To appear in: Proc. 4th Int. Workshop on High Performance Transaction Systems, Asilomar, CA, Sep. 1991, Lecture Notes in Computer Science, Springer-Verlag.

Use of Global Extended Memory for Distributed Transaction Processing

Erhard Rahm

University of Kaiserslautern, Dept. of Computer Science
6750 Kaiserslautern, Germany
E-Mail: rahm@informatik.uni-kl.de

Abstract

In current mainframe architectures, extended memory is used as a page-addressable main memory extension to improve I/O performance. If made non-volatile, extended memory can significantly increase transaction processing performance. In this paper, we study how locally distributed transaction systems may use shared extended memory to support high transaction rates and short response times. We outline how a specific store called GEM (Global Extended Memory) can be utilized to improve I/O performance and inter-system communication and cooperation. It turns out that distributed transaction systems of the "shared disk" type (data sharing systems) can benefit most from such a shared store. We describe simple yet efficient schemes using GEM for global concurrency control and constructing a global log file for such systems.

1 Introduction

Extended memory has first been used in IBM 3090 mainframe computers as a volatile main memory extension [CKB89]. In contrast to main memory, this so-called expanded storage (ES) is page-addressable similar to disks or other semiconductor memories like disk caches and solid-state disks. This means that in order to read data from extended memory, the corresponding page must be read into main memory. Similarly, data cannot directly be modified in extended memory but pages are altered in main memory and written back at a later point in time. This page-oriented access interface enhances failure isolation against processor failures and software errors. I/O performance can greatly benefit from page-addressable semiconductor

memories like extended memory, disk caches and solid-state disks that permit substantially better access times and I/O rates than disks.

Disk caches [Sm85, Gro85, Gro89] and solid-state disks (SSD) [Ku87] support the channel-oriented interface of disks so that their use is transparent to the accessing systems. The access time to such stores is largely determined by the page transfer time (channel speed) and delay at the device controller. Typically, page access times of 1 to 2 ms can be achieved resulting in a 10-fold improvement compared to disks. The I/O overhead, however, is the same than for disk accesses and typically accounts for thousands of instructions per I/O (process switch, etc.).

On the other hand, extended memory has no channel-oriented interface but is largely managed by software in the operating system (e.g. MVS and VM in the case of IBM's ES). Special machine instructions are provided to move pages between main memory and ES. Currently, access times are two to three orders of magnitudes faster than for SSDs and disk caches (about 75 microsec per 4 KB page including OS overhead [CKB89]). Since a process switch would be more expensive than this delay, accesses to ES are *synchronous*, i.e. the CPU is not released during the page transfer. While conceptually the ES sits between main memory and the disk subsystem in the storage hierarchy, pages cannot directly migrate from ES to disk. Rather all data transfers between ES and disk must go through main memory since page transfers are controlled by the accessing system rather than by a separate ES controller. Originally, the ES has only been used as a fast paging and swapping device controlled by the operating system (LRU replacement of pages in ES). Meanwhile more flexible OS services have been provided to permit programs (e.g. the DBMS) to maintain data in ES [Ru89].

The discussion shows that extended memory offers reduced I/O overhead and significantly shorter access times than disk caches or SSDs. On the other hand, the use of extended memory is no longer transparent but must be supported by system software. While the cost per megabyte is about half the cost of main memory, extended memory is about twice as expensive than SSD or disk caches [Ku87]. The cost per megabyte of SSD and disk caches, in turn, is approximately by a factor 50 - 100 higher than for disks [Ra91b]. The relative cost of extended memory is likely to increase if this store is made non-volatile, e.g. by using a battery backup or uninterruptable power supply. Of course, the mentioned cost ratios are only rough indicators and only partially be determined by technical characteristics, but may differ from manufacturer to manufacturer. At any rate, we expect disks (or disk arrays [KGP89]) to remain significantly cheaper than non-volatile semiconductor memories for the foreseeable future (at least for large systems). As a consequence, the majority of data will still be stored on disks and only the use of a limited amount of non-volatile semiconductor memory is likely to prove cost-effective. (For simi-

lar reasons, keeping entire databases resident in main memory is considered too expensive for very large databases of hundreds of gigabytes).

So far, the use of non-volatile semiconductor memories for transaction processing has received little attention in the database research community. Studies on main memory databases frequently assume that the log buffer resides in non-volatile main memory to eliminate a synchronous disk I/O for logging (a transaction is committed as soon as its commit record has been written to the log buffer). Some proposals (e.g. [LR88]) assume that the entire main memory is non-volatile in order to avoid keeping a log file and an archive copy of the database. This approach is not only very costly, but also has the problem that software errors and CPU crashes may compromise the consistency of the memory-resident database. In [CKKS89], the use of a so-called "safe RAM" is proposed to improve I/O performance in centralized database systems. Safe RAM is supposed to be a DRAM memory with enough backup power to copy the memory contents to a disk after a power failure. All write I/Os (database and log writes) should be directed to this store so that database reads remain the only I/O delays for transactions. The authors argue that a comparatively small store is sufficient to significantly improve performance compared to a disk-based architecture. They also provide cost estimates to demonstrate the cost-effectiveness of such an approach.

