In: Proc. 6. GI/ITG-Fachtagung "Messung, Modellierung und Bewertung von Rechensystemen", München, Informatik-Fachberichte 286, Springer, 1991, pp. 92-108.

Extended Memory Support for High Performance Transaction Systems

V. Bohn, T. Härder, E. Rahm
University of Kaiserslautern, West Germany

Abstract

To achieve high performance transaction systems vertical as well as horizontal system growth is considered. A prime obstacle for linear performance growth is the unfavourable ratio of I/O time vs. CPU time in systems using conventional disk architecture. For this reason, we propose a fast and non-volatile extended memory which provides synchronous page-oriented access for closely coupled processors. We discuss its properties supporting high volume transaction processing. Subsequently, we investigate its performance behavior using simulation driven by load models derived form real applications. Simulation results are presented for centralized and distributed computing environments.

1. Introduction

Transaction programs (TAPs) implement administrative functions by accessing a shared database. They are used in a variety of business applications such as airline reservation, electronic banking, securities trading, communications switching, etc. to enable the on-line user to execute preplanned functions (canned transactions) by sending a request containing the transaction code (TAC) and the actual parameters to the transaction processing (TP) system. The essential software components of a TP system are the set of TAPs, the DBMS and the so-called TP monitor which coordinates the flow of transaction requests between terminals and TAPs as well as the DBMS calls of the TAPs.

The execution of a (single-step) transaction is basically performed as follows: The TP system organized a set of queues for the incoming requests and activates the corresponding TAPs upon availability of resources. The transaction typically issues a short sequence of simple DB operations which read and update only a few records. Transaction commit implies the following DBMS or TP monitor tasks:

- guarantee repeatability of the transaction's updates (e.g. by writing REDO log information to a non-volatile storage)
- save the output message (e.g. by using a message log)
- deactivate the related TAP
- attach the outgoing message to an output queue
- return the response to the terminal.

A typical TP application contains relatively few transaction types, e.g. a banking application has sometimes less than ten (with DEBIT-CREDIT as the most prominent transaction type). Other applications such as government services probably have less than a few hundred. The typical resource consumption of a transaction ranges between 0-30 disk I/Os, 100K-1M machine instructions and 2-20 messages (depending on whether or not the application is distributed). The number of TP systems is growing rapidly as well as their sizes. Today, many TP systems have about 20000 terminals, some of them have more than 100000 terminals and 1000 disks.

What are the load characteristics and the expected performance requirements of such TP systems? During the last years the DEBIT-CREDIT transaction has been accepted as the standard transaction to measure the throughput performance of TP systems in terms of TPS (DEBIT-CREDIT transactions per second). Particularly, banking and airline reservation have always been the fore-runners in high performance DBMS applications with extraordinary service rates; nevertheless, just a few years ago the design of TP systems to support 1KTPS at peak load had been considered a major challenge [Gr85]. Today, several systems can sustain such workloads if the data to be accessed by the transactions can be partitioned in a flexible way (e.g. the dataset ACCOUNT by account number for DEBIT-CREDIT) and, as a consequence, if the mix of transactions can be allocated reasonably well to the data partitions to provide balanced system resource usage. Currently, high performance transaction systems (HPTS) are developed to master a few thousand TPS (e.g. the AMADEUS system for airline reservation). Even higher performance requirements are reported in [HGLW87] for communication switching systems, where transaction loads of > 10^4 TPS must be processed under severe availability constraints. Furthermore, future transactions are anticipated to consume much more computing resources because more complex tasks will be supported (increased number of DB I/Os and instruction path length) and more powerful interfaces and programming languages are used (graphics, 4GL). Another critical, even more performance-determining problem will occur as soon as the DB data cannot be smoothly partitioned anymore according to the data references of various transaction types (many application domains do not provide the data characteristics for "delightful transactions").

The traditional approach to TP systems is the use of large mainframe computers with a centralized DBMS handling the common database. One way to cope with increased performance requirements in such an environment is to design faster computers and/or tightly coupled multiprocessors to accelerate transaction processing (**vertical system's growth**). But even with a speed-up of about 20% per year of the computer hardware, this approach will be hardly successful, since many applications demand a growth rate of more than 50%. Furthermore, availability requirements often prohibit 'single point of failure' solutions and dictate the use of computing systems with some kind of distribution.

Hence, HPTS are typically based on a multisystem approach which offers a much greater potential for satisfying the ever increasing performance, availability and growth requirements (**horizontal system's growth**). From a hardware point of view, they consist of computer systems which can communicate with each other via a high-speed interconnect. They are called loosely coupled if system cooperation is exclusively based on messages, whereas close coupling takes advantages of kind of memory communication to speed up message and data flow across system boundaries (see Fig. 1). Each system has a local memory with an own copy of the OS, DBMS, TP monitor as well as (part of) TAPs. Since single system image and logical view of a common DB have to be continuously provided for all TP applications, appropriate software structures have to guarantee distribution transparency. Depending on how the common database is managed, two general classes of multisystem architectures may be distinguished. In so-called DB-sharing or data sharing systems [Sh85], multiple DBMSs share the DB at the disk level; hence, a single DBMS is able to execute an entire transaction locally. To do so, it possibly has to request resources such as data pages (currently in a 'foreign' system buffer) or locks (managed by a 'remote' lock manager) from another DBMS. As opposed to DB sharing, a DBMS (and the corresponding computer) owns a partition of the disk in a DB-partitioning system and handles all data requests for that disk partition including concurrency control and recovery services. As a consequence, the execution of a transaction may be spread over multiple DBMS by means of function shipping and subtransactions. To achieve atomicity, a distributed 2-phase commit protocol is necessary.

In this paper, we explore the use of non-volatile extended memory to serve as a facility for close system's coupling. Its prime role is to reduce I/O and communication times. Accordingly, we investigate its performance impact for high volume transaction processing and compare horizontal against vertical system's growth. Section 2 discusses the influence of I/O architecture on transaction processing and proposes a new type of memory. Its benefits for (close) multiple systems coupling are described in section 3. We concentrate on DB-sharing systems because they can take maximum advantage from such a memory. The following three sections cover our performance study including simulation and load models as well as simulation results. The final section summarizes our main findings.

2. Influence of I/O Architecture

Availability of sufficient computing resources (e.g. by horizontal or vertical growth) is a necessary prerequisite of high performance transaction processing. Response times, however, should not depend on the degree of parallelism in such transaction systems. Ideally, linear growth of transaction processing power should be achieved while not deteriorating the transaction response times. Such a design goal represents a real challenge from a database point of view since more transactions in the system will cause a higher degree of data contention (by locking protocols). More waiting transactions, on the other hand, imply the increase of the multiprogramming level in order to reach a given throughput goal. But, in turn, these additional transactions aggravate the data contention problem. Indeed, there seems to be a negative feedback in such a 'concurrency/throughput cycle' which may not be broken by the sheer CPU speed. A dominating role is played by transaction duration and resource allocation, since serializability of transactions requires strict 2P-locking protocols. Hence, a substantial reduction of lock contention may be anticipated by reducing the lock duration, that is, by minimizing the service time of transactions using better adjusted I/O-architectures.

