Evaluation of Closely Coupled Systems for High Performance Database Processing

Erhard Rahm

University of Kaiserslautern, Department of Computer Science,
P.O. Box 3049, Kaiserslautern, Germany, E-mail: rahm@informatik.uni-kl.de

Abstract

Closely coupled systems aim at a more efficient communication and cooperation between processing nodes compared to loosely coupled systems. This can be achieved by using globally shared semiconductor memory to speed up the exchange of messages or to store global data structures. For distributed database processing, the database sharing (shared disk) architecture can benefit most from such a close coupling. We present a detailed simulation study of closely coupled database sharing systems. A shared store called Global Extended Memory (GEM) is used for system-wide concurrency and coherency control, and to improve I/O performance. The performance of such an architecture is evaluated and compared with loosely coupled database sharing systems employing the primary copy approach for concurrency and coherency control. In particular, we study the impact of different update strategies (FORCE vs. NOFORCE) and workload allocation schemes (random vs. affinity-based routing). The use of shared disk caches implementing a global database buffer is also considered. Simulation results will be presented for synthetically generated debit-credit workloads and a real-life workload represented by a database trace.

Keywords: Databases, Shared Disk, Close Coupling, Extended Memory, Disk cache, Performance

1. Introduction

Growing demands for high performance, scalability, and high availability require distributed architectures for transaction and database processing. For this purpose, three general architectures termed "shared memory" (or "shared everything"), database sharing ("shared disk") and database partitioning ("shared nothing") are considered as most appropriate [St86, Bh88, DG92]. "Shared memory" refers to the use of multiprocessors for database processing. In this case, we have a *tightly coupled system* where all processors share a common main memory as well as peripheral devices (terminals, disks). While the shared memory supports efficient cooperation and synchronization between processors, it can also become a performance bottleneck thereby limiting the scalability of the architecture [DG92]. Furthermore, there are significant availability problems since the shared memory reduces failure isolation between processors, and since there is only a single copy of system software like the operating system or the DBMS [Ki84]. These limitations can be overcome by database sharing and database partitioning systems which are typically based on a loose coupling of processors. In *loosely coupled systems*, each processor is autonomous, i.e., it runs a separate copy of the operating system, the DBMS and other software, and there is no shared memory [Ki84]. Inter-processor communication takes place by means of message passing. Loose coupling can be used for interconnecting uniprocessors or multiprocessors. We use the term processing node (or node) to refer to either a uniprocessor or a multiprocessor as part of a loosely coupled system.

Database partitioning and database sharing differ in the way the external storage devices (usually disks) are allocated to the processing nodes. In *database partitioning* or "shared nothing" systems [St86, ÖV91, DG92], external storage devices and thus the data are partitioned among all nodes. A node can directly access only data of the local database partition; if remote data needs to be accessed a distributed transaction execution becomes necessary. For this purpose, the DBMS has to support construction of distributed query execution plans as well as a distributed commit protocol. In *database sharing* or "shared disk" systems [Ra86, Yu87, MN91], on the other hand, each node can directly access all disks holding the common database. As a result, no distributed transaction execution is necessary as for database partitioning. Inter-node communication is required for concurrency control in order to synchronize the node's accesses to the shared database. Furthermore, coherency control is needed since database pages are buffered in the main memory of every node. These page copies remain cached beyond the end of the accessing transaction making the pages susceptible to invalidation by other nodes. Both system architectures, database partitioning and database sharing, are supported by a number of commercially available DBMS (e.g., Tandem's and Teradata's relational DBMSs follow the shared nothing approach; IBM's IMS, DEC's RDB and CODASYL DBMSs, Oracle, Ingres, Fujitsu and Unisys support shared disk environments).

The performance of loosely coupled database partitioning and database sharing systems is largely influenced by the typically high cost of message passing. With the proliferation of high-speed networks message transfer times improved substantially, but the CPU overhead for executing the communication protocols remained very high in most general-purpose systems. *Closely coupled systems* aim at a more efficient communication and cooperation between nodes compared to loosely coupled systems. This can be achieved by using globally shared semiconductor memory to speed up the exchange of messages or to store global data structures [Ra91a]. To avoid the overhead of process switching, access times of the global memory should be short enough to allow a synchronous access, i.e., without releasing the CPU. This is only feasible for locally distributed (clustered) systems where all processors and the global memory reside in close proximity. The global memory should not be instruction-addressable like main memory to obtain better failure isolation than for shared memory in multiprocessors. Failure isolation is also improved compared to tightly coupled systems because the nodes connected to the global memory are autono-

0-8186-3770-6/93 $3.00

mous as for loose coupling, i.e., they have their own main memory and software copies. This also enhances scalability since the global memory is merely used for certain global functions.

In [KLS86], DEC's VaxClusters have been named a closely coupled system. According to our terminology however, the VaxClusters represent a loosely coupled shared-disk architecture since inter-node communication is based on message passing over an interconnection network. On the other hand, the shared disk controllers may maintain a disk cache which then represents a page-addressable global memory. This memory can be used for implementing a global database buffer or to exchange pages between different nodes. Shared disk caches cannot synchronously be accessed since access times are in the order of 1-5 ms per page [Sm85, Ra92a]. The shared-disk system TPF [Sc87] which is widely used in high-volume transaction processing systems also supports the use of shared disk caches. Furthermore, the shared disk controllers in TPF systems implement a simple lock protocol to avoid inter-node communication for global concurrency control [BDS79]. A similar approach is the use of special-purpose processors for global functions, e.g., a "lock engine" for global concurrency control in database sharing systems [Yu87]. Such an architecture can also be considered as a closely coupled approach, in particular if special machine instructions are provided to perform the global services so that inter-node communication and process switching are avoided.

In [Ra91a], we have considered the use of a special non-volatile semiconductor store called GEM (Global Extended Memory) for database processing in closely coupled systems. It turned out that database sharing systems can benefit most from such a shared store since it can be used to improve the performance of global concurrency and coherency control. Furthermore, I/O performance can be improved by keeping database or log files resident in GEM or by caching pages at an intermediate storage level between main memory and disk. In this paper, we present a performance evaluation of different GEM usage forms in database sharing systems. In particular, the performance of a concurrency control scheme using a global lock table in GEM is evaluated and compared with a distributed locking protocol in a loosely coupled database sharing system. Furthermore, we study the impact of a close coupling on workload allocation by comparing random with affinity-based transaction routing. With respect to coherency control, we consider different update propagation schemes between main memory and external storage (FORCE vs. NOFORCE). These important aspects were not covered in previous studies like [DIRY89, DDY91] where the intermediate memory was solely used as a global database buffer. Another unique feature of our study is the use of both synthetically generated workloads and database traces for performance evaluation.

In the next section, we briefly summarize the characteristics of GEM and its usage forms for database sharing. In section 3, we describe our simulation model. Section 4 presents the experiments conducted and analyses the simulation results. Finally, we discuss related work and provide the conclusions of this investigation.

