Efficient Locking for Shared Memory Database Systems [1]

Lory D. Molesky and *Krithi Ramamritham*
Computer Science Technical Report 94-10
Department of Computer Science
Lederle Graduate Research Center
University of Massachusetts
Amherst Ma. 01003-4610
March 1994

Keywords: *shared memory database system, autonomous locking, hot locks, database recovery, transaction processing.*

Abstract

Significant performance advantages can be realized by implementing a database system on a shared memory multiprocessor. An efficient implementation of a lock manager is a prerequisite for efficient transaction processing in multiprocessor database systems. To this end, we advocate adopting an *autonomous locking* strategy. In autonomous locking, transactions acquire and release locks via operations on shared memory, in contrast to conventional locking where communication with a lock manager is involved. We demonstrate the superior performance and scalability of autonomous locking through benchmarks of a prototype lock manager implemented on a large scale shared memory multiprocessor. For instance, when contention is high (i.e., for *hot lock* operations), the performance of autonomous locking is nearly an order of magnitude better than conventional locking. The paper also addresses issues related to recovery (assuming independent node failures).

[1]Partially supported by the National Science Foundation under grant IRI-9109210

Contents

1 Introduction

Although there has been a significant amount of work done on *shared disk* (SD) and *shared nothing* (SN) database systems, very little work has been done on *shared memory* (SM) database systems. In an SD system, each node is connected to all disks in the system, while in an SN database system, the database is partitioned among multiple systems. In an SM database system, in addition to having shared disks, each node has access to shared memory. The recent advances in multiprocessor technology have made *large-scale* cache coherent multiprocessors possible, enabling the construction of high performance SM database systems.

The lock manager is a critical component of any database system which uses locking to ensure the correctness of concurrently executing transactions. In a database system containing multiple nodes, many lock requests may be issued concurrently by transactions executing on separate nodes. By adopting a cache coherent shared memory system model, we demonstrate how database locking on multiple nodes can be performed very efficiently. In this strategy, called *Autonomous Locking* (AL), transactions acquire and release locks directly via operations on shared memory. The AL strategy is particularly effective in the presence of *hot spots.* Hot spots occur when a small set of data elements are requested concurrently by many transactions [20, 16, 12]. When multiple transactions attempt to lock the same data object in a compatible mode (creating a *hot lock*), the lock manager becomes a performance bottleneck. Our performance studies indicate that the performance of AL is significantly better (by almost an order of magnitude) than that of conventional locking techniques, especially for hot locks.

Given that a large volume of work has been done on locking techniques for SD database system, (which we call *conventional locking*, or CL), it is instructive to compare AL to CL. Typically, an SD database system employs one or several lock manager processes to control the acquisition and release of locks on behalf of transactions. Thus, when a transaction running on some node needs a lock on a database object, it must contact the appropriate global lock manager with the lock acquisition request. Similar inter-node communication occurs in order to release an acquired lock. However, performing inter-process communication without shared memory can be expensive. A centralized CL architecture imposes an inherent sequential bottleneck on the lock acquisition/release process. This has been recognized, and to mitigate this problem, multiple lock managers can be used, where each manages a distinct partition of the lock space [17]. Furthermore, to increase availability, each lock manager normally runs on a dedicated node. Although partitioning the lock space in this manner can be effective for CL, by adopting AL, we can further substantially decrease the overheads for database locking operations. Adopting AL over CL has a number of advantages:

- AL eliminates inter-process communication overheads.
- AL eliminates a sequential bottleneck, thus increasing potential concurrency.
- AL eliminates the need for dedicated lock manager processes.
- AL achieves high lock space availability without explicit partitioning of the lock space.

By eliminating the lock manager processes, AL eliminates a number of inherent inefficiencies associated with it. Since all lock acquisitions and requests operate directly on shared memory, no inter-process communication is required in AL. In CL, when lock space partitioning is done and many lock manager processes are employed, a given series of concurrent lock requests for different locks may still contact the same lock manager. In contrast, with our implementation of AL, concurrent lock requests for different locks are not subject to any sequential bottleneck. Furthermore, regardless of the CL partitioning scheme, AL is superior for access to *hot* locks.