Let us discuss the role of I/O by means of the DEBIT-CREDIT transaction with a database characterized by the following record types [An85]:

- ACCOUNT 10^7 records, direct access
- BRANCH 10^3 records, direct access
- TELLER 10^4 records, direct access
- HISTORY $20 * 10^7$ records/month, sequential access.

A DEBIT-CREDIT transaction performs the following actions:

- 3 reads and 3 updates (ACCOUNT, BRANCH, TELLER)
- 1 insert (HISTORY)
- 2 modification operations (on system tables)
- logging.

A typical path length of such a transaction is 250000 instructions which corresponds to 25 ms on a 10 MIPS or 10 ms on a 25 MIPS machine. The role of I/O will be discussed using some "extreme cases".

The worst case is an architecture where the DB buffer is too small to exploit locality of reference. Furthermore, page logging with UNDO and REDO information (still typical for many systems) is assumed. Given 15 ms for a sequential log write and 30 ms for other I/Os, we obtain the following I/O times:

- 6 reads and 6 writes for data and system pages: 360 ms
- 12 writes for log information: 180 ms.

Obviously, such an architecture embodies as dramatic imbalance between CPU and I/O time. To utilize the available CPU power, it is necessary to apply a high multiprogramming level. This fact, in turn, may prohibit smooth performance growth due to potentially long blocking times and their consequences.

The best case for a conventional I/O architecture is the use of a very large DB buffer which enables perfect locality of reference. We assume only 1 read access and the corresponding rewrite to ACCOUNT for transaction processing. Furthermore, we consider a more powerful logging/recovery mechanism (NOSTEAL, NOFORCE [HR83]) such that only REDO information has to be written at COMMIT time:

- 1 read and 1 replacement: 60 ms
- 6 log writes (to non-volatile storage): 90 ms.

It is obvious that page logging causes a major drawback in such an architecture which is amplified by further CPU speed-up. Therefore, the logging performance must be improved, e.g. by using entry logging and group commit. Hence, let us assume that with an optimized logging facility only one log I/O is necessary (15 ms).

Large buffers and optimized logging greatly reduce the I/O times for transaction processing, and as a consequence, the necessary multiprogramming level. According to our evaluation, we may expect as indicative factors for a centralized environment (10 MIPS) a fraction of 25 ms CPU time and 75 ms I/O time. With conventional I/O architectures, there is hardly more room for improvements. For this reason, CPU speed-up (e.g. to 25 MIPS) will have only limited impact on the overall transaction processing time and will enforce increased parallelism to reach satisfactory utilization. In a locally distributed system, the same observations on CPU- as well as I/O-costs apply. Moreover, message overhead will further increase resource allocation times. As a consequence, conventional I/O architectures do not seem to provide additional potential for reducing transaction execution times. Given lock waits are a serious problem, the aggregate I/O times of a transaction embody the lion share of this problem; due to their dominance, they prohibit the effective usage of 'vertical growth'. In a distributed system, we have additionally to consider the aggregate overhead of exchanging messages (e.g. lock requests, commit protocol) and data (e.g. page fetch from a remote DB buffer). Hence, communication network and, again, I/O architecture may limit the possible transaction processing power gained by 'horizontal system growth'.

Therefore, a number of alternative approaches are discussed in the literature. *Main memory DBMS* promise complete reduction of I/O (except for logging). However, they are restricted to the centralized case and incorporate poor cost-effectiveness (e.g. memory utilization [GP87]). Furthermore, they do not allow the use of existing DBMS (new algorithms, tailored storage structures) and may cause serious availability problems (long crash recovery delays). On the other hand, *disk caches or solid state disks* are proposed to reduce the cost for a single I/O. Such devices may have limited use in a distributed context, e.g. they cannot speed-up communication. Due to their channel-oriented interface, they are comparatively slow (~ 2ms) and require asynchronous calls, that is, the calling transaction has to be suspended (expensive process switches). Although they may offer some new quality (assuming non-volatility), we feel that the performance potential gained does not reach far enough for high performance transaction processing.

For these reasons, we advocate as a 'compromise' a new type of I/O and memory architecture supporting fast message exchange as well as safe storage of data pages. A substantial reduction of the transaction's I/O times (and consequently of the blocking times) may be expected by the use of fast non-volatile semi-conductor memory with page-oriented addressing. Given an I/O time of $\leq 50\mu s$ for a 4KB page, logging costs for transaction processing (and rewrite costs of modified pages) become negligible. If such a memory is large enough to act as kind of a global system buffer, we may further save some database I/O from disks, e.g. some ACCOUNT accesses in our reference example.

We call such a memory type a ***global extended memory (GEM)***. It enables direct cooperation of multiple locally allocated computers and guarantees non-volatility as its most important property for transaction processing. (Non-volatility may be achieved by a battery backup or uninterruptable power supply). Cooperation using such a memory with page-oriented access (close coupling) yields greater isolation than with shared main memory (tight coupling). We postulate the use of a simple interface to permit fast access times. Transaction processing requires frequent accesses to GEM if, for example, global system tables (lock information, DB buffer) are kept in it. Therefore, the calling transaction should not be suspended while the call is serviced, that is, GEM should not have an asynchronous calling interface (e.g. controller service). On the contrary, we require direct control of GEM by the accessing processors (passive storage unit). Fast exchange of data pages and caching of pages used by all computing modules (CM) are primary uses. To implement global data structures (e.g. lock tables), smaller access granules than pages (GEM entries) should additionally be supported. These entries could also be used to speed up the exchange of short messages (see section 3).

The use of a GEM in a DB-sharing system is sketched in Fig. 1. A centralized system could be seen as a special case of the architecture in Fig. 1. To deal with GEM failures, it is assumed that similarly to disk mirroring duplicate data storage in independent GEM storage units is supported.

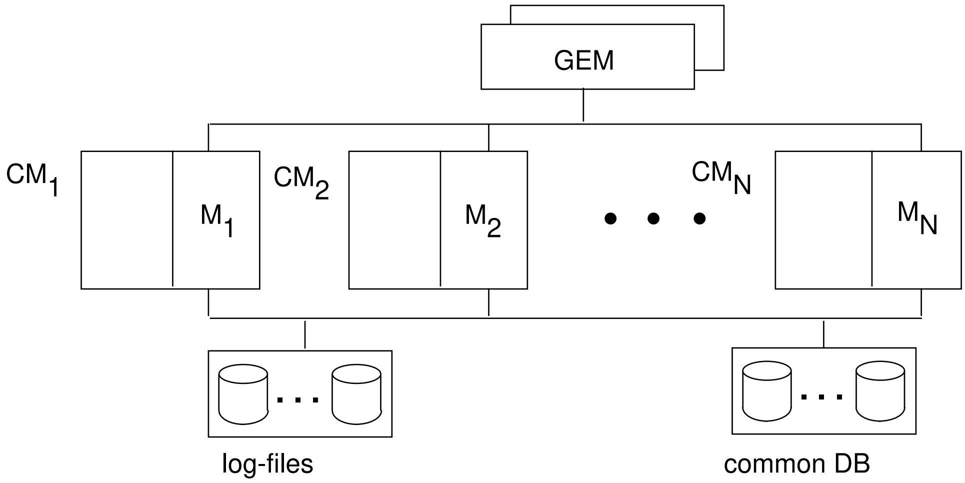

Fig. 1: Architecture of a DB-sharing system using a global extended memory

3. Use of Global Extended Memory in Data Sharing Systems

The most general application of a GEM is to use it for inter-processor communication such that all messages are exchanged across the GEM. In this case, sending and receiving a message basically encompasses three steps:

1.) Write message to GEM

2.) Send interrupt (with GEM location of the message) to receiver processor

3.) Read message from GEM.