In [Ra91b], we present a comprehensive performance evaluation that compares the use of non-volatile extended memory, SSD and disk caches for transaction processing. Three usage forms of these storage types to improve I/O performance were considered:

1. *Keeping entire files (database partitions, log file) resident in non-volatile semiconductor memory*
   This usage form is applicable for non-volatile extended memory and SSD. It permits the best I/O performance since all disk reads and writes are avoided for the respective files. The resulting transaction performance was found to be comparable to main memory databases but could be achieved at a lower storage cost.
2. *Use of a write buffer in non-volatile semiconductor memory (extended memory or disk cache)*
   This usage form achieved the second-best performance since (almost) all synchronous disk writes could be eliminated and were transferred into asynchronous ones (deferred propagation of modified pages to disk). This usage form has the advantage that a small buffer was sufficient to greatly improve performance for log and database writes.
3. *Caching of database pages in a second-level page cache*
   This usage form is applicable for volatile and non-volatile extended memory or disk caches and aims at reducing the number of disk reads by caching database pages at an additional level of the storage hierarchy. The second-level cache was found to achieve similar I/O savings than an increased main memory buffer, but at a lower cost. The use of disk caches, however, in addition to main memory caching was little effective since current disk caches are designed for one-level caching (the same pages were frequently cached in main memory and in the disk cache) and only modified pages migrated from the main memory buffer to the disk cache. More effective is the use of an extended database buffer in extended memory directly managed by the DBMS.

Non-volatile extended memory supports all three usage forms at the best performance, but at the highest storage cost. The lower cost of disk caches and SSD can make the combined use of two or even three of the storage types desirable (e.g. SSD for usage form 1, non-volatile disk caches for usage form 2, and extended memory for usage form 3).

In this paper, we study the use of shared, non-volatile extended memory in locally distributed transaction systems (all computing systems reside in close proximity, e.g. in the same machine room). The major goal is to improve performance compared to loosely-coupled systems that solely communicate by means of message passing over a network (communication lines). Since this communication form creates a high CPU overhead in typical mainframe architectures / operating systems, there is a large potential for improvement. A secondary objective is to simplify global coordination tasks by using shared data structures in the global memory. Potential disadvantages / problems include the high cost of such a special store, restricted extensibility (limited number of systems that may be connected to the shared store), and reduced failure isolation compared to loosely coupled systems. On the other hand, message passing over an interconnection network and the use of shared stores are not mutually exclusive, since every node in a loosely coupled system could consist of multiple autonomous processors connected to a shared extended memory.

The design and use of such shared stores poses a variety of challenges and tradeoffs. One critical aspect, for instance, is the design of the access interface. A simple page-oriented interface limits the applicability of the store to the usage forms of the central case aiming at an improved I/O behavior. (Such a store called SSU (System Storage Unit) is already offered by Fujitsu since 1990. The SSU is non-volatile, has a capacity of up to 2 gigabytes and can be connected to four Fujitsu mainframes. The transfer rate between main memory and SSU is 300 MB/s.) In order to utilize a shared store for inter-system cooperation, an extended access interface becomes necessary that defines which functions are offered by the storage controllers and what must be done by the accessing systems. One approach would be to off-load certain database functions (e.g. global locking) to the storage controllers resulting in a comparatively high and specialized access interface. The other extreme which is more compatible with current extended memory would be to choose a simple but more general interface that leaves most of the processing (including administration tasks) to system software of the accessing systems. The store that we consider in this paper adheres to the latter philosophy.

If a general access interface is defined, the next question is which functions can benefit in which way by a shared store and which software component (operating system or subsystem) should implement the function. General functions, e.g. message passing via the shared store, should certainly be provided by operating system (OS) components to make this services available to multiple subsystems and to pro-

vide device independence (that is, existing applications and subsystems can benefit from a fast device like extended memory without program modifications). As we will see, more specialized usage forms may have to be implemented by privileged subsystems (e.g. the DBMS, TP monitors, etc.) thereby avoiding the overhead (supervisor call) for invoking operating system primitives.

In the next section, we describe the main characteristics of the Global Extended Memory (GEM) considered in this paper. In section 3, we discuss how inter-system communication via GEM may be accomplished. The use of GEM in data sharing systems is outlined in section 4. In particular, we describe new schemes for global concurrency control and for creating a global log file. Conclusions appear in section 5.

2 Global Extended Memory

We consider a specific store called *Global Extended Memory (GEM)* which is assumed to be non-volatile and accessible by multiple systems or CMs (computing modules) as indicated in Fig. 1. Such a coupling of systems is referred to as a *close coupling* which aims at combining the advantages of loose and tight coupling. Similar to tightly coupled systems, a shared memory (GEM) is used to permit a more efficient communication and cooperation than with message passing in loosely coupled systems. On the other hand, availability must be better than for tightly coupled systems. As in loosely coupled systems, the individual CMs are therefore autonomous, i.e. they have their own main memory and separate copies of operating system and DBMS to improve failure isolation. In addition, GEM contents cannot directly be manipulated in contrast to shared memory in tightly coupled systems. As for extended memory in a single system, all data transfers between GEM and disks must go through main memory of the connected CMs.

Like in the central case, GEM may be used independently by each CM to improve I/O performance. For instance, every CM could use a dedicated GEM partition for allocating write buffers, managing an extended database cache or to store entire files. These usage forms can be controlled by system software of the respective CM (operating system, DBMS) since only one system is accessing the respective GEM partition.