On the surface, the autonomous locking strategy is no different than the technique that a multiprocessor operating system developer would use to implement a semaphore. However, there are two significant differences between semaphores and database locks. First, unlike semaphores, database locks must be managed so to ensure *failure atomicity* [2] of transactions, and second, in addition to shared and exclusive modes, database locks normally support other lock modes. Ensuring the the lock space reflects failure atomicity of transactions is especially difficult when nodes can fail independently. Consider a node crash which occurs while some active transaction has acquired shared locks. If some of the lock control blocks (the data structures indicating, among other things, which transactions are currently sharing the lock) had migrated to another node (due to shared lock requests at other nodes), a crash may lose some but not all of a transaction's lock control blocks. In this case, some strategy must be devised in order to ensure that a consistent state of the lock space can be reached during recovery. Our implementation of AL meets the requirement of failure atomicity.

We have conducted an evaluation of AL and CL on a large scale shared memory multiprocessor. Our experiments demonstrate that under common locking scenarios, the locking cost

[2]Failure atomicity [15] of a transaction, also called the *all-or-nothing* property, means that either all or none of the transaction's operations are performed.

for CL increases linearly with contention while the locking costs for AL remain fairly constant. In fact, for hot locks, AL's performance is better than CL's by almost an order of magnitude.

Section 2 reviews some background material, including our assumptions about the system model, transaction model, a general discussion of database locking, and reviews conventional locking. Section 3 discusses autonomous locking. Section 4 discusses the performance of autonomous and conventional locking. Our conclusions are presented in section 5.

2 Background

This section briefly reviews background material related to database locking, SD and SM database systems, and conventional techniques for performing locking in SD database systems. Suitable platforms for SM database systems, namely, a cache coherent shared memory multiprocessors and cache coherent distributed shared memory machines, are discussed in section 2.1. Locking is the preferred method for enforcing serializability in commercial relational and object-oriented database systems, and is discussed in section 2.2. Section 2.3 discusses hot spots and hot locks. Finally, conventional locking (CL) in an SD database system is discussed in section 2.4.

2.1 System and Transaction Model

Our implementation platform for an SM database system is a a cache coherent shared memory multiprocessor. A coherency protocol, implemented in hardware, ensures that any read operation sees the most recently written value for any data item. Each node has its own cache, and before an operation is performed on a data item, the data item must first be brought into the cache. In general, a data item will be either in the local node's cache, another node's cache, or on disk.

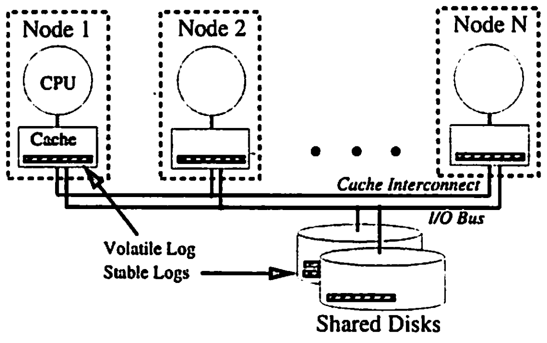

Figure 1: SM Database Architecture

Typically, the operations execution time is minimal if the data item is already in the cache, more expensive if the data item is in another node's cache, and the most expensive if the data item must be fetched from disk. We assume a page based write-invalidate snoopy cache model [1, 19], where, before a page write by one node occurs, all other cache copies the page are first invalidated. Typically, the hardware elements implementing the cache coherency scheme include a cache controller, a cache directory, and the cache itself. The cache contains the cached data, while the cache directory contains the addresses of all cached data. Figure 1 illustrates the basic SM database architecture. We assume each node maintains a non-volatile log, and that in-place updating is used in conjunction with the write-ahead log protocol [2]. All operations on this log take place in the node's cache. This in-cache log is volatile, but can be made stable by writing it to one of the shared disks.