If interrupt handling in step 2 is inexpensive compared to the communication overhead associated with traditional multi-layered communication protocols, GEM could substantially improve performance. Naturally, not only data sharing but any locally distributed system could profit from such a fast storage-based communication.

For data sharing, GEM can be utilized in a number of more specific areas. For instance, *GEM entries* may be used to implement global data structures, e.g. for system-wide concurrency control, or for administration of the GEM itself. Furthermore, *GEM pages* allow us to store database pages and logging data in order to speed up the exchange of modified pages between processors and to reduce I/O delay and overhead.

In the following, we briefly sketch different forms of GEM usage for concurrency control, coherency control, global buffer management, logging and load control. A discussion of alternatives for GEM administration concludes this section.

Concurrency Control

Global concurrency control is obviously necessary for data 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.) A storage-based communication with the GEM could help solve these problems and limit the communication overhead. This would reduce the influence of inter-processor communications and therefore the dependency on the chosen algorithm or workload profile.

Another possibility is to store a global lock table in GEM that can be accessed by all processors. 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 an entry of the global lock table requires (at least) two GEM accesses: one to read the entry into main memory and another one to write the modified value back to the GEM. For coordinating these GEM accesses, a special Compare&Swap instruction may be used to write back a modified entry. If the entry's value has been changed by another processor since it has been read from GEM, the Compare&Swap instruction fails and the lock request must be processed again. It is to be noted, that it is possible to keep a reduced lock information of fixed length in GEM by recording lock ownerships/requests on a per-processor basis rather than for individual transactions. Details of such a scheme, however, are beyond the scope of this paper.

If GEM entries can be accessed in a few microseconds, overhead and delay for global concurrency control would almost be negligible compared to message-based concurrency control protocols in loosely coupled systems.

Coherence Control

Coherence control has to deal with the so-called buffer invalidation problem. This problem is to be addressed in data sharing systems since every node maintains a (local) database buffer in main memory to cache pages from the shared database. Thus, modification of a page in one buffer makes all copies of that page in other buffers (and on disk) obsolete. Coherence control has to make sure that these buffer invalidations are either avoided or detected and that all transactions get access to the current page versions.

Fortunately, it is possible to detect buffer invalidations with no extra communication by using extended lock information (e.g. sequence numbers that are incremented by every page modification) [Ra86]. If we use a global lock table in the GEM for concurrency control, coherence control can also be accomplished by recording the additional information for modified pages in this table. Furthermore, modified pages can be exchanged between processors very fast across GEM, irrespective of whether we employ message-based concurrency control or a global lock table in GEM.

Global Database Buffer

In addition to a local database buffer (LDB) in each processor, it is desirable to maintain a global database buffer (GDB) in GEM. If a page is to be replaced from a LDB, it is written to the GDB for possible rereferences by any processor. This can reduce I/O overhead and delay for disk reads since every GDB hit avoids a disk access. Write I/Os benefit from a GDB even more since replacing a modified page from the LDB only incurs a GEM write rather than a disk write. If a page is modified in multiple processors, many modifications can be accumulated per page before it is eventually written to disk.

An important design decision in the context of a GDB is the choice of the strategy for update propagation. As in traditional database systems with two main levels of storage (main memory, disk), we can here distinguish between a FORCE and NOFORCE strategy [HR83]. In our case, FORCE means that all pages modified by a transaction must be written to the non-volatile GDB before commit, while NOFORCE does not require such a force-out out of modified pages. Thus, NOFORCE permits a reduced access frequency to the GDB that could be a potential problem for FORCE at high transaction rates. In addition, GDB hit ratios are likely to be lower for FORCE. This is because the higher write frequency to the GDB requires more pages to be replaced so that the average duration a page can reside in the GDB is shorter than for NOFORCE. Replacing modified pages from the GDB is also expensive since pages cannot directly be written from GEM to disk (Fig. 1). On the other hand, NOFORCE requires to redo modifications lost by a processor crash and complicates coherence control since pages in the GDB may be obsolete, too. The most recent version of a modified page may have to be requested from the processor where the latest modification has been performed (the name of this processor can also be stored in the global lock table).

Logging

Due to the non-volatility of GEM and its significantly faster access time compared to disks, logging is a prime candidate for use of a GEM. The GEM can be used to hold the local log files of every processor as well as a global log file where the modifications of all processors are stored in chronological order. The fast access time permits a simple construction of the global log file without creating a throughput bottleneck. This can be achieved by holding the address of the current end of the global log file in a special GEM entry. At the end of an update transaction, before the locks are released, this entry is read and incremented by the number of pages needed for the transaction's after-images. Only this increment operation needs to be serialized among the processors so that a bottleneck is avoided even for thousands of transactions per second. The log data itself can asynchronously be written to the reserved GEM pages of the global log file. However, the limited GEM size makes it necessary to permanently write the global log data from the GEM to slower backup devices.

Load Control

To effectively utilize the capacity of a distributed transaction system like a data sharing complex, it is important to employ dynamic load control policies that supervise the state of the system and distribute the workload according to current state conditions (e.g. CPU utilization or lock ownerships). If multiple processors take part in such a load control, they could maintain global data structures on the system state or routing strategy in GEM. Message queues are also candidates for storage in GEM since an additional message logging could be avoided (due to the non-volatility of GEM) and because messages could quickly be transferred between different processors.

Administration of the GEM

Since the GEM offers only a simple interface, the 'users' of the GEM, i.e. software components of the accessing processors (operating system, DBMS), are responsible for administration of GEM data. In particular, they have to agree on where to store different types of data (tables, database pages, logging data) and how to organize and use data structures and page areas in GEM. For instance, if GEM is to be used for inter-processor communication, it has to be determined at which GEM location a 'message' can be written without overwriting data still needed by another node. Similarly, the management of a GDB requires free space management and a replacement policy that should consider GEM accesses of all processors.