Apart from these usage forms, GEM should also support efficient inter-system cooperation to reduce the communication overhead. A simple form of cooperation is to permit shared access to GEM-resident files; this can be realized with the page-oriented access interface analogous to file sharing in “shared disk” systems. Such a support for shared GEM-resident files should be provided by the OS file manager, transparent to the DBMS.

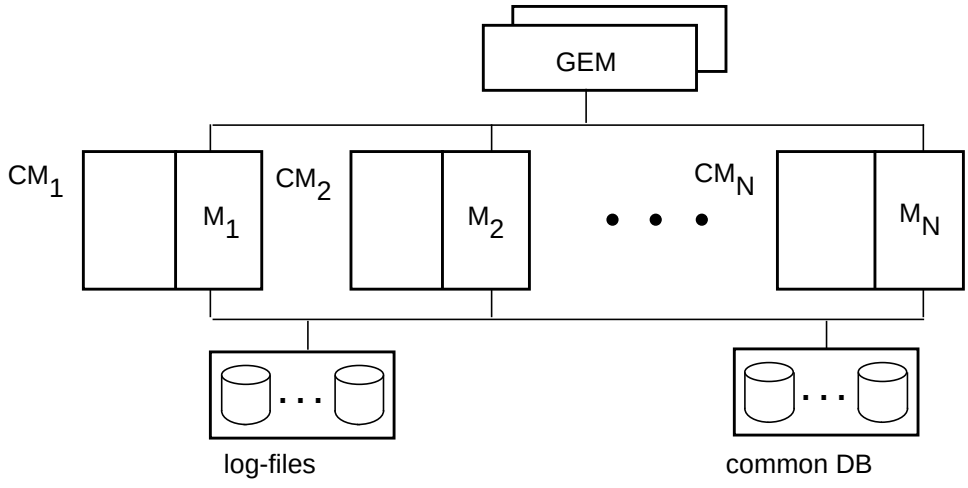


Fig. 1: Architecture of a data sharing system using a Global Extended Memory

For further usage forms like managing shared data structures or exchanging messages via GEM, however, the page-oriented interface is not flexible enough. In particular, concurrent GEM accesses by different systems (e.g. to modify shared data structures) must efficiently be synchronized to avoid lost updates. As mentioned in the introduction, one possibility to make coordinated use of the GEM is to put more functionality into the GEM controllers. For data sharing, these controllers could perform global buffer management, global locking and global logging. On the other hand, such a design would be incompatible with the central case (and other applications than transaction processing) where these special functions are not needed. Furthermore, GEM access times are likely to be higher than with a simple interface so that synchronous accesses may no longer be feasible. We will therefore assume a low-level interface, similar to current extended memories, resulting in a largely passive GEM that is managed by software in the connected CMs. Apart from read and write accesses to GEM pages, we require that smaller granules (*entries*) can also be read or written from/to GEM, e.g. to realize simple data structures. To deal with GEM failures, duplicate data storage in independent GEM storage units should be supported (analogous to disk mirroring). Alternatively, a logging scheme could be used to deal with media (GEM) failures.

Thus the GEM internally consists of multiple page areas and entry areas that may be duplicated if desired. For transaction processing, page areas are used to hold either database or log pages. Entry areas are used to store global data structures or to exchange (short) messages. 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[1]. This low-level interface promises a simple design of the GEM hardware and fast access times (in performance estimates, we will assume about 20

microseconds per page and about 1 microsecond per unit entry). The administration of the page and entry areas and their coordinated use by different systems must be implemented by CM software (OS, DBMS).

In [DIRY89, DDY91], a similar store as the GEM has been considered for use in data sharing systems. They assumed a purely page-oriented intermediate memory managed by a central controller. A central controller was also assumed to provide global concurrency control with little overhead. A performance evaluation based on analytical models is presented that predicts large performance improvements by such a store. In [DDY91], however, queueing delays for accesses to the shared buffer (controller) apparently were not considered. They could be a serious problem for higher transaction rates since every local buffer miss results in a look-up in the global buffer. In addition, every page modification was written to the global buffer at transaction commit also resulting in a very high access frequency to the shared buffer.

3 Communication via GEM

The most general application of a GEM in (locally) distributed systems is to use it for inter-processor communication such that all messages are exchanged across the GEM. Such a fast communication is a general service that can be utilized for distributed transaction processing (either in data partitioning or data sharing systems) as well as for other applications. Hence, the operating system should implement the exchange of messages via GEM. Data partitioning (“shared nothing”) systems do not rely on shared system components but coordinate all activities by message passing. Therefore, in addition to using the GEM for communication extended memory may only be used within each node as in centralized systems (e.g. for keeping local log files GEM-resident or caching pages from the local database partition). For data sharing, on the other hand, the GEM can be utilized to a larger extent. For instance, GEM entries may be used to implement global data structures, e.g. for system-wide concurrency and coherency control. Furthermore, database pages stored in GEM

1. Although we can modify individual double words with the Compare&Swap instruction, GEM still cannot directly be modified like main memory since the modifications need to be performed in main memory before they are written back to GEM.

(either in a shared cache or as part of GEM-resident files) can be accessed by all systems. Finally, a global log file can also be constructed in GEM. In this section we will discuss how communication via GEM may be realized, while section 4 covers the use of GEM in data sharing systems.