2. Close coupling with Global Extended Memory

Global Extended Memory (GEM) is assumed to be a non-volatile semiconductor memory accessible by all nodes in the system (Fig. 2.1). Non-volatility may be achieved either by using a battery backup or uninterruptible power supply. GEM shares many characteristics of extended memory for centralized (mainframe) systems, e.g., IBM's Expanded Storage [CKB89, Ra91a]. In particular, a page-oriented interface with special machine instructions to transfer pages between GEM and main memory is supported. Furthermore, GEM accesses are synchronous with page access times in the order of 10-50 microseconds. GEM is a largely passive storage unit that is managed by system software (e.g., the operating system or the DBMS) of the accessing systems. A consequence of this approach is that all page transfers between extended memory and the disk subsystem must go through main memory (Fig. 2.1).

In the central case, extended memory is used to improve I/O performance. This is possible because extended memory supports much better access times and I/O rates than magnetic disks, albeit at a substantially higher storage cost. Our previous study of extended storage architectures in centralized systems [Ra92a] showed that the best I/O performance is obtained if non-volatile extended memory is used to keep entire database or log files resident in semiconductor memory. This is because all disk accesses are avoided for the respective files in this case. Non-volatile extended memory can also be used as a write buffer for disks. This approach only requires a small amount of non-volatile semiconductor memory to substantially speed up page writes. A modified page is written to the write buffer at first, while the disk copy is updated asynchronously, i.e., without increasing response time. A third usage form of extended memory is for caching database pages at an intermediate storage level to reduce the number of disk reads.

These usage forms are also supported by GEM to enhance I/O performance in the distributed case. To additionally improve inter-node communication and cooperation, GEM supports an extended interface with a second access granularity called *entry*. Entries are smaller than pages and can be used to implement global data structures in GEM. The entry size may be defined in multiples of a unit size, e.g., a double word. Hence, the main instructions for GEM usage are read and write operations to pages and entries. In addition, a Compare&Swap operation is assumed to be available for the unit entry size in order to synchronize concurrent GEM accesses. Note that despite the fact that we can modify individual double words with the Compare&Swap instruction, GEM still cannot directly be modified like main memory. Rather all page and entry modifications need to be performed in main memory before they are written back to GEM. A further advantage of the sketched access interface is its upward compatibility with extended memory in centralized sys-

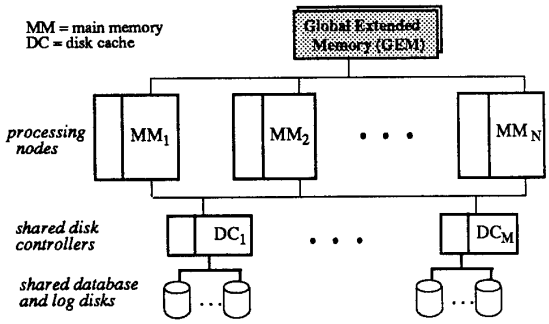

Fig. 2.1: Architecture of a closely coupled shared-disk system using Global Extended Memory

tems. The GEM interface is general and simple permitting fast access times. The use of special-purpose processors implementing global functions in hardware or microcode, on the other hand, results in a specialized interface (e.g., lock and unlock operations) and incurs a high hardware development cost.

A general application of GEM is to use it for inter-node communication such that all messages are exchanged across the GEM. Such a fast communication is a general service to be implemented by the operating system and can by used by database sharing and database partitioning systems without affecting the DBMS. Database sharing systems can utilize GEM in a number of more specific ways to be supported by the DBMS. In particular, the most performance-critical functions of database sharing systems, concurrency control and coherency control, may greatly benefit from GEM (see below). Furthermore, GEM can be used to support load control functions, e.g., by keeping system-wide status information for transaction routing, and to efficiently construct a global log by merging local log data [Ra91a].

Global *concurrency control* is necessary for database sharing to synchronize accesses to the shared database and enforce serializability. In loosely coupled systems, the communication overhead for concurrency control can substantially affect overall performance so that it is of paramount importance to find algorithms that reduce the number of remote lock requests as far as possible. (In this paper, we consider only locking schemes for concurrency control.) An overview of concurrency control protocols for data sharing can be found in [Ra91b]. A storage-based communication with GEM could already improve performance by reducing the communication overhead. An alternative is to store a global lock table (GLT) in GEM that can be accessed by all nodes. Information on lock ownerships and waiting (incompatible) lock requests of the entire system has to be stored in this table to permit every node to decide upon whether or not a lock request can be granted. Changing control information in the GLT, e.g., for acquiring or releasing a lock, requires (at least) two GEM accesses: one to read the lock entry into main memory and another one to write the modified value back to the GEM. In [Ra91a], we have described such a protocol in detail. In the simplest approach, every lock request is processed with the GLT in GEM. A refinement to reduce the number of GEM accesses is to authorize the node's local lock managers to locally process certain lock requests.

Coherency control is required to avoid access to invalidated (obsolete) database pages cached in local main memory database buffers. Fortunately, it is possible to detect these buffer invalidations with no extra communication by using extended lock information (e.g., page sequence numbers that are incremented for every modification) [Ra86, Ra91b]. If we use a global lock table in GEM for concurrency control, the additional information for modified pages can also be recorded in this table to support coherency control.

Coherency control further requires an appropriate update propagation strategy to provide transactions with the current versions of database pages. The solution to this problem closely depends on whether a FORCE or NOFORCE scheme [HR83] is used for writing modified data to external storage (e.g., disk). With FORCE all pages modified by a transaction are forced (written) to the permanent database on external storage before commit. This approach simplifies coherency control since it ensures that the most recent version of a page can always be obtained from external storage. However, FORCE introduces a high I/O overhead and significant response time delays for update transactions when all force-writes go to disk. Therefore, high performance systems usually adopt the NOFORCE alternative which only requires writing log data at commit. Since the permanent database is generally not up-to-date for NOFORCE, 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, *page requests* may have to be sent to the node holding the current page copy in its buffer. The modified page may be returned across the shared disks or, preferably, across the communication system to avoid the disk delays. Alternatively, pages may be exchanged across GEM in a closely coupled system.

Our performance study for centralized systems has shown that FORCE can approach the performance of NOFORCE when the force-writes go to non-volatile semiconductor memory [Ra92a]. In this study, we will investigate whether this also holds for closely coupled database sharing systems and how the use of GEM affects coherency control performance.

3. Simulation model

We developed a comprehensive simulation system to study the performance of closely coupled database sharing systems and to carry out a comparison with loosely coupled systems. In the closely coupled configurations, we use GEM for global concurrency and coherency control and to store database files. For loose coupling, we employ the primary copy locking protocol for concurrency and coherency control [Ra86] which showed the best performance of several alternatives in trace-driven simulations of loosely coupled database sharing systems [Ra88]. For both coupling modes, we support the FORCE as well as the NOFORCE strategy. The shared disks may have a volatile or non-volatile disk cache to implement a global database buffer. Our simulation model supports both synthetic workloads and the use of database traces. Workload allocation can be at random or affinity-based in order to support locality of reference. The simulation model has been implemented using the DeNet simulation language [Li89].