One approach is to use global GEM data structures for this purpose, e.g. bit vectors to indicate whether or not corresponding GEM pages can be overwritten or hash tables to determine whether a particular database page resides in the GDB. As pointed out above, GEM entries can also be used to coordinate accesses to the global log file. Of course, the use of this administration data results in an increased number of GEM accesses. In addition, GEM entries are not well suited to realize complex data structures of variable length, e.g. as needed for LRU-like replacement strategies. The alternative is to use a *partitioned approach* where the GEM is partitioned among the processors such that write accesses are limited to the partition owner while read accesses are possible by any system. This permits that each partition can partly be controlled using main memory data structures of the partition owner (e.g. for free place management). On the other hand, the realization of global functions may be restricted or require additional inter-processor communication. In the case of message exchange via GEM, it may thus be necessary that the receiving processor sends an acknowledgment to the sender after it has read the 'message' from GEM to indicate that the corresponding GEM location can be overwritten again. A partitioned administration of a GDB is also difficult since the partition owner is not aware of other systems' read accesses to its pages. A restricted form of a GDB would be to use a GDB partition solely as a (non-volatile) GEM extension of the LDB and to speed up the explicit exchange of pages between nodes.

4. Simulation Model

Our simulation study is the first step in a project that aims at analyzing the performance impact of new storage architectures for high-end transaction processing. In this paper, we investigate how the use of non-volatile extended memory like GEM affects performance in centralized DBMS and in closely coupled data-sharing systems. For centralized systems, we investigate potential throughput limitations for vertical growth, i.e. the use of faster CPUs and larger main memory for transaction processing. Horizontal growth aspects are then analyzed by studying data-sharing configurations with a varying number of nodes. Simulation results will be presented for a synthetic DEBIT-CREDIT workload as well as for a real-life DB/DC application represented by a database trace.

The simulation study is based on a detailed simulation system that was developed for loosely coupled data-sharing complexes [Ra88]. This system has been extended so that configurations using a GEM can also be analyzed. In [Ra88], the original simulation system was used to compare the performance of different protocols for concurrency and coherency control for data sharing. The best results were observed for a primary copy locking protocol that solves the buffer invalidation problem in an integrated way to avoid extra messages for coherency control. This protocol will also be used in the data-sharing configurations studied in this paper.

The rest of this section is organized as follows. We start with providing some information on the workloads for which simulation results will be presented. It follows a brief description of the structure and realization of our simulation system together with the used parameters.

Workload Characteristics

Our simulation system can use either synthetic workloads or traces from real database applications. In both cases, the execution of a transaction consists of BOT processing, a number of page references and EOT processing. Trace information includes the transaction type for BOT records, and the page identifier and access mode (read or write) for page references.

Though we have conducted simulation runs with traces of six different transaction loads, we can only discuss results for one application (due to space limitations) and contrast them with results for a synthetic DEBIT-CREDIT load. The chosen trace, named DOA, represents the load of a real-life DB/DC-application and consists of about 17,500 transactions and more than one million page references. The condensed trace data occupies more than 90 MB disk space; on average, a single simulation run takes approximately 10,000 CPU seconds on a 15 MIPS mainframe.

The DOA workload is dominated by read accesses. Though 18 % of the transactions modify the database, merely 1.6 % of all page accesses are writes. On average, a transaction references 58 pages, but 65 % of the transactions access less than 30 pages. 50 % of all page references are done by transactions that access more than 200 pages (4 % of the transactions). One transaction (an ad-hoc query) performs more than 11,000 page accesses. Database accesses are spread over 13 database areas (files) and a total of about 66,000 different pages (this corresponds to approximately 8.5 % of the total database size). The reference matrix in Fig. 2 depicts the access distribution of the 12 transaction types against the 13 database areas. The table shows that the major areas are accessed by almost every transaction type and that the important transaction types access all major areas. This means that in the distributed case the workload cannot generally be assigned such that transactions of different nodes operate on disjoint database partitions.

For the *DEBIT-CREDIT workload*, transactions are generated synthetically according to the benchmark definition in [An85]. This workload is completely homogeneous and consists of update transactions only. In our simulation, every transaction performs 9 page accesses (read and write access to one ACCOUNT record, TELLER record and BRANCH record, write access on HISTORY, and 2 write accesses to system tables). Accesses to the ACCOUNT, BRANCH and TELLER record types are uniformly distributed with the exception that K % (parameter) of the transactions access an account of a non-local branch (in [An85], K = 15 %). The following database sizes were used in the simulation::

- 10 million ACCOUNT records (625,000 pages)
- 10,000 TELLER records,
- 1,000 BRANCH records (1000 pages for TELLER and BRANCH).
- 1,000 pages system tables.

The page size is assumed to be 2 KB. TELLER records are clustered with their associated BRANCH record into the same page (10 tellers per branch). The size of the HISTORY record type is immaterial for our simulations. We assume an implementation that avoids a hot spot for HISTORY by reserving as many HISTORY pages as there are concurrent transactions (multiprogramming level). Concurrent transactions always insert their records into different pages so that lock conflicts for HISTORY are avoided.

Structure of the Simulation System

The simulation system has been implemented in PL/I and employs discrete event simulation. It models centralized transaction processing as well as data-sharing systems with an arbitrary number of nodes (parameter). The (primary copy) protocol for concurrency and coherency control (see below) as well as local buffer management and logging have been completely implemented. Buffer management and logging is based on the DB-cache approach [EB84] that employs a NOFORCE strategy and uses a sequential log file to speed up logging. For logging, a separate log buffer is maintained for each processor to hold the after-images of modified pages. A log buffer is written out (by one sequential I/O) when it is full or at commit time of an update transaction. LRU is employed for page replacement. With loose coupling, modified pages are directly transferred between the nodes across the communication system rather than across the shared disks. Affinity-based transaction routing, controlled by routing tables, can be used to distribute the workload among the processors. Such a routing aims at improving locality of reference, and thus reducing the frequency of I/Os and buffer invalidations, by assigning transaction types with affinity to the same database portions to the same node.

Fig. 2.: Reference matrix of DOA transaction load

In configurations using GEM, this store is used for inter-processor communication, for logging and as a global database buffer (in the centralized case as an extended database buffer). For data sharing, concurrency and coherency control is still based on the primary copy approach; GEM is only used for a faster message exchange. Contrary to the protocols for concurrency/coherency control and local buffer management, we did not explicitly code the administration of the GEM. Rather we assumed that these functions can be realized along the lines of section 3 and considered only the delay and CPU cost for GEM accesses. For global buffer management, we further assumed that the entire database is GEM-resident (unlimited GEM size). In this respect, the simulation results can be viewed as best-case results since no disk I/Os occur in configurations using GEM.

Table 1 shows the main simulation parameters together with their settings. Parameters varied for the two transaction loads include the use of GEM, the number of nodes, the multiprogramming level, CPU speed, and buffer size.