In a locally coupled system, the main communication cost is not transfer time but the overhead associated with message passing often requiring tens of thousands of instructions per send or receive operation. To reduce this overhead, streamlined protocols tailored to the local case should be used rather than the traditional multi-layered protocols for general networks[1]. One possibility to reduce the overhead is the use of shared memory like GEM for communication. In this case, sending and receiving a message basically encompasses three steps (Fig. 2):

1.) Write message to GEM
2.) Send interrupt (with GEM location of the message) to receiver CM
3.) Read message from GEM.

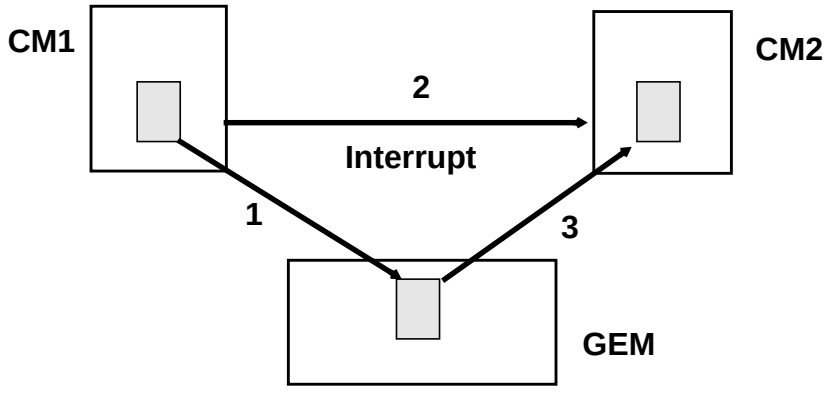


Fig. 2: Message transfer via GEM

The GEM can be used for exchanging messages of arbitrary size. Small messages may be mapped to GEM entries permitting very short access times, while large messages can be written into one or more GEM pages. Although step 2 could be considered as a message transfer by itself, this interrupt is assumed to create comparatively small overhead (e.g. it may be sent by a special instruction).

Apart from the three basic steps, additional administration tasks are necessary to implement a message transfer via GEM. First of all, before a message is written to GEM in step 1 a m*essage buffer* (a sufficiently large number of entries or pages) that can be overwritten must be allocated in GEM. Secondly, while the message is "in transit" it has to be ensured that it is not overwritten by another CM. Finally,

1. In fact, there is a controversy about whether or not optimized protocols for the local case should be supported since hardware (CPU, network) is getting faster at a high rate. However, as long as the ratio between communication overhead and useful work does not improve, even the fastest CPUs cannot as effectively be utilized than with streamlined protocols (=> lower throughput).

after the message has been read by the destination CM in step 3, the message buffer is to be released.

For administration tasks like these, basically two approaches can be chosen. With a *partitioned administration*, the GEM space reserved for a particular task (e.g. buffering of messages) is partitioned so that each CM "owns" one partition. While a CM may read any GEM partition, only the partition owner is permitted to perform write accesses. This has the advantage that write accesses to GEM need not globally be coordinated but can be performed under local control of the partition owner. Furthermore, free space information for a partition can be maintained in main memory data structures of the owning CM. Thus, the administration tasks for communication can be locally performed except for the acknowledgment by the destination CM that it has read the message from GEM. This notification is to be send to the partition owner (sender CM) so that it can release the message buffer in its GEM partition. The acknowledgment can be sent by an additional interrupt or may be piggy-backed with other messages.

The partitioned approach permits a comparatively simple administration, but may require additional communication to coordinate the shared use of GEM. In addition, the partitioning generally results in suboptimal utilization of GEM space. The alternative is to use *central data structures in GEM* for administration. In order to allocate and block GEM space for message buffering, a simple bit list can be used that is stored in a number of GEM entries. To allocate a message buffer in GEM, this bit list is read by the sender CM into main memory and a sufficient number of free GEM entries/pages is reserved by setting the corresponding bits. The modified entries of the bit list are written back to GEM with the Compare&Swap operation. This operation is successful if no other CM has modified the bit list in the meantime (if the operation fails, the bit list must be read again to allocate a message buffer). After the receiver CM has read the message from GEM, it releases the message buffer by resetting the corresponding bits in the bit list. Thus, allocating and releasing a message buffer results in at least four extra accesses to GEM entries. Still, the fast GEM access times should permit communication delays of less than 100 microseconds even for large messages (1 page) as opposed to milliseconds needed with conventional protocols. Since message buffers are held in GEM only for short periods of time, comparatively little GEM space should be required for communication.

4 Use of GEM in Data Sharing Systems

In this section we discuss how GEM can be used in data sharing ("shared disk") systems to improve performance for transaction processing. It turns out that all major functions critical to performance in data sharing systems can benefit from such a shared store. We describe simple methods for concurrency control using a global lock table in GEM and for constructing a global log. In addition, we sketch different forms of GEM usage for coherency control, global buffer management and load control. The GEM usages tailored to data sharing have to be implemented by the DBMS rather than the operating system.

4.1 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.) An overview of concurrency control protocols for data sharing can be found in [Ra88]. A storage-based communication with the GEM as discussed in the previous section could help solve these problems and limit the communication overhead. This would reduce the importance of a low number of inter-system communications and therefore the dependency on the chosen algorithm or workload profile.

Another possibility is to store a global lock table (GLT) in the GEM that can be accessed by all CMs. 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 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. With GEM access times of 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.