Our simulation system consists of three major parts (Fig. 3.1): a SOURCE which generates and distributes the workload of the system, several processing nodes to execute the workload, and a set of peripheral devices like GEM and magnetic disks with or without disk cache. In subsection 3.1, we describe the SOURCE component as well as our database model. Subsections 3.2 and 3.3 cover the models for processing nodes and external storage devices, respectively. For the sake of brevity, we omit a specification of all parameters. However, the major parameter settings used in the experiments are discussed in section 4.

3.1 Workload generation and allocation

To cover a wide range of applications, we have built several workload generators supporting synthetic workloads and the use of database traces. In this paper, we consider two workload types namely synthetically generated debit-credit workloads and the trace-driven approach.

Debit-credit currently represents the most important workload type for evaluating the performance of transaction and database systems since it is the base for the widely used TPC-A and TPC-B benchmarks [Gr91]. This well-known workload consists of a single transaction type of a banking application that accesses/

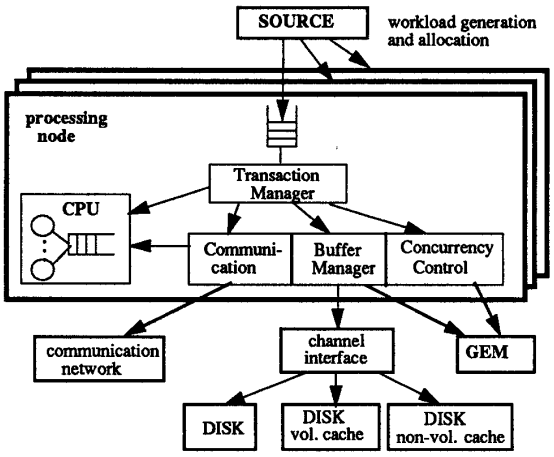

Fig. 3.1: Structure of the simulation system

updates four record types (ACCOUNT, BRANCH, TELLER and HISTORY). In our model, each record type is stored in a separate database *partition*. Partitions are primarily used to allocate the database to external storage devices. A partition consists of a number of database pages which in turn consist of a specific number of records. The number of records per page is determined by the *blocking factor* which can be specified on a per-partition basis. Differentiating between records and pages is important in order to study the effect of clustering which aims at reducing the number of page accesses (disk I/Os) by storing related objects into the same page.

There is a many-to-one relationship between ACCOUNT and BRANCH records and between TELLER and BRANCH records. While the BRANCH record is randomly selected for a transaction, the TELLER record is (randomly) selected from the set of TELLER records associated with the selected BRANCH record. As required by the TPC benchmark definitions [Gr91], we ensure that 85% of the ACCOUNT accesses are to an account associated with the selected branch, while the remaining accesses go to an account of another branch. The HISTORY partition is sequentially accessed by all transactions. Our simulation system permits clustering of BRANCH and TELLER records. In this case, TELLER records are stored in the same page where their associated BRANCH record is stored. This reduces the number of page accesses per transaction to three and is likely to improve hit ratios; for page-locking the number of locks per transaction is also reduced by one.

Every transaction references the four record types in the same order so that no deadlocks can occur. The small TELLER and BRANCH record types are accessed last to keep lock holding times for them as short as possible.

In the trace-driven simulations, the database and load model is largely determined by the trace information and the underlying application. A trace consists of a certain number of transactions of different types. For every transaction, the transaction type and all database (page) references with their access mode (read or write) are recorded in the trace. Our workload generator simply extracts the transactions from the trace and submits them to the processing node according to a specified arrival rate. There may be a common arrival rate for all transactions in the trace preserving the original execution order of the workload. Alternatively, we can specify a different arrival rate per transaction type.

For both workload types, we support random routing as well as an affinity-based routing for workload allocation. For random routing, we merely ensure that every node is assigned about the same number of transactions to support load balancing. Affinity-based routing aims at supporting load balancing as well as a maximum degree of node-specific locality [Ra92b]. This is easy to achieve for debit-credit by a BRANCH-based partitioning of the workload so that every node has to process transactions for the same number of branches. Since a transaction always accesses TELLER and HISTORY records associated with the respective branch, this routing also results in a partitioned access to the HISTORY and TELLER record types. For ACCOUNT, a maximum of 15% of the accesses may be to a branch associated with another processing node. In the trace-driven simulations, workload allocation can be defined by a so-called routing table to achieve an affinity-based routing. The routing table defines for every transaction type in the trace to which node(s) transactions of this type should be routed. To determine the routing tables, we applied iterative heuristics that use the reference distribution of the workload and the number of nodes as input parameters to find an affinity-based workload assignment [Ra92b].

3.2 Model of processing nodes

As indicated in Fig. 3.1, a processing node is represented by a transaction manager, a buffer manager, a concurrency control component, a communication subsystem and CPU servers. The *transaction manager* controls execution of the transactions received from the SOURCE. The maximal number of concurrently active transactions is restricted by a specified multiprogramming level (MPL). When all MPL "processing slots" are occupied, newly arriving transactions must wait in an input queue until they can be served. To account for the execution cost of a transaction, the transaction manager requests CPU service at the beginning of a transaction, for every record access and at the end of a transaction. The actual number of instructions for each of these services is exponentially distributed over a mean specified as a parameter. Processing a record access also entails requesting an appropriate (read or write) lock from the concurrency control component and asking the buffer manager to bring the corresponding database page into the main memory buffer (if not there already). Commit processing consists of two phases. In phase 1, the buffer manager is requested to write log data and possibly to force modified database pages to non-volatile storage. In phase 2, the concurrency control component is requested to release the transaction's locks.

The *buffer manager* is responsible for caching database pages in main memory and for logging. Furthermore, it implements coherency control in conjunction with the concurrency control component. The database buffer is managed according to a LRU (least recently used) replacement strategy. Logging is modelled by writing a single page per update transaction to the log file at commit. In the case of a FORCE update strategy, all pages modified by a transaction are also written out at commit time. In the case of NOFORCE, we have ignored the checkpointing overhead assuming a fuzzy checkpointing scheme [HR83] which incurs little overhead during normal processing. For every I/O, the buffer manager requests CPU service to account for the I/O overhead. I/O operations to files allocated in GEM are performed synchronously, i.e., the CPU is kept busy until the read

or write access is completed. *Concurrency control* is based on strict two-phase locking (long read and write locks) together with a deadlock detection scheme. Page-level locking is employed to permit an integrated treatment of concurrency and coherency control. For comparison purposes, it is also possible to switch off concurrency control for a database partition (no lock conflicts). Switching off concurrency control may also be appropriate for objects for which accesses are synchronized by using latches or tailored protocols incurring negligible data contention.