CPU, Communication, and I/O Costs

We simulate a single CPU server per node and distinguish 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 disk I/O (disk read or write, log I/O) or GEM accesses. The remaining CPU requests for transaction processing have lowest priority. The number of instructions per request type are specified as parameters. The costs for transaction processing are modeled by requesting a certain number of instructions for every page reference and for BOT and EOT processing. Page references, BOT and EOT events are collectively referred to as *units of processing (UP)*. For instance, a DEBIT-CREDIT transaction consists of 11 UP in our model (BOT, 9 page references, EOT). The average transaction path-length, excluding overhead for I/O and communication, is thus 11 * 23,000 (#instructions per UP). The UP cost for DOA has been chosen considerably smaller since this workload primarily performs read accesses.

Parameter	Settings
workload	DOA, DEBIT-CREDIT
number of nodes N	1, 2, 4
multiprogramming level P	1 - 600
CPU capacity per processor	10, 25, 100 MIPS
Use of GEM	no, yes
buffer size	1024 , 2048, 4096 pages
GEM size	unlimited
disk I/O delay	30 - 60 ms
GEM access time (page)	25 µs
GEM access time (entry)	2 µs
log buffer size	16 pages
log buffer write time (to disk)	9 - 20 ms
MAX-WAIT	1 - 10 s
K (for DEBIT-CREDIT load, see text)	15 %
transmission rate	3 MB/s
message length	100 B (+ page size)
#instructions per UP	2500 (DOA), 23,000 (DEBIT-CREDIT)
#instructions per disk I/O	2500
#instructions per send	2500 (500 for GEM communication)
#instructions per receive	2500 (500 for GEM communication)
#instructions for processing one message	1000

Table 1: Simulation parameters

The loosely coupled data-sharing configurations use a point-to-point connection between any two nodes. Communication costs are represented by CPU overhead for sending, receiving and processing messages as well as communication delays for the transmission over the network. Every point-to-point connection is modeled as a separate server; the net transmission time is calculated from the message length and the bandwidth parameters. Message transmission over the GEM is modeled by two GEM accesses (write access by sender, read access by receiver node) and a reduced communication overhead for the interrupt notification. For page transfer messages the GEM access time per page is used, otherwise (short messages) the GEM access time per entry.

I/O costs are represented by CPU overhead and I/O delay for every I/O operation. I/O (disk) servers have not been explicitly modeled, assuming that bottleneck situations can be prevented by a sufficiently large number of disk drives. In contrast to disk I/Os, GEM accesses are synchronous, i.e. the CPU is kept busy during the access. This limits the number of concurrent GEM accesses to 1 per CPU.

Modeling of Transaction Processing

For each of the N nodes, a fixed multiprogramming level P is applied indicating 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 trace (or synthetic load generator) in chronological order. The processing of a reference record, in turn, depends on the concurrency and coherency control protocol and the current system state (e.g. whether or not a lock conflict occurs or a page can be 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 (commit processing) finishes the processing of a transaction and starts the next transaction from the trace (closed model). The assignment of transactions to nodes can be controlled by a routing strategy (see below).

Concurrency and Coherency Control

As mentioned above, the primary copy approach [Ra86] is employed for concurrency and coherency control. 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. 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 PCA for the respective partition. Workload and PCA allocation should be coordinated to minimize the number of remote lock requests (but without sacrificing load balancing). For *coherency control* an on-request invalidation (check-on-access) scheme is applied. It uses extended information (sequence numbers) in the lock table which allow the PCA lock manager to decide upon the validity of a buffer page together with the lock request processing. Since the exchange of modified pages can also be combined with regular concurrency control messages, coherency control can be achieved without extra messages. For more details on the primary copy approach, the reader is referred to [Ra86, Ra88].

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).

Concurrency control takes place at the page level (due to the integrated treatment of buffer invalidations). Furthermore, we support level-2-consistency rather than serializability (level 3) to reduce the lock conflict probability. Level-2-consistency is used in most practical database applications to reduce lock contention by releasing read locks before EOT ('short' read locks).

Workload and PCA Allocation

For our simulations, a routing table can be used to control workload allocation. The routing table specifies for every node which transaction types it may process and aims at supporting node-specific locality of reference (affinity-based routing) as well as load balancing. Dominating transaction types may be splitted among multiple nodes to achieve load balancing. For the DOA workload, iterative heuristics (using the reference matrix and number of nodes as input parameters) were used to determine suitable PCA and load allocations [Ra88]. For the DEBIT-CREDIT load, the ACCOUNT, BRANCH and TELLER records are equally partitioned among the N nodes. Transactions are assigned to the node holding the respective BRANCH and TELLER records (only for ACCOUNT, remote lock requests may be necessary).

5. Simulation Results for the Central Case

This section and section 6 will analyse simulation results for the two workloads (DEBIT-CREDIT, DOA) and parameter settings from Table 1. While the next section concentrates on the distributed (data-sharing) configurations, we start here with the centralized case (one CPU). The main focus of this section is to study the impact of GEM usage, CPU speed and main memory buffer size on performance, particularly on throughput.

Results for DEBIT-CREDIT Workload

Fig. 3 plots the transaction rates of DEBIT-CREDIT transactions against the multiprogramming level P for three different CPU speeds (10, 25, 100 MIPS), three buffer sizes (1024, 2048, 4096 pages), and with or without GEM usage. In the GEM configurations, the buffer size did not significantly influence performance so that only results for one buffer size (4096 pages) are shown in the diagrams.
For all CPU speeds and buffer sizes, the best transaction rates were observed for the configurations using the GEM. Without GEM, the peak transaction rates of the GEM configurations could almost be attained, however at much higher multiprogramming levels. Peak throughput grows linearly with CPU speed from 39 TPS for 10 MIPS (Fig. 3a) to about 380 TPS for 100 MIPS (Fig. 3c). Given an average path-length of 253,000 instructions per transaction (11*23.000), this corresponds to an effective CPU utilization of close to 100 % in the best cases. The low conflict potential of the DEBIT-CREDIT load (uniformly distributed database accesses) facilitated such a high CPU utilization for all CPU speeds and buffer sizes.

In the configurations using GEM, a CPU utilization of 100 % was already achieved in single user mode (P=1). This was because of our assumption of GEM-resident databases, so that not only all log I/Os but also all buffer misses were absorbed by the GEM. Since GEM accesses are synchronous, i.e. the CPU is kept busy during the GEM 'I/O', there is no need to overlap I/O delays by increasing the multiprogramming level. However, even when we increased the multiprogramming level P, the peak throughput could be sustained (at the expense of a P-fold response time) due to the negligible lock contention.

Since we assumed a fixed (average) GEM access time of 25 µs per page, the *overhead for GEM accesses increases with the CPU speed*. Holding the CPU for 25 µs corresponds to 250 instructions for 10 MIPS, but already 2500 instructions for 100 MIPS. For instance, GEM accesses consumed 7.3 % of the CPU capacity for 100 MIPS and buffer size 1024, leaving 'only' 92.7 % for effective work (367 TPS). For a 10 MIPS CPU, merely 0.7 % of the capacity was consumed by GEM accesses. Since we used 2500 instructions overhead per disk I/O, the usage of the GEM did no longer result in a reduced I/O overhead for 100 MIPS. This explains why the differences in the peak transaction rates between configurations with and without GEM shrink with increasing CPU speed (Fig. 3). Furthermore, it underlines that synchronous (GEM) accesses make sense only as long as the corresponding CPU overhead is smaller than that of a process switch.