The storage of a GLT in GEM is complicated by the fact that GEM is not directly addressable like main memory. Rather we must map the global lock information onto a fixed number of GEM entries of fixed length. Lock information, however, typically is of variable size since first the number of concurrently locked objects is varying, and second the number of granted and waiting lock requests per object is not limited. Since it is not feasible to reserve a GLT control block for every database object, we have to use a hash table of fixed length. To deal with collisions, control blocks of the same hash class must be linked in chains of variable length. Since only one GEM entry can be read at a time, thus multiple GEM accesses may

be needed in order to locate a specific control block. To add a control block to a hash class (for which the first entry is already used), a new entry from a separate pool can be allocated and linked to the hash class. These and other list operations are cumbersome to realize with the simple GEM interface and result in additional GEM accesses. So only in the best case (e.g. for hash classes with no collisions), a GLT control block can be read with a single GEM access.

Similar problems would be introduced by control blocks of variable length. To reduce the complexity of the GLT implementation and to limit the number of GEM accesses per lock request, we therefore demand a fixed and short size of control blocks. This is possible by maintaining a reduced lock information in the GEM by recording lock ownerships/requests on a per-system basis rather than for individual transactions. This approach assumes that transaction-specific lock information is maintained by every CM (DBMS) in a local lock table (LLT) in main memory.
If we distinguish between shared read (R) and exclusive write (X) locks, a GLT control block may hold the following information:

object ID;
GRANTED-MODE: array [1.. NMAX] of [0, R, X];
*#WE: 0.. 2*NMAX; { current number of wait entries }*
GWL: array [1..#WE] of (CM-ID, mode); {global wait list }

NMAX refers to the maximum number of CMs that can be connected to the GEM. The vector GRANTED-MODE indicates for every CM the maximum lock mode granted to a transaction at the respective system (0 stands for no lock granted). The waiting information has also been reduced to a fixed size, although the number of waiting transactions is potentially unlimited. This was possible by storing at most one waiting lock request per CM and lock mode, reducing the maximum number of wait entries to 2*NMAX for two lock modes (R and X). If only the most restrictive lock mode for waiting requests were stored, then concurrency could suffer (e.g. R-locks can now be granted to multiple CMs concurrently even if one of the CMs has requested an X-lock). The request order is also preserved in the global waiting list GWL to support a fair treatment of waiting requests.
The lock information on granted and waiting lock requests is very compact supporting short control blocks. For NMAX=4, we merely need 1 B for GRANTED-MODE, 4 bits for #WE and 3 B for GWL. In addition, 6 B may be required to store the object identification and 2 B for the GEM address of a successor control block of the same hash class. So a total size of 16 B per control block would be sufficient (leaving some space for additional information, e.g. for coherency control).

To process a lock request, the hash class of the respective object is determined and the control block from that GLT position is read from GEM. If that control block is “empty”, the lock can be granted and the newly initialized control block is written back to the GLT with the Compare&Swap operation (if this operation fails,

the lock control block must be read and modified again since another CM may have set an incompatible lock). If there is already a GLT control block for the object, it is checked whether the lock request is compatible with already requested locks. If so, the lock is granted by updating GRANTED-MODE and writing back the updated control block to GEM. If there are already waiting requests or incompatible locks granted, a wait entry is appended to the global wait list provided a local transaction has not yet a request waiting for the same lock mode. A lock release only results in a GLT access if the maximum mode of locally granted locks changes (e.g. after the last read lock is released). In this case, GRANTED-MODE is updated and it is checked whether there are waiting lock requests that can now be granted. If so, a message is sent to the respective CMs to activate the waiting transactions.