As mentioned above, two concurrency/coherency control schemes for database sharing have been implemented. For close coupling, we use a global lock table in GEM to process all lock requests and releases. Entries of the global lock table are synchronously accessed and written back using a Compare&Swap operation. Detection of buffer invalidations is based on page sequence numbers that are stored in the page header and in the global lock table. For NOFORCE, we additionally record in the global lock table the current "page owner", i.e., the node where the page was modified most recently. When a buffer invalidation is detected during lock processing (by comparing the page sequence number of the GLT with the one of the cached page copy) or when no copy of the page is locally cached, the page is requested from the current owner rather than read from external storage. Pages are returned in a "large" message across the communication system. After the owner has written out a modified page, the GLT entry is adapted to indicate that the most recent page version can be read from the permanent database.

For loose coupling, primary copy locking (PCL) is employed for concurrency/coherency control. In this distributed scheme, the database is logically partitioned so that each node is assigned the synchronization responsibility (or *global lock authority, GLA*) for one partition. Lock requests against the local partition can be handled without communication overhead and delay, while other requests have to be directed to the authorized processor holding the GLA for the respective partition. To reduce the number of remote lock requests, GLA and workload allocation should be coordinated. Our heuristics for determining routing tables also compute GLA assignments to achieve a maximum degree of local lock processing. For debit-credit, each node is assigned the GLA for an equal number of BRANCH records and their associated TELLER, ACCOUNT and HISTORY records.

Coherency control for PCL also uses pages sequence numbers to detect buffer invalidations without extra messages. For NOFORCE, a special update propagation scheme is used that avoids extra messages for page transfers as well. In this case, the GLA node additionally functions as the owner for all pages of its partition. Therefore, when a page is modified at a node not holding the GLA, the new page version is returned to the GLA node at commit. This does not require an extra message since the page transfer can be combined with the lock release message. A consequence of this approach is that the current version of a page can always be obtained from the GLA node or from the permanent database. This has the advantage that the current version of a page can be supplied by the GLA node together with the lock grant message, thereby avoiding extra messages and delays for page requests. For more details of the protocol, the reader is referred to [Ra86, Ra91b].

The communication subsystem processes send and receive operations for messages introduced by the concurrency/coherency control protocol. Messages are exchanged over an interconnection network and cause CPU overhead at the sending and receiving node. The CPU overhead for "long" messages (page transfers) is higher than for "short" messages. The number of CPUs per node and the capacity per CPU (in MIPS) are provided as simulation parameters.

3.3 External devices

Database and log files can be allocated to a variety of external storage devices. In this study, we consider allocation to conventional disks, disks with volatile or non-volatile disk caches, and to GEM. All storage units are modelled as servers to account for possible queuing delays. Disk access times consist of three major components: transmission delay for exchanging pages between main memory and disk controllers, controller delay and disk delay. The disk delay can be avoided in some cases when volatile or non-volatile disk caches are used. For the management of these caches we followed the realization of commercial (IBM's) disk caches [Gr89]. In particular, LRU is used for page replacement. Volatile disk caches avoid a disk access for reads that can be satisfied in the cache (read hit). Non-volatile disk caches additionally try to satisfy all write I/Os in the disk cache and to update the disk copy of a modified page asynchronously. The GEM implementation supports different service times for page and entry accesses. The communication network is represented by a simple delay model characterized by a fixed transmission bandwidth.

4. Performance Analysis

In this section, we present some results obtained with the described simulation model. Response time will be the primary performance metric in this study since our simulation system uses an open queuing model. (Detailed statistics on the composition of response time and device utilization, waiting times, queue lengths, lock behavior, hit ratios, etc. are also available in order to explain the results). Most of our experiments (subsections 4.2 through 4.5) use the debit-credit workload. In 4.1, the parameter settings for these runs are described. We first analyze closely coupled configurations for different routing and update propagation strategies (4.2), buffer sizes (4.3) and database allocations (4.4). In subsections 4.5 and 4.6, we present a performance comparison between closely and loosely coupled database sharing systems for debit-credit and a real-life workload represented by a database trace, respectively.

4.1 Parameter settings

Table 4.1 shows the most important parameter settings for the debit-credit experiments. The number of nodes (N) has been varied from 1 to 10 with an arrival rate of 100 TPS (transactions per second) per node. The database size grows proportionally with the arrival rate as required by the TPC benchmarks [Gr91]. Thus we have a 10-fold database size for 10 nodes (100 million ACCOUNT records) compared to the central case. In all experiments, we used clustering of BRANCH and TELLER records (see 3.1) so that BRANCH and TELLER records reside in the same partition and only three different pages are accessed by a transaction. The size of the HISTORY partition is immaterial here since every transaction adds a new record at the end of this sequential file.

The CPU capacity is 40 MIPS per node (4 processors of 10 MIPS) leading to a utilization of at least 62.5% for 100 TPS and an average pathlength of 250.000 instructions per transaction

Parameter	Settings
number of nodes N	1 - 10
arrival rate	100 TPS per node
DB size (per 100 TPS)	BRANCH: 100 records, blocking factor 1 TELLER: 1.000 records, bl. factor 10 (clustered w. BRANCH) ACCOUNT: 10.000.000 records, blocking factor 10 HISTORY: blocking factor 20
path length	250.000 instructions per transaction
lock mode	page locks for BRANCH, TELLER, ACCOUNT; no locks for HISTORY
CPU capacity	per node: 4 processors of 10 MIPS each
DB buffer size	200 (1000) pages per node
GEM parameters	1 GEM server; 50 μs avg. access time per page; 2 μs avg. access time per entry
communication	bandwidth: 10 MB/s 5000 instr. per send or receive of "short" messages (100 B) 8000 instr. per send or receive of "long" messages (4 KB)
I/O overhead	3000 instructions per page (GEM: 300 instr. for initialization)
avg. disk access time	15 ms for DB disks; 5 ms for log disks
other I/O delays	avg. controller service time: 1 ms; avg. transfer time per page: 0.4 ms

Tab. 4.1: Parameter settings for debit-credit workload

(not including I/O and communication overhead). The default buffer size per node is 200 pages resulting in an aggregate buffer size that is twice as large as the BRANCH/TELLER database partition. We did not set locks for HISTORY assuming an implementation that synchronizes accesses to the current end of this file by latches. Without I/O queuing delays, the average access time per page is 50 μs for GEM, 1.4 ms for disk cache, 6.4 ms for log disks and 16.4 ms for disks storing database partitions. For log disks, a reduced access time has been assumed since the log file is sequentially accessed shortening disk seek times. The average GEM access time per entry is assumed to be 2 μs. The communication overhead is 5000 (8000) instructions for sending or receiving a short (long) message. Thus a remote lock request for PCL costs at least 20.000 instructions, a page request 26.000 instructions. The multiprogramming level has been chosen high enough to avoid queuing delays at the transaction manager.

Parameters that are changed include the concurrency/coherency control strategy (close vs. loose coupling), the number of nodes, the routing strategy, the update strategy, buffer size, the allocation of log and database files, and the workload type (debit-credit vs. trace-driven).