We focus on transaction workloads where each transaction executes entirely on a single node. In contrast, a query processing approach may entail executing a single transaction on multiple nodes. Although an SM database system is well suited for query processing, the presentation of the recovery strategies is simplified when each transactions executes on a single node. However, as shown in section 3.2.2, our results generalize to include transactions which execute on multiple nodes.

If the failure of a single node implies the failure of the entire multiprocessor, then the database recovery issues are essentially the same as for a uniprocessor. In this paper, we address the more complex recovery issues which surface when node failures are independent. Next generation cache coherent commercial multiprocessors are being designed to support independent node failures [6]. Furthermore, cache coherent shared memory can also be efficiently supported on a *distributed shared memory* (DSM) machine. Stanford's FLASH [10] research architecture is one example of a DSM machine. In FLASH, hardware support for cache coherency is implemented by connecting several off-the-shelf microprocessors, caches, and custom interconnect controller chips via a high speed network. This looser coupling of system nodes facilitates the identification and isolation of single node hardware failures. To take advantage of these future enhancements, in our model, we assume that nodes in the shared memory multiprocessor fail independently. A node failure implies all volatile memory (the node's cache, and possibly the cache directory) is lost. When node a, which holds the only copy of a page, crashes, we assume that the page is unavailable for use by other nodes until a recovery procedure is executed, which reconstructs some or all of the dirty data of cache a. The volatile and stable logs facilitate the reconstruction of the database data and database lockspace located on a node after a crash.

	NL	*IS*	*IX*	*S*	*SIX*	*X*
NL	√	√	√	√	√	√
IS	√	√	√	√	√	-
IX	√	√	√	-	-	-
S	√	√	-	√	-	-
SIX	√	√	-	-	-	-
X	√	-	-	-	-	-

Figure 2: Lock Mode Compatibility Table

Although node failures in an SM database system should be infrequent, some additional measures can be taken to increase the availability of the system. Although an expensive proposition, all volatile memory could be mirrored with redundant hardware. This would increase availability, but the difficulty of recovery issues would remain if the backup memory also failed.

2.2 Locking in Database Systems

To ensure serializable executions, virtually all commercial database systems use locking. The basic lock modes are shared (S) and exclusive (X). An X lock on record (or database object) r guarantees that no other transaction will read or modify r, while an S lock on r ensures that no other transaction will modify r. Note that several S requests on r can be granted concurrently. In addition to the basic lock modes, intention locks are also included to facilitate multigranularity locking [9]. In a multigranularity locking protocol using intention locks, the database is characterized by a hierarchy (on a directed acyclic graph), with locks are requested in root to leaf order, and released leaf to root. Before requesting an S or IS lock on a node, all ancestor nodes of the requested node must be held in IX or IS mode by the requestor. Also, before requesting an X, SIX, or IX lock on a node, all ancestor nodes (if any) of the requested node must be held in SIX or IX mode by the requestor. The lock mode compatibility table which enables multigranularity locking is illustrated in figure 2. Object oriented systems use similar compatibility tables to ensure serializability [7].

2.3 Hot Locks

The discussion in the previous subsection indicates that there are many cases where locks may be concurrently held by many transactions. While *hot spots* [13, 16] refer to data which is frequently accessed, we use the term *hot locks* to refer to locks on database objects which are frequently accessed in a non-conflicting mode.

> A _Hot Lock_ on r is a lock where a non-conflicting lock operation (acquire or release) is performed by many concurrent transactions.

Hot locks may arise under many situations in multi-node database systems. For example, suppose many query transactions simultaneously attempt to lock r in S mode. In this case, since there are many concurrent compatible requests, the lock on r is hot. In a multigranularity locking scheme, it is very likely that intention locks will be concurrently requested at the nodes close to the root. For instance, multiple IS and IX locks may be concurrently held by many transactions on the root node of the lock hierarchy; lock requests accessing this root node may easily form a hot lock. These two factors, the likelihood of hot locks in a multi-node database system, and the potential for concurrent transaction execution in these situations, motivate our lock manager design. Our performance studies indicate that autonomous locking performs extremely well under hot lock traffic.

