63. RAHM, E Recovery concepts for data sharing systems Computer Science Dept., Univ. of Kaiserslautern, Tech. Rep. 14/89, 1989. A shorter version of this paper appeared in *Proceedings of the 21st International Symposium on Fault-Tolerant Computing* (Montreal, June 1991), IEEE Computer Society Press, 368–375.
64. RAHM, E. Use of global extended memory for distributed transaction processing. In *Proceedings of the 4th International Workshop on High Performance Transaction Systems* (Asilomar, 1991).
65. RAHM, E. Concurrency and coherency control in database sharing systems. Computer Science Dept., Univ. of Kaiserslautern, Tech. Rep. 3/91, 1991.
66. RENGARAJAN, T. K., SPIRO, P. M., AND WRIGHT, W. A. High availability mechanisms of VAX DBMS software. *Digital Tech. J. 8* (Feb. 1989), 88–98.

67. REUTER, A. Load control and load balancing in a shared database management system. In *Proceedings of the 2nd IEEE International Conference on Data Engineering* (1986), 188–197.
68. REUTER, A., AND SHOENS, K. Synchronization in a data sharing environment. IBM San Jose Research Lab., Tech. Rep., 1984.
69. ROBINSON, J. T. A fast general-purpose hardware synchronization mechanism. In *Proceedings of the ACM SIGMOD Conference* (1985), 122–130.
70. SACCA, D., AND WIEDERHOLD, G. Database partitioning in a cluster of processors. *ACM Trans. Database Syst. 10*, 1 (1985), 29–56.
71. SCRUTCHIN JR., T. W. TPF: Performance, capacity, availability. In *Proceedings of the IEEE Spring CompCon* (1987), 158–160.
72. SEKINO, A., ET AL. The DSC—a new approach to multisystem data sharing. In *Proceedings of the National Computer Conference* (1984), 59–68.
73. SHOENS, K., ET AL. The AMOEBA project. In *Proceedings IEEE Spring CompCon* (1985), 102–105.
74. SHOENS, K. Data sharing vs. partitioning for capacity and availability. *IEEE Database Eng. 9*, 1 (1986), 10–16.
75. SMITH, A. J. Disk cache—Miss ratio analysis and design considerations. *ACM Trans. Comput. Syst. 3*, 3 (1985), 161–203.
76. SON, S. H. Synchronization of replicated data in distributed systems. *Inf. Syst. 12*, 2 (1987), 191–202.
77. STONEBRAKER, M. Concurrency control and consistency of multiple copies in distributed INGRES. *IEEE Trans. Softw. Eng. 5*, 3 (1979), 188–194.
78. STONEBRAKER, M. The case for shared nothing. *IEEE Database Eng. 9*, 1 (1986), 4–9.
79. STRICKLAND, J. P., UHROWCZIK, P. P., AND WATTS, V. L. IMS/VS: An evolving system. *IBM Syst. J. 21*, 4 (1982), 490–510.
80. *The Tandem Database Group*. NonStop SQL, a distributed, high-performance, high-availability implementation of SQL. In *Proceedings of the 2nd International Workshop on High Performance Transaction Systems* (Asilomar, 1987).
81. *TPF2 General Information*. IBM Manual, GH20-7540, 1988.
82. TRAIGER, I. Trends in systems aspects of database management. In *Proceedings of the 2nd International Conference on Databases (ICOD-2)*, (1983), 1–20.
83. WANG, Y., AND ROWE, L. A. Cache consistency and concurrency control in a client/server DBMS architecture. In *Proceedings ACM SIGMOD Conference* (Boulder, Colo., May 1991), 367–376.
84. WEST, J. C., ISMAN, M. A., AND HANNAFORD, S. G. PERPOS fault-tolerant transaction processing. In *Proceedings 3rd IEEE Symposium on Reliability in Distributed Software and Database Systems* (1983), 189–194.
85. YEN, W. C., YEN, D. W. L., AND FU, K. Data coherence problem in a multicache system. *IEEE Trans. Comput. 34*, 1 (1985), 56–65.
86. YU, P. S., ET AL. On coupling multi-systems through data sharing. *Proc. IEEE 75*, 5 (1987), 573–587.

Received June 1988; revised June 1990, December 1991, March 1992; accepted April 1992