4.2 Influence of workload allocation and update strategy

The first experiments concentrate on closely coupled database sharing configurations using a global lock table in GEM for concurrency control. Fig. 4.1 shows the average transaction response times for up to 10 nodes and different workload allocation (random vs. affinity-based routing) and update propagation schemes (FORCE vs. NOFORCE). All database and log files are allocated to a sufficient number of disks (without disk caches) to avoid I/O bottlenecks. While random routing may rarely be used for debit-credit workloads in practice, considering this case illustrates the performance implications when no affinity-based routing can be found.

Fig. 4.1 shows that for both update strategies response times remain almost constant in the case of affinity-based routing when increasing the number of nodes, despite the linear increase in throughput by 100 TPS per node. This was possible although we do not utilize locality of reference for locking in the closely coupled configurations, but access the global lock table in GEM for every lock. However due to the short entry access times of GEM, this resulted in an almost negligible delay for concurrency control. Even for 1000 TPS (10 nodes) GEM utilization was less than 2% so that no significant queuing delays occurred.

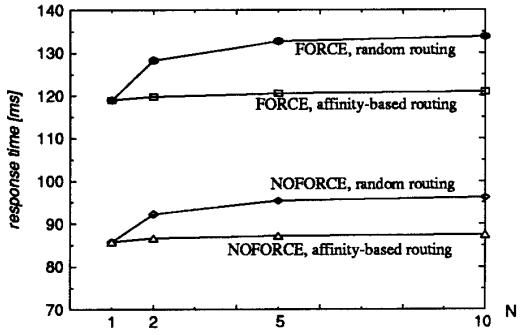

Fig. 4.1: Influence of workload allocation and update strategy for GEM locking (100 TPS per node)

Response times for FORCE are higher than for NOFORCE primarily because of the I/O delays for the force-writes at commit. In contrast to affinity-based routing, response times for random routing increase with more nodes. This was not because of concurrency control since the cost of GEM locking is very low independent of the workload allocation. Rather the response time increase for random routing is due to a growing number of buffer invalidations and reduced effectiveness of main memory buffering. These effects are exclusively caused by the small BRANCH/TELLER partition since HISTORY pages are sequentially accessed[1] and no hits could be achieved for the huge ACCOUNT record type. With random routing the same BRANCH/TELLER pages are referenced and modified in all nodes so that with a growing number of nodes these pages are redundantly cached and invalidated to an increasing extent. This caused the hit ratios for BRANCH/TELLER accesses to drop from 71% in the centralized case to 13% for 5 and merely 7% for 10 nodes. With affinity-based routing, we achieved in all database sharing configurations the same hit ratios as for one node since BRANCH/TELLER references occur completely partitioned in this case (see 3.1).

For random routing, the drop in hit ratios led to a larger response time increase for FORCE than for NOFORCE (Fig. 4.1). FORCE results in a disk access for every buffer invalidation and buffer miss to read in the respective page. For NOFORCE, on the other hand, many BRANCH/TELLER pages could be obtained much faster by sending a page request to another node[2]. Since the number of page requests per transaction increased with the number of nodes, the difference between the FORCE and NOFORCE results also increased.

4.3 Influence of buffer size

To study the influence of the buffer size, we conducted simulations with an increased buffer size of 1000 pages per node and compare the results with those for buffer size 200 (Fig. 4.2). Since response times remain largely unchanged for affinity-based routing when increasing the number of nodes, we only consider random routing in this experiment. The larger buffer could hold all BRANCH/TELLER pages in the centralized case

1. Due to the blocking factor of 20, we have a hit ratio of 95% for HISTORY.
2. A page request caused an average delay of about 6.5 ms until the requested page was received, compared to more than 16.4 ms for a disk access. Most of the delay for a page request was due to CPU queuing and service times for message processing.

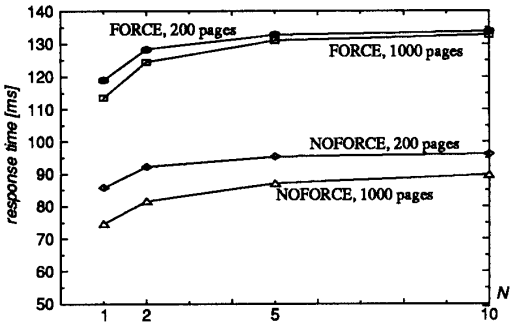

Fig. 4.2: Influence of buffer size for random routing

permitting an optimal hit ratio. In the database sharing configurations however, the effectiveness of the larger buffer decreases with more nodes since it caused even more buffer invalidations than for the small buffer size. This was particularly negative for FORCE which benefits much less from the larger buffer than NOFORCE (Fig. 4.2). With NOFORCE, almost all buffer misses for BRANCH/TELLER could be satisfied by a page request while for the small buffer size this was only partially possible. Thus the negative impact of buffer replication on hit ratios and the number of invalidations is reduced by the fact that it also supports a fast update propagation between nodes. On the other hand, page requests cause a higher CPU overhead than disk I/Os causing increased CPU queuing delays and lower throughput (see below).

4.4 Influence of database allocation

Due to the high access frequency to the "hot" BRANCH-/TELLER partition this database file is largely responsible for the I/O and coherency control performance for debit-credit. Therefore, we completely allocated this partition to GEM in our next experiment (All other files were still allocated to magnetic disks). Fig. 4.3 compares the resulting response times with the disk-based allocation for both routing and update strategies. Note that both diagrams in Fig. 4.3 use the same scale to facilitate a comparison of the results.

Fig. 4.3a illustrates that allocating the BRANCH/TELLER partition to GEM has almost no effect on response times for NOFORCE: with both routing strategies no significant improvements compared to the disk-based allocation are achieved. For buffer size 1000 and affinity-based routing there are almost no I/Os on BRANCH/TELLER for NOFORCE so that the GEM allocation did not improve performance. With random routing, the GEM allocation also showed little impact on response times since almost all buffer misses for BRANCH/TELLER are satisfied by page requests and direct page transfers between nodes. Hence, for NOFORCE the GEM allocation can only improve performance for small buffer sizes when not all buffer misses can be avoided or satisfied by a page request. GEM could also be used for implementing the page transfers.

On the other hand, the GEM allocation of BRANCH/TELLER significantly improved performance for FORCE, in particular with random routing (Fig. 4.3b). One disk access is already saved by this allocation for writing the modified BRANCH/TELLER page at commit. In addition, all buffer misses can be satisfied very fast from GEM rather than from disk. This was particularly helpful for random routing with its high number of buffer invalidations and misses. As a result, the decrease in hit ratios with more nodes had almost no impact on response times. Thus, GEM allocation of the BRANCH/TELLER partition in combination with the fast GEM locking achieved almost the same response times for random routing than for affinity-based routing in the case of FORCE! Furthermore, no significant response time increase compared to the central case occur anymore.