Fig. 3: Throughput results for the DEBIT-CREDIT workload (central case)

Without GEM, the multiprogramming level needed to fully utilize the CPU was much higher than in the GEM configurations and is mainly determined by the workload, buffer size, and CPU speed. A basic observation is that for a given buffer size, *higher multiprogramming levels have to be applied to utilize faster processors*. For the DEBIT-CREDIT workload, for instance, a multiprogramming level of 10 was sufficient to utilize a 10 MIPS CPU, while we needed up to 100 concurrent transactions to utilize the 100 MIPS CPU (Fig. 3). Thus, when we increase the CPU speed by a certain factor, the multiprogramming level must also be raised by this factor if the I/O delay per transaction remains unchanged. For workloads with a higher conflict potential than DEBIT-CREDIT, such an increase of concurrency could result in serious lock contention levels that prevent utilization of fast CPUs.

To some extent, larger database buffers permit a full CPU utilization at lower multiprogramming levels. For instance, a multiprogramming level of 50 was sufficient to utilize a 100 MIPS CPU for buffer size 4096, while we needed 75 and 100 concurrent transactions for buffer sizes 2048 and 1024, respectively (Fig. 3c). This was because for buffer size 4096 all database pages except for the ACCOUNT record type could be held in main memory, while this was not the case for the smaller buffers. However, due to the large size of the ACCOUNT record type (625,000 pages) and the uniform access distribution a further increase of the buffer size is of limited effectiveness for the DEBIT-CREDIT load. So even when we double the buffer size to 8192 pages, about 99 % of the ACCOUNT accesses will still cause a disk I/O. Therefore, the I/O delays for ACCOUNT accesses as well as for logging have to be overlapped (by means of a sufficiently high multiprogramming level) even for very large database buffers.

Fig. 4: Response time composition for DEBIT-CREDIT workload (buffer size 4096)

Fig. 4 depicts the average transaction response time and its composition for some configurations with buffer size 4096. The response time fraction 'Tx processing' corresponds to the CPU service time per transaction and is therefore determined by the CPU speed (25.3 ms for 10 MIPS, 2.53 ms for 100 MIPS). The I/O portion includes not only the disk access time, but also the CPU service time for the I/O operations (or GEM accesses).

As expected, the GEM configurations (P=1) clearly permitted the best response times due to the fast GEM accesses and avoidance of disk I/O. While the total delay for GEM accesses was merely 0.2 ms per transaction, the I/O delay was about 94 ms, irrespective of the CPU speed (10 or 100 MIPS). This I/O delay corresponds to one log I/O and about two disk I/Os. In our simulation, an ACCOUNT access generally causes two I/Os: one to read the corresponding page from disk and one to write back a modified page that has been selected for replacement. (For simplicity, we did not implement an asynchronous replacement strategy. Since there are only modified pages in the buffer for the DEBIT-CREDIT workload, every disk read caused two I/Os.). The large I/O delay is responsible for the fact that response times for the 100 MIPS configurations are not much shorter than for 10 MIPS. Lock waits did not have a significant impact. The largest delay was observed for P=100 (100 MIPS) with an average of 0.04 lock conflicts per transaction and a mean wait of 44 ms. This resulted into an average of 1.8 ms lock delay per transaction.

Simulation Results for DOA

Fig. 5 shows the throughput results for our real-life workload DOA for three CPU speeds (10, 25, 100 MIPS) and two buffer sizes (2048, 4096 pages). The results obtained with the GEM are also plotted, however only for P=1 (single dots on the left side). Since there are different transaction types in DOA, we measure throughput in 'UPs per second (UPS)' rather than transactions per second. On average, a transaction encompasses 60 UPs for the DOA workload (BOT, 58 page accesses, EOT), but transaction size varies significantly (see above).

The curves show a similar behavior than for the DEBIT-CREDIT load and confirm the observations discussed above. First of all, the GEM configurations achieved the best results at the lowest multiprogramming level. Second, we had to raise the multiprogramming level proportionally with the CPU speed to achieve a satisfying utilization for the disk-based configurations. Larger buffers can reduce the required multiprogramming level by cutting the I/O delays per transaction. The peak throughput of the GEM configurations could be approached by the disk-based configurations at high multiprogramming levels indicating that lock contention was not a limiting factor (because of the dominance of read accesses). As for the DEBIT-CREDIT load, the throughput differences between the configurations with and without GEM shrink with increasing CPU speed.

Fig. 5: Throughput results for DOA (central case)

The UP cost for DOA has been chosen considerably smaller than for DEBIT-CREDIT to increase the effect of I/O overhead. One consequence of this is that we had to apply much higher multiprogramming levels in order to overlap I/O delays. So, even 600 concurrent transactions could only achieve a CPU utilization of 89 % (including 19 % I/O overhead) for 100 MIPS and buffer size 2048. The pronounced role of the I/O overhead also increased the differences between the results for different buffer sizes, particularly in the case of 100 MIPS. Similarly, the throughput differences between the GEM configurations and disk-based counterparts were greater, particularly for slower CPU speeds (recall, that for 100 MIPS a disk I/O and a GEM access incur the same overhead). So, for 10 MIPS we achieved an effective CPU utilization of 98.6 % using the GEM (3944 UPS), compared to 87.3 % (77.7 %) in the disk-based configuration and buffer size 4096 (2024). Thus, usage of the GEM increased the peak throughput by 13 % (27 %) for 10 MIPS. For 100 MIPS, the peak throughput was 35,058 UPS (87.6 % effective CPU utilization) using the GEM, and 34,264 UPS (85.7 %) without GEM.

Despite the dominance of read accesses and the usage of short read locks, lock conflicts and deadlocks were more frequent than for the DEBIT-CREDIT load. Though they did not limit the maximal throughput, up to 25 %

of the average response time was caused by lock waits (for P=400 and 600). The CPU consumption of transactions that were aborted because of deadlock (wasted work) was up to 2 %.

Discussion

In the central case, GEM is solely used to improve the I/O behavior, i.e. to cut the I/O delays for disk accesses for logging and database accesses. As our simulation results have shown, peak transaction rates can then be attained at very low multiprogramming levels and very short response times. Of course, even with GEM multiple transactions will have to be executed concurrently, in general, e.g. to utilize multiprocessors or to overlap think times for conversational transactions. In addition, disk I/Os are still possible if the database is not completely GEM-resident, but when GEM is primarily used to eliminate (synchronous) disk writes and to cache only the most recently used database portion not kept in main memory.