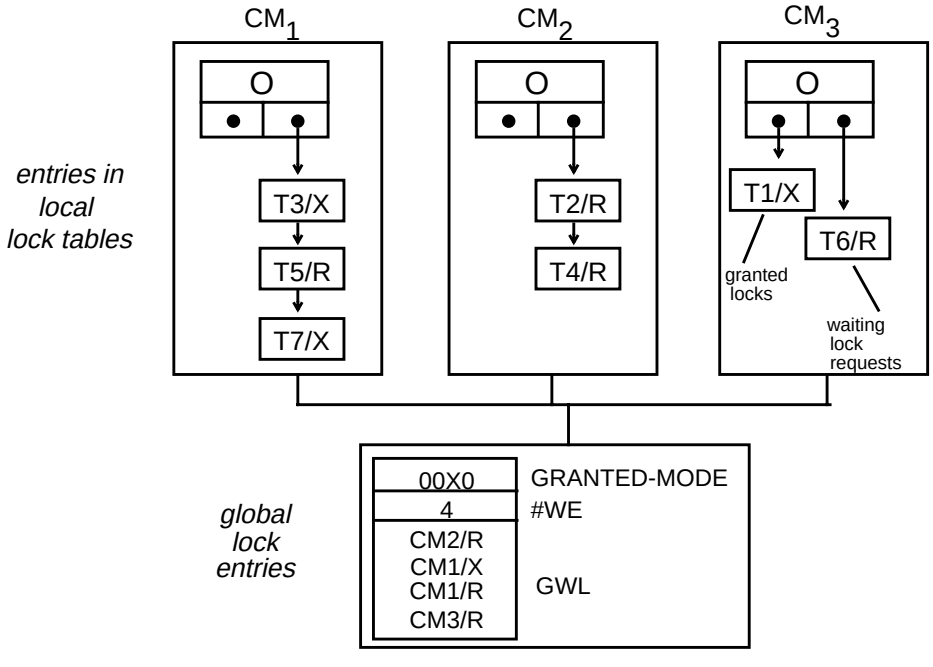


Fig. 3: Lock scenario

Fig. 3 illustrates the use of this lock information. The example refers to three CMs (NMAX=4) and shows lock information in the GLT and the LLTs for object O. In the GLT, GRANTED-MODE indicates that an X-lock is granted to a transaction at CM3; the LLT of CM3 reveals that transaction T1 is holding this lock. While there are six waiting lock requests in the LLTs, only four of them are represented in the GLT. So only one X-request from CM1 and one R-request from CM2 is recorded in the GLT since we can have only one wait entry per CM and lock mode. When T1 releases its X-lock on O, CM3 recognizes from the GWL that there is a waiting R-request from CM2 that can now be granted. After having received the notification by CM3, CM2 grants the R-lock to T2 and T4 and updates the GLT entry (GRANTED-MODE := 0R00; #WE := 3; removal of first GWL element). When the last R-lock is released in CM2, CM1 is informed that its X-request can be granted.

At this point, the second X-request in CM1 (by T7) may be appended to the GWL. This deferred propagation of waiting requests to the GLT guarantees that starvation can be avoided despite the use of reduced waiting information in GEM.

The simple lock protocol requires at least two GEM accesses per lock request and two per lock release resulting in about four to five accesses per lock (assuming hash class collisions are rare). The GLT should be duplexed to deal with GEM and CM failures (a system crash may leave the GLT in an inconsistent state; keeping two copies of the GLT permits detection of corrupted control blocks). Performing write operations twice results in two additional GEM accesses per lock. For a transaction with 10 locks, we would have 60 to 70 accesses to GEM entries resulting in a locking delay in the order of merely 100 microseconds. Lock conflicts result in additional overhead in order to activate waiting transactions. A throughput of 1000 TPS would cause a GEM utilization of about 6% - 7%.

A further reduction of GEM access rates can be achieved by an extended lock protocol where not every lock request is decided with the GLT, but where the CMs are authorized to grant and release locks locally. Such a scheme can be realized according to proposals for reducing the number of global lock requests in protocols for loosely coupled systems, e.g. by using so-called read or write authorizations [Ra88]. A read authorization may be granted to multiple CMs at the same time and authorizes them to grant and release *read* locks locally (i.e. without accessing the GLT). Similarly, a write authorization which can be assigned to only one CM at a time permits a local synchronization of *read and write* lock requests. In the case of many read accesses or strong locality of reference these extensions can significantly reduce the number of GEM accesses, but at the expense of a considerably more complex protocol. Furthermore, additional communication is necessary to revoke the authorizations after another CM has requested an incompatible lock (e.g. an X-request requires to revoke all read authorizations).

4.2 Logging

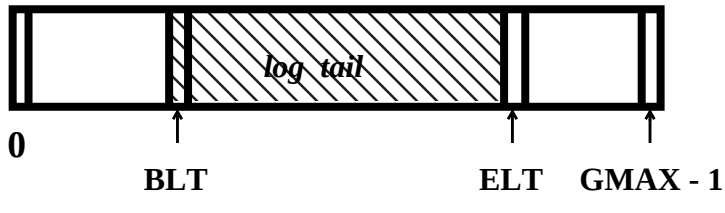
Due to the non-volatility of the GEM and its significantly faster access time compared to disks, logging is a prime candidate for use of a GEM. In disk-based DBMS, the log file is frequently the first bottleneck when higher transaction rates are needed since a single disk is limited to about 60 I/Os per second. This bottleneck is removed with GEM-resident log files supporting thousands of I/Os per second.
The local log files can be held GEM-resident under local control of the respective CMs. However, data sharing also requires the construction of a global log file (e.g. for disk failure recovery) where the database modifications of all CMs are recorded in chronological order [Ra91a]. In existing data sharing systems, the global log is mostly constructed by an offline utility that merges the local log files. This approach results in significant availability problems since a disk failure cannot be re-

covered until the global log has been completed by this utility. Merging the local log files on-the-fly reduces this problem but is very difficult to implement [MNP90]. We propose a simple method that avoids an explicit merging of local log data. Instead we create the global log directly in GEM by always writing the after-images of a transaction at the current end of the global log. While this approach would result in a significant throughput bottleneck if the global log were kept on a single shared disk, the fast GEM access times resolve this problem.

Since the global log is constantly growing, we keep only the current tail of the global log in GEM. A dedicated process running in one of the CMs is assumed to be responsible for writing the global log data from GEM to disk in order to limit the tail size. Thus, the log tail is accessed by this writer process and by committing update transactions. Two special GEM entries are used to mark the begin and end of the log tail:

BLT: 0..GMAX-1; {Begin of log tail}
ELT: 0..GMAX-1; {End of log tail}

The global log tail is organized as a ring buffer of GMAX page frames that are cyclically overwritten by committing transactions. As indicated in Fig. 4, BLT points to the first log page in GEM that has not yet been written out to disk by the writer process. ELT points to the first page after the last page of the tog tail; this entry is only updated by committing transactions. The length of the log tail can easily be computed as shown in Fig. 4. An empty log tail is given if ELT = BLT holds. Similarly, the log tail is full if GMAX-1 pages still have to be written out from GEM (ELT = BLT - 1).
To coordinate read accesses of the writer process with write accesses of committing transactions, we can again use a simple bit list (GMAX bits) stored in GEM entries.