Since I/O delays can also be reduced by disk caches, we additionally considered allocation of the BRANCH/TELLER partition to disks with volatile or non-volatile disk caches. The response time results for these configurations in the case of FORCE are shown in Fig. 4.4. For random and affinity-based routing, a non-volatile disk cache achieved almost the same response times as with the GEM allocation. This was because all BRANCH/TELLER pages could be buffered in the shared disk cache. Hence, all misses in main memory could be satisfied from this global buffer saving the disk access. In addition, the disk delay for the force-write at commit could be avoided due to the non-volatility of the cache. In contrast, the volatile disk cache can only avoid reading disk accesses. This helped to achieve about the same response times for random routing than with an affinity-based routing since all buffer invalidations could be satisfied from the shared disk cache. For affinity-based routing, a volatile disk cache is not useful because in this case

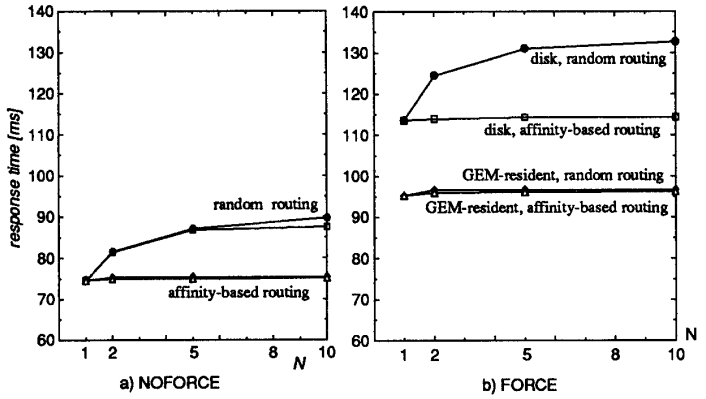

Fig. 4.3: Influence of storage allocation for BRANCH/TELLER (buffer size 1000)

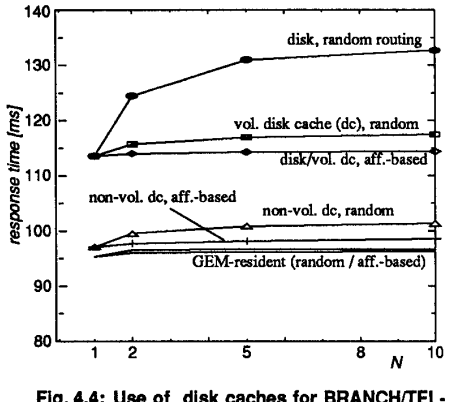

Fig. 4.4: Use of disk caches for BRANCH/TELLER partition (FORCE, buffer size 1000)

no main memory misses occur on BRANCH/TELLER for the chosen buffer size (1000). Response times for FORCE could be further improved by using a non-volatile disk cache for the HISTORY and ACCOUNT disks to speed-up the force-writes for these files.

Our results illustrate that a close coupling via GEM permits a very efficient solution for concurrency and coherency control. Even for random routing and update-intensive workloads such as debit-credit, the negative impact of buffer invalidations can largely be avoided by allocating "hot" database partitions to GEM or using a shared disk cache to hold frequently modified pages. Hence, such a storage allocation permits the choice of simpler workload allocation strategies without sacrificing performance.

4.5 Loose vs. close coupling

To compare close with loose coupling we repeated most of the above experiments for loosely coupled systems using PCL for concurrency and coherency control. Fig. 4.5 shows the resulting response time results for both PCL and GEM locking and for the different buffer sizes, update and workload allocation strategies. All files are allocated to magnetic disks without cache.

A first observation is that in the case of affinity-based routing, PCL always achieved virtually the same response times than GEM locking. This was because the coordination between workload and GLA allocation permitted a local concurrency control for PCL for almost all accesses[3]. Furthermore, the same I/O behavior than with close coupling was achieved without replicated caching of pages and buffer invalidations. These results underline that the debit-credit workload permits an ideal partitioning for distributed systems supporting a linear throughput increase without significant response time deterioration.

Significant differences between the loosely and closely coupled configurations occur for random routing. In this case, PCL is always worse than GEM locking because of the communication overhead for concurrency control. While GEM locking does not incur a significant overhead and delay, the communication overhead for PCL grows with the number of nodes leading to increasing response time differences compared to close coupling.

3. Communication may only be needed for up to 15% of the ACCOUNT accesses. Hence, at most 0.15 global lock requests (0.6 messages) per transaction are needed for PCL and affinity-based routing.

While 50% of the lock requests could be locally processed for two nodes with PCL, this share is reduced to 10% in the case of 10 nodes. However, the maximal number of messages is still comparably small since there are at most two remote lock requests per transaction (for ACCOUNT and BRANCH/TELLER).

As can be seen from Fig. 4.5, the difference between PCL and GEM locking is smaller for NOFORCE than for FORCE and for NOFORCE it is further reduced for buffer size 1000 compared to buffer size 200. This is because PCL supports a more efficient update propagation for NOFORCE than with GEM locking. For GEM locking buffer invalidations cause extra messages for page requests and transfers, while PCL always combines these messages with regular concurrency control messages (see 3.2). As a result, page transfers only cause a slight increase in communication overhead for PCL which was particularly helpful for a larger buffer. Of course, the total communication overhead is still higher for PCL since the number of global lock requests is more than twice as high than the number of page requests (for ACCOUNT lock requests but no page requests occur).

The communication overhead limits the achievable transaction rates for PCL compared to GEM locking. This is illustrated by Fig. 4.6 depicting the transaction rates per node for a CPU utilization of 80%. For affinity-based routing there is almost no communication overhead permitting a linear throughput increase. With random routing, the maximal throughput is about 15% lower for the message-based PCL protocol compared to close coupling. FORCE supports higher transaction rates than NOFORCE for random routing since the overhead per I/O is much smaller than for page transfers. For GEM locking the page requests/transfers introduce a notable overhead since they cannot be combined with other messages as for PCL.