With faster CPUs, the slow disk access times increasingly dominate transaction response time and require high multiprogramming levels to achieve a high CPU utilization. Though high multiprogramming levels did not cause performance (throughput) problems for our loads, this cannot generally be expected. This is because lock contention increases with the number of concurrent transactions so that a thrashing behavior sets in at some point and throughput starts to decrease [Ta85, ACL87]. The probability that this multiprogramming limit is reached before the CPU can be fully utilized increases with CPU speed thus potentially limiting vertical growth. In [FRT90], simulation results are presented for high contention environments where two-phase locking fails to utilize fast processors. The authors recommend alternative (optimistic) concurrency control methods that rely on transaction aborts. Though abortions can result in a significant amount of wasted work, they found this more affordable than under-utilizing fast processors. Other authors [BBD82, ACL87] recommend to dynamically adapt the multiprogramming level in order to control lock contention. A secondary effect is that the internal overhead in the DBMS and operating system tends to increase with the number of concurrent users, particularly if every transaction runs in its own address space.

Using the GEM, a high CPU utilization can be obtained at low multiprogramming levels. This significantly reduces the concurrency control problem and supports vertical growth even for loads that otherwise cause high data contention. High performance is thus attainable without sacrificing serializability (e.g. by using short read locks) or implementing new concurrency control algorithms and dynamically controlling the multiprogramming level. Furthermore, coarser concurrency control granules can be chosen than in disk-based environments, e.g. page-level instead of record-level concurrency control. This reduces the lock overhead (fewer lock requests, smaller lock tables) and may simplify the lock manager implementation. Record-level concurrency control is particularly difficult for data sharing because of the buffer invalidation problem (see [Ra88]).

6. Simulation Results for the Distributed Case

This section concentrates on the distributed (data-sharing) configurations to study the impact of GEM usage on horizontal growth characteristics. Our performance comparison of the loosely and closely coupled configurations will evaluate the role of inter-processor communication as well as the I/O and locking behavior.

Results for DEBIT-CREDIT

Fig. 6 depicts the DEBIT-CREDIT transaction rates for 1, 2 and 4 nodes (N) for 25 MIPS CPUs and a buffer size of 2048 pages per node. Results are shown for multiprogramming level 1, 10 and 25 (per node).

Fig. 6: Throughput results for DEBIT-CREDIT

We observe that the GEM configurations reached the best transaction rates for the distributed configurations, too. Moreover, similar to the central case the peak throughput was almost achieved for P=1 already and could be sustained for higher multiprogramming levels. The loosely coupled configurations required a multiprogramming level of 25 per node to approach the peak transaction rates. Horizontal growth was perfect for DEBIT-CREDIT since throughput could be linearly increased with the number of nodes. This was because not only lock contention but also communication overhead was almost negligible for the DEBIT-CREDIT load. As expected, the share of transactions accessing a remotely controlled ACCOUNT record was about K * (N-1)/N, i.e. the number of global lock requests per transaction was 0.075 for two nodes and 0.11 for four nodes (K = 15 %). The corresponding communication overhead consumed merely 0.4 % of the CPU capacity for N=2 and 0.7 % for N=4 in the loosely coupled configurations. Communication using GEM permitted an even lower communication overhead (0.3 % for N=4).

In the GEM configurations, CPU utilization for P=1 was slightly below 100 % for data sharing since the CPU is idle while a remote lock request is processed by the responsible PCA node. However, a multiprogramming level of 2 per node was already sufficient to overlap these delays and achieve 100 % utilization. For the loosely coupled configurations, it is interesting to note that throughput increases super-linearly for lower multiprogramming levels. For P=10, for instance, we had a transaction rate of 68 TPS in the centralized case compared to 160 TPS for N=2 (factor 2.35) and 356 TPS for N=4 (factor 5.2). This was because we left the database size unchanged but increased the aggregate buffer size proportionally with N (N * 2048 pages). Hence, the I/O delay per transaction dropped with growing N and permitted a higher CPU utilization for a given multiprogramming level. However, this effect was limited to multiprogramming levels where CPU utilization can still be improved; peak throughput (P=25) did not increase super-linearly.

Simulation Results for DOA

Fig. 7 plots the throughput results for the DOA workload (in UPS) for 1 to 4 nodes, 25 MIPS per CPU and buffer size 2048. For the central case, which was I/O bound for buffer size 2048, the results for buffer size 4096 have also been included. The multiprogramming levels were differently chosen for GEM configurations and disk-based/loosely coupled configurations. The results for the GEM configurations are shown for P=1, P=10 and P=25, while the other results refer to multiprogramming levels 25, 50 and 100.

Fig. 7: Throughput results for DOA

In contrast to the previous experiments, the GEM configurations clearly outperform the other configurations in terms of peak throughput. For two nodes, using the GEM for communication and I/O improved peak throughput by 27 % compared to the loosely coupled configurations, and for four nodes even by 34 %. As in the other experiments, very low multiprogramming levels were sufficient for GEM configurations to fully utilize all CPUs, while in the loosely coupled data sharing configurations multiprogramming level 100 was required to overlap the delays for I/O and communication. With loose coupling, the communication overhead (see below) significantly limited horizontal growth. So, throughput could only be improved by a factor 1.62 for N=2 and a factor 3.0 for N=4 compared to the central case (without GEM, buffer size 4096). Even with GEM, throughput was degraded to some extent by communication overhead, but to a far smaller degree. Compared to the GEM-based configuration for the central case, we obtained a throughput improvement of factor 1.8 for N=2 and 3.52 for N=4. Compared to the central case with disk-based I/O, the closely coupled data-sharing configurations achieved super-linear throughput improvements.

An analysis of the CPU utilization helps to explain these results. Fig. 8 shows the composition of the average CPU utilization for the configurations with and without GEM for 1, 2 and 4 nodes and buffer size 2048. The results are shown for multiprogramming level 25, 50 and 100, except for GEM-based configurations that refer to

P=1. The total utilization is composed of the effective CPU utilization ('Tx processing') and the overhead for I/O and communication.

Fig. 8: CPU utilization for DOA (25 MIPS, buffer size 2048)

We observe that the GEM configurations achieved the highest effective CPU utilization (even for P=1) and thus the best throughput. Remote lock requests caused a CPU utilization of less than 100 % in the closely coupled configurations for P=1. (To overlap these delays a low multiprogramming level was sufficient; the peak throughput in Fig. 7 was obtained for P=10 where CPU utilization was at 100 %). The high effective CPU utilization was made possible because communication and I/O overhead was significantly lower than without GEM, despite the fact that about the same number of global lock requests and buffer misses occurred per transaction. The communication overhead for 4 nodes was almost twice as high as for N=2 since the fraction of remote lock requests increased from 12.9 % to 24.2 %. Despite the coordination of PCA and workload allocation, this increase of global lock requests was unavoidable for the reference behavior of the DOA workload (see reference matrix, Fig. 2). Note that with a hash-based PCA allocation and random workload assignment even 50 % (75 %) remote lock requests have to be anticipated for two (four) nodes.