2.4 Locking and Recovery in Shared Disk Database Systems

In the conventional approach to locking for SD database systems, global concurrency control is enforced by having one lock manager process per database object. To acquire or release a lock, messages are sent to the appropriate lock manager process. In a survey of concurrency and coherency control for SD database systems [17], Rahm discusses three locking schemes for SD systems: central locking, distributed locking with fixed authorities, and distributed locking with dynamic lock authorities. In these protocols, for each database object, a single lock manager enforces global concurrency control. In central locking, there is only one lock manager, while in dynamic locking, there are multiple lock managers, each responsible for a separate partition of the database object space. Distributed locking is further distinguished based on whether the partitions are predetermined (fixed authorities) or change dynamically (dynamic lock authorities).

The disadvantage of the central locking approach is that the lock manager process can become a performance bottleneck. In order to mitigate this performance bottleneck, normally, an entire node is reserved for the central lock manager. Further performance gains can be achieved by partitioning the lock space with distributed locking. By partitioning the lock space, one hopes to decrease the overheads of lock acquisition and release by enabling multiple lock managers to process requests concurrently.

Although very little work has been published on recovery in SD systems, recovery issues are addressed (in the context of fine-granularity locking in an SD transaction environment) in ARIES [12]. Under Rahm's taxonomy, this system falls under the category of central locking. In ARIES, locking is managed by a single GLM (Global Lock Manager) process and a per node LLM process (local lock manager). The transaction system first sends lock requests to the LLM. If the lock is currently held by by the LLM, the LLM itself processes the request. Otherwise, the lock request is forwarded to the GLM. On a GLM failure, the global lock table is reconstructed from the information contained in the LLMs. When the GLM notices that a LLM l has failed, all of l's locks held at the GLM are released except those which were asked to be *retained* by the local system (normally update locks are retained). For increased availability, one node in the system is designated as the backup GLM. The backup GLM will replace the GLM in the case where the GLM and at least one LLM have failed. During recovery, no surviving node will be granted any of l's locks by the backup GLM until all nodes recover completely.

To the best of our knowledge, only one paper has addressed issues in implementing a lock manager on an SM database system. In [18], instead of using a GLM per object, it is suggested that a *global lock table* be used in conjunction with a non-volatile *global extended memory.* Database locks are acquired directly from the global lock table stored in non-volatile shared memory. However, this assumption of *non-volatile* shared memory is a significant departure from our implementation of AL on an off-the-shelf shared memory multiprocessor wherein the cache – where (parts of) the lock space may reside – is volatile. Although assuming non-volatile memory basically ensures failure atomicity without additional logging mechanisms, non-volatile memory is much more expensive than volatile memory. Furthermore, in [18], no quantitative performance results are given, only conjectures are provided. In contrast, we provide quantitative performance measurements for AL, and address recovery from failure in the context of volatile memory.

3 Autonomous Locking

In this section, we discuss the basic operation (section 3.1) of AL and the related recovery issues in section 3.2. These recovery issues include ensuring a failure atomic lock space[3], and

[3]We use the term *failure atomic lock space* to mean that the lock space reflects failure atomicity of transactions.

integrating AL into a DB system which supports fine granularity locking.

3.1 Basic Operation

Consider acquiring an object lock under AL. A lock request consists of a lock *name* and a lock *mode*. Using a hash function, the name is translated to a *lock control block* (LCB) address specific to one lock. An LCB stores the current mode of the lock, plus two transaction lists, one containing the current holders of the lock, the other containing any transactions waiting for the lock. All updates to the LCB are performed inside a critical section. If the requested mode is compatible with the mode stored in the LCB, and there are no conflicting waiters, a tuple containing the requesting transaction and requested mode is added to the holder list, and the lock is granted. Otherwise, the transaction/mode tuple is added to the wait list, and a not-granted flag is returned to the requestor. The strategy for releasing a lock is similar. After finding the appropriate LCB, the tuple identified by the transaction is deleted from the holder list, and any lock requests in the wait list which become compatible due to the release are granted. A similar strategy for acquiring and releasing locks is employed in CL. However, in CL, since each lock has exactly one process (node) responsible for all lock operations, no explicit critical section is required.