4.6 Results for real-life workload

For a further evaluation of closely coupled database sharing systems we conducted a number of trace-driven simulation runs. We present the results for one of the traces with a high share of read accesses. The trace consists of more than 17.500 transactions of twelve transaction types and about 1 million database accesses. There are significant variations in transaction size; the largest transaction (an ad-hoc query) performs more than 11.000

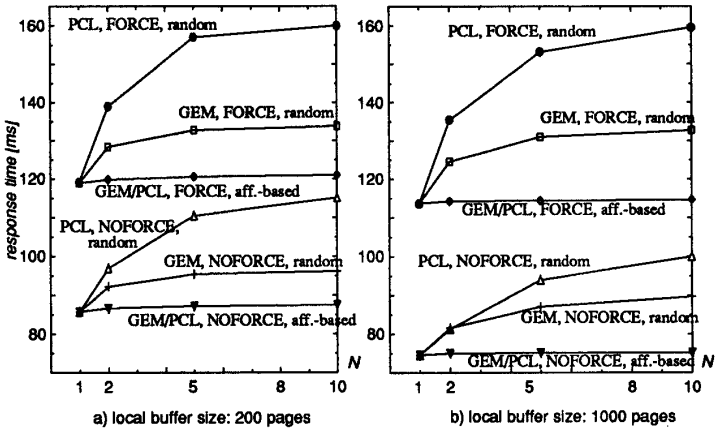

Fig. 4.5: Primary Copy Locking (PCL) vs. GEM locking

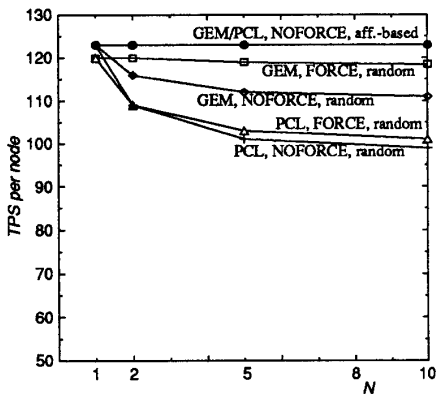

Fig. 4.6: Throughput per node for PCL and GEM locking (buffer size 1000)

accesses. The database size is about 4 GB, but merely 66.000 different pages in 13 files were referenced during the trace period. Access distribution is highly non-uniform even within transaction types making it difficult to support an affinity-based routing. About 20% of the transactions perform updates, but only 1.6% of all database accesses are writes.

Fig. 4.7 shows the response time results obtained for this workload for GEM locking and PCL with random and affinity-based routing. Response times refer to an artificial transaction performing the average number of database accesses. The parameter settings have largely been chosen as for debit-credit (Table 4.1), in particular the CPU and device characteristics. The number of nodes has been varied only between 1 and 8. We selected an arrival rate of 50 TPS per node and a local buffer size of 1000 pages. We only consider the NOFORCE results for the real-life workload since FORCE did not cause much differences due to the low update activity.

Fig. 4.7 shows that for both workload allocation strategies close coupling clearly outperforms loose coupling and that the response time differences increase with the number of nodes. As for the debit-credit workload, GEM locking did not cause a significant delay for global concurrency control. Therefore, the results for close coupling are largely influenced by the I/O behavior and thus by the workload allocation. With affinity-based routing, we achieved better response times for the closely coupled database sharing configurations than in the central case. This is because the aggregate buffer size in the system increases with the number of nodes while the database size remains constant in the trace-driven simulations (in contrast to debit-credit). This supported better hit ratios for database sharing than in the central case. With random routing, on the other hand, response times for database sharing are higher and deteriorate with increasing number of nodes despite the higher buffer capacity. Buffer effectiveness for random routing is lowered by a high degree of replicated caching. Furthermore, random routing apparently caused a lower degree of inter-transaction locality than in the central case. Due to the low update frequency, buffer invalidations as well as lock conflicts had no significant impact on performance for the real-life workload.

While the loosely coupled configurations experienced a similar I/O behavior, their performance suffered from the communication overhead for concurrency control. For PCL, the number of global lock requests increased with more nodes, even in the case of affinity-based routing. This was caused by the non-uniform access distribution for the real-life workload and its limited "partitionability". Furthermore, since the database size remains constant the average GLA partition size decreases with the number of nodes almost inevitably leading to more global lock requests. Thus the share of locks that could be locally processed dropped from 63% for 2 to 35% for 8 nodes with affinity-based routing (for random routing from 50% to 12.5%). These shares could be improved by a read optimization permitting a node in many cases to locally process read locks even without GLA [Ra86, Ra91b]. This optimization which is already reflected in Fig. 4.7 allowed a local processing for 78% (65%) of the locks for 2 nodes and 65% (33%) for 8 nodes with affinity-based (random) routing.

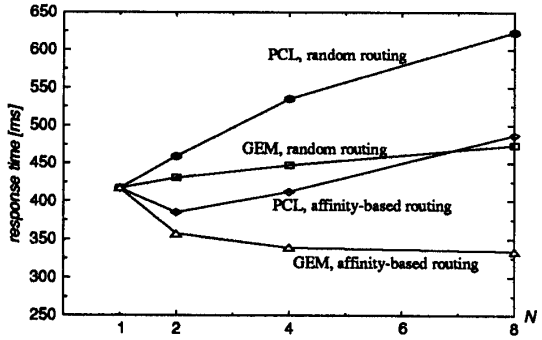

Fig. 4.7: PCL vs. GEM locking for real-life workload (50 TPS and 1000 pages per node)

In the loosely coupled configurations, CPU utilization was substantially higher and more unbalanced than for close coupling thereby reducing the achievable throughput. With GEM locking CPU utilization was balanced and merely about 45% for 50 TPS per node. The communication overhead for PCL caused an average CPU utilization of 78% for 8 nodes and random routing with some nodes utilized by more than 85%. This led to significant queuing delays and increased delays for remote lock requests. For 8 nodes the achievable throughput for loose coupling is about 30% lower than with GEM locking, with random routing even by more than 40%. Close coupling supports a linear throughput increase for the real-life workload even with random routing.

5. Related Work

Numerous papers have studied concurrency and coherency control in loosely coupled database sharing systems. A discussion of these studies is beyond the scope of this paper, but can be found in our survey of the field [Ra91b]. Here, we will only discuss papers that have considered a close coupling and are therefore directly related to this study.

In [Yu87], the performance of database sharing systems using a centralized lock engine for concurrency control was analyzed. Details of the lock engine realization were not discussed. In the evaluation, lock service times between 100 and 500 μs were assumed so that much smaller transaction rates than with GEM locking could be supported. Performance was largely determined by lock contention and an inefficient coherency control approach requiring a broadcast message for every update transaction to detect buffer invalidations. Furthermore, a disk-based FORCE scheme was used for update propagation.

In [DIRY89, DDY91, YD91], a global semiconductor store called Shared Intermediate Memory (SIM) was considered for use in database sharing systems. In contrast to GEM, SIM is assumed to be managed by a central controller and is purely page-oriented. The only usage form of SIM is for caching database pages to improve the I/O behavior as it can already be achieved by a shared disk cache. All three studies assumed a FORCE strategy for update propagation and used analytical performance models. The studies found that the SIM-based configurations support much better response times than without such a memory. This was primarily because the response time delay of FORCE could be avoided by updating the disk asynchronously from SIM. However, such an improvement is not a database sharing-specific optimization but can already be achieved in centralized systems, e.g., by a non-volatile disk cache [Ra92a]. [DDY91] also found that response time increases due to buffer invalidations can be kept very low when frequently modified pages are cached in SIM. This corresponds to our observations

for using shared disk caches or a GEM allocation of those files in the case of FORCE. The focus of [DDY91], however, was on comparing different choices of which pages are cached in the shared buffer. Limitations of the study (besides the restriction to FORCE) include the assumption of random routing and the use of broadcast invalidation for coherency control.