$$\text{\# of pages in log tail} = \begin{cases} \text{ELT - BLT} & \textit{if } ELT >= BLT \\ \text{GMAX + ELT - BLT} & \textit{otherwise} \end{cases}$$

Fig. 4: Organization of a global log tail in GEM

Commit processing of an update transaction consists of local logging followed by global logging and release of the locks. Since a transaction is committed after a successful local logging, only modifications of committed transactions are writ-

ten to the global lock. Holding the locks during global logging guarantees that the redo data in the global log is written in chronological order. Similar to group commit, we can write the global log data (after-images) of multiple transactions together in order to reduce the number of GEM accesses. To write J log pages to the global log tail, the following procedure is followed:

```
read BLT entry from GEM;
read ELT entry from GEM;
IF (number of free pages < J)
THEN wait            {should not occur if GMAX is sufficiently large}
ELSE
      ELT := (ELT + J) MOD  GMAX;
      read bit list from GEM;  {entry containing the J relevant bits}
      set J bits in bit list;
      Compare&Swap (bit list);
      Compare&Swap (ELT);
      write J log pages to GEM;    {from address ELT to (ELT+J-1) MOD GMAX}
      read bit list;
      reset J bits;
      Compare&Swap (bit list);
END IF;
```

Only the update operation to the ELT entry needs to be synchronized between concurrent transactions. After the ELT update, the pages themselves are reserved and can be asynchronously written to GEM. The reserved page frames are blocked against the writer process by setting the corresponding bits in the bit list. This is necessary to avoid that log pages are written out to disk before they are written to GEM. Assuming an access time of 1 microsecond per entry and 20 microseconds per page, we yield (7 + J*20) microseconds for global logging. Duplex logging increases the overhead to (10 + J*40) microseconds, that is about 50 microseconds for one page.

To write the next N pages from the global log tail to disk, the writer process can use the following procedure:

```
read BLT entry from GEM;
read ELT entry from GEM;
LOOP
      IF (log tail empty)
      THEN wait
      ELSE
          read bit list from GEM;      {test if pages can be read}
          read N pages from the log tail;
          write N pages to disk;        {sequential I/O}
          BLT := (BLT + N) MOD  GMAX;
          Compare&Swap (BLT);
      END  IF;
      read ELT;
END LOOP;
```

While the log tail is empty, the writer process periodically reads the ELT entry to check whether new log pages have been written to GEM. For a non-empty log tail, it is first checked whether the next N log pages actually reside already in GEM (indicated by a 0-bit in the bit list). If so, these pages are read into main memory and written to disk. After the disk write, the BLT entry is updated thereby releasing N page frames from the log tail. Note that the writer process does not need to block pages by setting bits since committing transactions do not overwrite GEM pages in the relevant portion of the log tail.

The read accesses by the writer process utilize the GEM by another 20 microseconds per log page. If we collect the global log data for 5 transactions in one page on average, 1000 TPS result in 200 pages per second of global log data. Writing and reading these pages causes a GEM utilization of merely 1.4% (50 + 20 microseconds per page). Sequential I/O also permits writing 200 pages per second to a single disk, although a bottleneck may here be possible for higher transaction rates. This problem can be solved by keeping the global log data for different database partitions in separate (smaller) files. In this case, we would have a separate global log tail in GEM, a separate writer process as well as separate backup disks for every database partition.

4.3 Support for additional functions

Coherency Control

Coherency 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. Coherency 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 for every page modification) [Ra86, Ra88]. If we use a global lock table in the GEM for concurrency control, coherency control can also be accomplished by recording the additional information for modified pages in this table. Furthermore, modified pages can be exchanged between systems very fast across the GEM, irrespective of whether we employ message-based concurrency control or use a global lock table in the GEM.

Caching of database pages in GEM

In addition to a local database buffer (LDB) in each CM, it is desirable to maintain

a global database buffer (GDB) in the GEM. If a page is to be replaced from a LDB, it is written to the GDB for possible rereferences by any system. 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 systems, many modifications can be accumulated per page before it is eventually written to disk.

Unfortunately, the administration of a GDB is much more complicated than for a file cache or extended database buffer (EDB) in the central case. This is because administration tasks like page replacement and update propagation from GEM to disk now have to be coordinated between all systems. In addition, global data structures for free place administration and for locating pages in the GDB are needed. In principle, it is possible to solve these problems with data structures in GEM, but the sole use of GEM entries for this purpose results in a difficult and inflexible realization. A global file cache for data sharing is most easily realized by disk caches being managed by shared disk controllers.
The use of GEM-resident files, however, is easily possible as discussed in section 2. Similarly, every CM can keep a separate EDB in GEM and control it by main memory data structures. Buffer invalidations in the EDB can be detected in the same way than obsolete pages in the LDB.

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 the GEM. Message queues are also candidates for storage in the GEM since an additional message logging could be avoided (due to the non-volatility of the GEM) and because messages could quickly be transferred between different systems.