By exploiting the shared memory available in an SM database system, AL eliminates the need for a lock manager process by using *data* synchronization. By replacing *process* synchronization with *data* synchronization, significant performance gains result. In CL, the lock operation request must be channeled through an intermediate process, the global lock manager. By eliminating the global lock manager, AL eliminates one source of overhead involved in the execution of lock operations. The global lock manager can become a sequential bottleneck when many lock operations are sent concurrently to the same global lock manager. This traffic pattern can occur under the following situations:

- Under a poor partitioning scheme.
- Under hot lock operations.

Even when multiple global lock managers are used, for a given partitioning scheme, it may be the case that multiple requests for lock operations on *different* database objects are channeled to the same global lock manager. An AL scheme will not encounter this problem, since, on each node, the lock operation code can be executed independently. AL also has performance advantages especially for hot lock operations. Under CL, all lock operations on a hot lock will necessarily go to a single global lock manager (recall in CL all lock operations on a particular database object go to a single global lock manager). Although under AL, consistency of access to a particular LCB is ensured by requiring all accesses to the LCB to occur within a critical section, this mechanism is generally much more efficient than ensuring consistency via inter-process communication.

3.2 Recovery Issues

In this section, we address recovery issues for SM database systems in the context of independent node failures. The difficulty of recovery in multiple node database systems stems from the fact that uncommitted data of a single transaction can become partitioned on multiple nodes (due to page migration). Furthermore, for AL, the lock space of a transaction can both (a) coexist in the volatile memory of nodes executing transactions, and (b) become partitioned on multiple nodes. We discuss two main issues in the recovery of an SM database system: (1) the support for fine granularity locking, and (2) ensuring a failure atomic lock space.

3.2.1 Supporting Fine Granularity Locking

When multiple records are stored in a single page, using page locks to ensure serializability would unnecessarily restrict concurrency. To mitigate this problem, contemporary database systems use fine granularity locking [12], i.e., locking is performed on individual database objects. In an SD database system, pages located in the database buffer are transferred between lock manager processes, while in an SM database system, the cache coherency protocol performs inter-node page transfers transparently. When supporting fine granularity locking, it is possible for uncommitted data to migrate between nodes in either of the SD or shared memory architectures (due to the disparity between lock granularity and data transfer granularity).

When uncommitted data migrates, additional steps must be taken to avoid inter-node cascading aborts. For example, suppose records $r1$ and $r2$ are stored in page p, and record-level locking is performed using 2PL (two-phase locking). Transaction t^a on node a locks, then updates $r1$. Then, transaction t^b on node b locks, then updates $r2$. Consider the effects on transaction a if either node a or b crash. If node b crashes, aborting t^a can be avoided if sufficient redo information is maintained to redo t^a's update to $r1$. Similarly, if node a crashes, aborting t^a can be avoided if sufficient undo information is maintained to undo t^a's update to $r1$.

Maintaining the redo information can be efficiently supported by (volatile) logging updates, and maintaining the undo information is typically supported by stable logging the undo of the update. In the example, prior to allowing p to migrate, a redo record for t^a's update to $r1$ is (volatile) logged, and an undo record for t^a's update to $r1$ is stable logged at node a. These methods are employed by the ARIES SD transaction environment [12]. In ARIES, the GLM is used to enforce these policies. Methods for ensuring a cascadeless system in the context of a multiple node implementation of multi-level transactions are also discussed in [14].