6. Conclusions

We have presented a performance evaluation of closely coupled database sharing systems using Global Extended Memory (GEM). GEM is a fast, non-volatile semiconductor memory that can be synchronously accessed by all processing nodes. Its access interface consists of read and write operations to pages and so-called entries as well as operations for access synchronization (Compare&Swap). GEM can be used to improve I/O performance (reduce the number of disk accesses), to implement a fast message transfer between nodes, and to store global data structures, e.g., a global lock table for concurrency control.

We compared the performance of closely coupled database sharing systems using a global lock table in GEM with loosely coupled systems applying primary copy locking. We found that GEM locking introduces an almost negligible delay and overhead for concurrency control leading to significantly better response times and transaction rates than with loose coupling. This was particularly significant for real-life workloads for which it is often difficult to determine an affinity-based workload allocation supporting a high degree of local lock processing and balanced system utilization. With GEM locking high performance could even be achieved for random routing since the cost of global locking is independent of the routing strategy and reference distribution of the workload. Hence, GEM locking reduces the need for affinity-based routing making it easier to effectively utilize the processing capacity of distributed systems.

However, the routing strategy also affects I/O performance so that considering the workload's reference characteristics for transaction routing may still be important. In particular, random routing leads to many buffer invalidations for update-intensive workloads thereby lowering hit ratios. As our study has shown, the performance implications of these buffer invalidations largely depend on the update strategy and the chosen database allocation. For a conventional, disk-based database allocation NOFORCE supports significantly better performance than FORCE. With FORCE, each buffer miss results in a disk access while NOFORCE frequently permits requesting the page much faster from another node, in particular for larger buffers and with a higher number of nodes. On the other hand, allocating frequently modified files in non-volatile semiconductor memory like GEM or using a global database buffer for them (e.g., a shared disk cache) results in a more efficient coherency control for FORCE than for NOFORCE. This is because accessing the shared semiconductor memory is much faster and causes less CPU overhead than requesting and transferring pages across the communication system. Using GEM for implementing the page transfers would also improve coherency control performance for NOFORCE. Thus even for random routing, the negative performance impact of buffer invalidations can largely be avoided by utilizing shared semiconductor memory.

For the real-life workload, hit ratios were significantly lower for random routing than for the (sub-optimal) affinity-based workload allocation even so almost no buffer invalidations occurred. This was because a lower degree of inter-transaction locality than in the central case could be preserved and because the higher amount of replicated caching reduced the buffer effectiveness. Without affinity-based routing, these effects can only be compensated by paying the extra cost for increased buffer sizes (larger main memory).

References

BDS79 Behman, S.B.; DeNatale, T.A.; Shomler, R.W. : Limited lock facility in a DASD control unit. Technical report TR 02.859, IBM General Products Division, San Jose, 1979.

Bh88 Bhide, A.: An analysis of three transaction processing architectures. *Proc. of the 14th Int. Conf. on Very Large Data Bases*, Long Beach, CA, 339-350, 1988.

CKB89 Cohen, E.I.; King, G.M.; Brady, J.T.: Storage hierarchies. *IBM Systems Journal* 28 (1), 62-76, 1989.

DDY91 Dan, A.; Dias, D.M.; Yu, P.S.: Analytical modelling of a hierarchical buffer for a data sharing environment. *Proc. ACM SIGMETRICS Conf.*, 156-167, 1991.

DG92 DeWitt, D.J., Gray, J.: Parallel database systems: the future of high performance database systems. *CACM* 35 (6), 85-98, 1992.

DIRY89 Dias, D.M.; Iyer, B.R.; Robinson, J.T.; Yu, P.S.: Integrated concurrency-coherency controls for multisystem data sharing. *IEEE Trans. on Software Engineering* 15 (4), 437-448, 1989.

Gr89 Grossman, C.P.: Evolution of the DASD Storage Control. *IBM Systems Journal* 28 (2), 196-226, 1989.

Gr91 Gray, J. (ed.): *The Benchmark Handbook for Database and Transaction Processing Systems*. Morgan Kaufmann 1991.

HR83 Härder, T.; Reuter, A.: Principles of transaction-oriented database recovery. *ACM Computing Surveys* 15 (4), 287-317, 1983.

Ki84 Kim, W.: Highly available systems for database applications. *ACM Computing Surveys* 16 (1), 71-98, 1984.

KLS86 Kronenberg, N.P.; Levy, H.M.; Strecker, W.D.: VAX clusters: a closely coupled distributed system. *ACM Trans. Comp. Systems* 4 (2), 130-146, 1986.

Li89 Livny, M.: DeNet User's Guide. Version 1.6, Comp. Science Dept., Univ. of Wisconsin, Madison, 1989.

MN91 Mohan, C., Narang, I.: Recovery and coherency-control protocols for fast intersystem page transfer and fine-granularity locking in a shared disks transaction environment. *Proc. of the 17th Int. Conf. on Very Large Data Bases*, Barcelona, 193-207, 1991.

ÖV91 Özsu, M.T., A Valduriez, P.: *Principles of distributed database systems*. Prentice-Hall, 1991.

Ra86 Rahm, E.: Primary copy synchronization for DB-sharing. *Information Systems* 11 (4), 275-286, 1986.

Ra88 Rahm, E.: Empirical performance evaluation of concurrency and coherency control protocols for data sharing. IBM Research Report RC 14325, IBM T.J. Watson Research Center, 1988. An extended version will appear in *ACM Trans. on Database Systems*.

Ra91a Rahm, E.: Use of Global Extended Memory for distributed transaction processing. *Proc. of the 4th Int. Workshop on High Performance Transaction Systems*, Asilomar, CA, Sep. 1991.

Ra91b Rahm, E.: Concurrency and coherency control in database sharing systems. Technical report ZRI 3/91, Univ. of Kaiserslautern, Dept. of Computer Science, Dec. 1991.

Ra92a Rahm, E.: Performance evaluation of extended storage architectures for transaction processing. *Proc. ACM SIGMOD Conf.*, San Diego, CA, 308-317, 1992.

Ra92b Rahm, E.: A framework for workload allocation in distributed transaction processing systems. *Journal of Systems and Software* 18, 171-190, 1992.

Sc87 Scrutchin Jr., T.W.: TPF: performance, capacity, availability. *Proc. Spring CompCon*, 158-160, 1987

Sm85 Smith, A.J.: Disk cache - miss ratio analysis and design considerations. *ACM Trans. on Computer Systems* 3 (3), 161-203, 1985.

St86 Stonebraker, M.: The case for shared nothing. *IEEE Database Engineering* 9 (1), 4-9, 1986.

YD91 Yu, P.S., Dan, A.: Comparison on the impact of coupling architectures to the performance of transaction processing systems. *Proc. 4th Int. Workshop on High Performance Transaction Systems*, Asilomar, CA, Sep. 1991.

Yu87 Yu, P.S. et al.: On coupling multi-systems through data sharing. *Proceedings of the IEEE* 75 (5), 573-587, 1987.