5 Conclusions

Non-volatile semiconductor memories like disk caches, solid-state disks (SSD) or extended memory can significantly improve I/O performance for transaction processing. They are less expensive than main memory (per MB) and provide better failure isolation than shared main memory in tightly coupled systems. The non-volatility permits us to eliminate synchronous disk writes by writing log data and database pages to the semiconductor store. Furthermore, disk reads can be saved by

keeping entire database files resident in non-volatile semiconductor memory or by employing a second-level database cache.

Our considerations concentrated on the use of shared extended memory for locally distributed transaction systems. Extended memory is directly attached to main memory supporting reduced I/O overhead and significantly faster access times than disks, disk caches or SSD (microseconds instead of milliseconds). While simple machine instructions are provided for data transfer to/from main memory, extended memory is largely controlled by software in the operating system or subsystems like the DBMS. This software control permits a more flexible use than with disk caches and SSD, in particular for distributed systems. Specific usage forms were presented for a special type of extended memory called GEM (Global Extended Memory). GEM is non-volatile and can be connected to multiple systems. Its access interface consists of read and write operations to pages and so-called entries which may be used to implement global data structures. In addition, operations are provided for access synchronization (Compare&Swap) and interrupt propagation.

In locally distributed systems, GEM may be utilized to implement a fast message transfer between systems. Such a storage-based communication not only permits short communication delays, but may also reduce communication overhead to a large extent. A smaller communication overhead facilitates horizontal growth since a reduced overhead permits a higher effective CPU utilization (higher transaction rates). Furthermore, if remote requests incur little overhead and delay, their minimization is no longer the overriding design goal. This permits simpler algorithms for global coordination and facilitates load balancing.
Apart from inter-system communication, data sharing systems can utilize GEM in a number of more specific areas. As outlined in section 4, it is comparatively easy to use a global lock table in GEM for system-wide concurrency (and coherency) control and to construct a global log. Compared to loosely coupled systems, this cannot only reduce the number of messages to a large extent, but the implementation of these important functions is also simplified. We feel that the availability of shared extended memory like GEM will make the data sharing approach for distributed transaction processing very attractive for high-volume applications.

In [BHR90], we have presented a preliminary performance evaluation of GEM usage for data sharing systems. The reduced overhead and delay for I/O and communication was found to significantly improve throughput and response times compared to loosely-coupled, disk-based architectures. This was particularly the case for real-life workloads (represented by database traces) for which lock contention and unbalanced system utilization often were a performance bottleneck without GEM.

References

[BHR90] Bohn, V.; Härder, T.; Rahm, E.: Extended Memory Support for High Performance Transaction Processing. Techn. Report 5/90, Dept. of Comp. Science, Univ. Kaiserslautern, to appear in: Proc. 6th (German) Conf. on Measurement, Modelling and Evaluation of Computer Systems, Springer-Verlag, Munich, Sep. 1991.

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

[CKKS89] Copeland, G.; Keller, T.; Krishnamurthy, R.; Smith, M.: The Case for Safe RAM. Proc. 15th VLDB, 1989.

[DDY91] Dan, A.; Dias, D.M.; Yu, P.S.: Modelling a Hierarchical Buffer for the Data Sharing Environment. IBM Research Report RC 15707, Proc. ACM SIGMETRICS, 1991

[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.

[Gro85] Grossman, C.P.: Cache-DASD Storage Design for Improving System Performance. IBM Systems Journal 24 (3/4), 316-334, 1985.

[Gro89] Grossman, C.P.: Evolution of the DASD Storage Control. IBM Systems Journal 28 (2), 196-226, 1989.

[KGP89] Katz, R.H.; Gibson, G.A.; Patterson, D.A.: Disk System Architectures for High Performance Computing. Proc. of the IEEE 77 (12), 1842-1858, 1989.

[Ku87] Kull, D.: Busting the I/O Bottleneck. Computer & Communications Decisions, 101-109, May 1987.

[LR88] Leland, M.D.P.; Roome, W.D.: The Silicon Database Machine: Rationale, Design, and Results. in: Database Machines and Knowledge Base Machines, Kitsuregawa, M.; Tanaka, H. (eds.), 311-324, 1988 (Proc. 5th Int. Workshop on Database Machines, 1987).

[MNP90] Mohan, C.; Narang, I.; Palmer, J.: A Case Study of Problems in Migrating to Distributed Computing: Data Base Recovery Using Multiple Logs in the Shared Disks Environment. IBM Research Report RJ 7343, San Jose, 1990.

[Ra86] Rahm, E.: Primary Copy Synchronization for DB-Sharing. Information Systems 11 (4), 275-286, 1986.

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

[Ra91a] Rahm, E.: Recovery Concepts for Data Sharing Systems. Proc. 21st Int. Symp. on Fault-Tolerant Computing, Montreal, IEEE Computing Press, 368-375, 1991.

[Ra91b] Rahm, E.: Performance Evaluation of Extended Storage Architectures for Transaction Processing. Techn. Report, Computer Science Dept., Univ. Kaiserslautern, 1991.

[Ru89] Rubsam, K.G.: MVS Data Services. IBM Systems Journal 28 (1), 151-164, 1989.

[Se90] Selinger, P. G.: The Impact of Hardware on Database Systems.. Proc. Int. IBM Symp. "Database Systems of the 90s", Lecture Note in Computer Science 466, 316-334, 1990.

[Sm85] Smith, A.J.: Disk Cache - Miss Ratio Analysis and Design Considerations. ACM Trans. on Computer Systems 3 (3), 161-203, 1985.