In configurations without GEM usage, the effective CPU utilization grows with the multiprogramming level until CPU saturation sets in. The overhead for I/O and communication also increases proportionally with the multiprogramming level (since more transactions are ready to issue I/O and lock requests). While the communication overhead grows with N, the opposite is true for the I/O overhead. This is because the increased aggregate buffer size improved hit ratios and partly increased the effective CPU utilization (P=100 was not sufficient in the central case to completely overlap the I/O delays). The affinity-based workload allocation also supported the improved I/O behavior (better locality of reference) in the data sharing configurations and thus a partial compensation of the communication delays. Still, the communication overhead was high and reduced peak throughput considerably for the loosely coupled configurations. As a consequence, linear speed-up (horizontal growth) could not be achieved.

The fact that an average CPU utilization of over 95 % could be achieved indicates that neither lock contention nor load balancing was a problem for the DOA workload (all nodes could be utilized). The comparatively small effect of lock contention, however, was influenced by the fact that we did not consider lock conflicts on system tables, e.g. for free space management and address translation. Usually, lock contention on this type of information is kept low by special implementation techniques and/or record-level locking. In experiments where we did apply page locking on these tables (by setting a special simulation parameter), performance for the disk-based/loosely coupled configurations was disastrous due to lock conflicts on the newly created "hot spots". The GEM-based configurations, however, remained largely unaffected due to the low multiprogramming levels.

Discussion

In the distributed case, GEM is mainly used to reduce the communication overhead. Communication is required for concurrency/coherency control in data sharing systems and should be minimized by means of appropriate algorithms (such as the primary copy approach) and an affinity-based transaction routing. In general, the communication overhead is strongly determined by the workload. DEBIT-CREDIT can be viewed as an ideal case since transactions and data can easily be distributed to keep the communication overhead small and to utilize all processors (load balancing). In real workloads such as DOA, however, an ideal partitioning of data and workload can hardly be achieved leading to an increased communication overhead and less than linear speed-up (horizontal growth). These applications, however, offer an increased optimization potential that can be exploited by using GEM. In fact, GEM helped to reduce the communication overhead considerably for DOA and to achieve an almost linear horizontal growth.

Thus, GEM supports high performance and horizontal growth even for real-life workloads. Dependencies on workload characteristics and on affinity-based routing are reduced facilitating load balancing and the effective utilization of a larger number of nodes. Achieving high transaction rates at low multiprogramming levels reduces the concurrency control problem and guarantees good response times.

7. Conclusions

In this paper, we have investigated the use of non-volatile semiconductor memory for high performance transaction processing. A specific store, called Global Extended Memory (GEM), promises significant performance benefits in centralized and distributed environments over traditional disk-based and loosely coupled architectures. Key characteristics of the GEM are fast, synchronous access by multiple nodes, non-volatility and support of different access granules (pages and entries). Usage of the GEM primarily aims at improving the I/O behavior (logging, fast writes and caching of database pages) and reducing communication overhead and delay. As discussed in section 3, data sharing systems can take advantage of a GEM in a number of critical areas like global concurrency/coherency control and global logging.

An extensive simulation study using a synthetic DEBIT-CREDIT workload as well as real-life database traces has been conducted for a preliminary performance evaluation of GEM usage in centralized DBMS and in data-sharing systems. The main findings of our study can be summarized as follows:

1. Usage of GEM can significantly reduce I/O and communication delays thus supporting very fast response times.
2. The great reduction of I/O and communication delays permits to achieve a high CPU utilization (high transaction rates) at very low multiprogramming levels. This becomes increasingly important with faster CPUs and supports vertical growth. Without GEM usage, the required multiprogramming level and therefore lock contention increases with the CPU speed so that only limited vertical growth may be achievable.
3. Reduced lock contention decreases the need for fine-granularity locking or alternative concurrency control methods that rely on transaction aborts rather than blocking.
4. The reduced communication overhead for storage-based communication via GEM facilitates horizontal growth and load balancing. This is particularly important for real-life workloads for which it is more difficult to find suitable allocations of data and transactions than for DEBIT-CREDIT.
5. Synchronous GEM accesses become more expensive with increasing CPU speed (section 5). Thus, throughput improvements for GEM-based architectures may shrink with faster CPUs unless GEM accesses can also be made faster.

For the future, we plan to further evaluate GEM usage for transaction processing, in particular the role of lock contention. In addition, we will investigate whether concurrency control via global lock tables in GEM can improve performance compared to message-based protocols such as the primary copy approach. Also, the issue of cost-effectiveness needs to be considered.

References

[ACL87] Agrawal, R.; Carey, M.J.; Livny, M.: *Concurrency Control Performance Modeling: Alternatives and Implications*. ACM Transactions on Database Systems, Vol. 12, No. 4, 609-654, 1987.

[An85] Anon et al.: *A Measure of Transaction Processing Power*. Datamation, April 1985, 112-118.

[BBD82] Balter, R.; Berard, P.; DeCitre, P.: *Why Control of the Concurrency Level in Distributed Systems is More Fundamental than Deadlock Management.* Proc 1st Symp. on Principles of Distributed Computing, 183-193, 1982.

[EB84] Elhardt, K.; Bayer, R.: *A Database Cache for High Performance and Fast Restart in Database Systems.* ACM Transactions on Database Systems, Vol. 9, No. 4, 503-525, 1984.

[FRT90] Franaszek, P.A.; Robinson, J.T.; Thomasian, A.: *Access Invariance and its Use in High Contention Environments*. Proc. 6th IEEE Data Engineering Conf., 1990.

[GP87] Gray, J.; Putzolu, F.: *The 5 Minute Rule for Trading Memory for Disk Accesses and the 10 Byte Rule for Trading Memory for CPU Time*. Proc. ACM SIGMOD conf., 395-398, 1987.

[Gr85] Gray, J. et al.: *One Thousand Transactions per Second*. Proc. IEEE Spring CompCon, San Francisco, 96-101, 1985.

[HGLW87] Herman, G.; Gopal, G.; Lee, K.C.; Weinrib, A.: *A Datacycle Architecture for Very High Throughput Database Systems*, Proc. SIGMOD ’87 Conf., San Francisco, CA, 97-103, 1987.

[HR83] Härder, T., Reuter, A.: *Principles of Transaction-Oriented Database Recovery*. ACM Computing Surveys, Vol. 15, No. 4, 287-317, 1983.

[Ra86] Rahm, E.: *Primary Copy Synchronization for DB-Sharing*. Information Systems, Vol. 11, No. 4, 275-286, 1986.

[Ra88] Rahm, E.: *Design and Evaluation of Concurrency and Coherency Control Techniques for Database Sharing Systems.* Technical Report 182/88, Computer Science Dept., Univ. Kaiserslautern, 1988.

[Sh85] Shoens, K. et al.: *The AMOEBA Project*, Proc. IEEE Spring CompCon. 102-105, 1985.

[Ta85] Tay, Y.C.: *Locking Performance in Centralized Databases*. ACM Trans. on Database Systems, Vol. 10, No. 4, 415-462, 1985