In AL, avoiding inter-node cascading aborts must be achieved without a global lock manager. In AL, ensuring that the redo information exists before a page migrates can be performed by holding a short-term page lock until the record update is logged. For ensuring that undo's can be performed in the case where uncommitted data had migrated from node a prior to the crash of node a, two alternatives are the exist:

1. Extend the cache coherency protocol to force the log to disk prior to the migration of a "dirty" page.
2. Add an additional field to each database object, enabling the storage of the before image.

Solution 1 requires an extension of the cache coherency protocol. We envision that only those pages updated by transactions need this extension, so a per page flag will indicate which dirty pages require the invocation of a "migration handler". For these pages, the migration handler is invoked prior to page transfer, which is responsible for forcing the undo log to disk. Solution 2 avoids the delays associated with disk I/O at the cost of an extra field per updated database object. This extra field stores the before image of the database object on the same page as its associated uncommitted image. To effect an undo, the uncommitted data is simply replaced by the before image.

3.2.2 Ensuring a Failure Atomic Lock Space

Ensuring a failure atomic lock space is especially difficult when nodes can fail independently. Compatible locks, i.e., those which may be concurrently held by transactions on different nodes, pose a special problem for recovery, since, as a side effect of the coherency protocol, the LCB will be valid at only one of the nodes. For example, after two transactions running on different nodes have acquired a compatible lock, the LCB will be valid at the node which acquired the lock last. In this case, a crash may lose some but not all of a transaction's LCBs. For instance, note that this scenario is only applicable to uncommitted transactions, since committed transactions have no effect on the lock space (once a transaction has committed, all its locks are released). In contrast, each lock acquired by an uncommitted transaction will have a corresponding entry in the lock space.

AL ensures failure atomicity of transactions in the presence of independent node failures as follows. Before a transaction acquires any locks, its transaction identifier (TID) is stable logged. In addition, prior to any lock acquisition request, a log record, containing the lock name and TID of the lock request, is volatile logged. These log records are used to ensure consistency of the lock space after a node failure occurs. Let us examine what steps should be taken if an LCB (*lcb*), stored on node *l*, is lost due to the crash of node *l*. Active transactions which have entries in *lcb* fall into two categories with respect to recovery of the lock space:

1. An active transaction *l*, which executes on the same (<u>l</u>ocal) node where *lcb* is stored, or
2. An active transaction *r*, which executes on a (<u>r</u>emote) node which is different from where the *lcb* stored.

In case 1, the transaction *l* should be aborted, since the node it was running on crashed. In this case, losing the LCB entry of transaction *l* poses no problems for recovery, since the effects of aborted transactions should be erased. (To complete the abort of transaction *l*, additional measures must be taken, as discussed in the previous section. Recall that objects updated by transaction *l* may have migrated to another node. In this case, to undo the update, locks need to be acquired prior to undoing the update.)

In case 2, it is not necessary to abort the transaction *r*, since the node it was running on is still active. In this case, all of transaction *r*'s LCB entries should be restored. Since they were logged at node *r* prior to allowing the LCB to migrate, this information can be used to reconstruct the lost LCB entries.

This discussion has considered only a single node failure. However, to recover in the case where multiple nodes have failed, the same cases, and thus the same techniques, apply. In this context, all transactions running on nodes which have failed are aborted, and any locks held by these transactions must be removed from the lock space (case 1). Any locks of non-aborted transaction's which were lost by node crashes must be restored (case 2).

Although we have presented this material under the assumption of independent transactions *executing* on a single node, these recovery techniques apply to transactions which execute on multiple nodes. Consider a transaction running on multiple nodes. If one node crashes, generally, it will be necessary to abort the entire transaction. Once this is done, the same mechanisms are used to ensure a failure atomic lock space.

3.2.3 Failures on a Cache Coherent Shared Memory Multiprocessor

In this section we address the possible types of single node failure on a cache coherent shared memory multiprocessor, and suggest solutions to recovering from these failures. A system failure could be caused by any of the hardware elements implementing the cache coherency scheme, including the cache controller, the cache directory, or the cache itself. If any of these components fails, the entire node could be isolated and taken off-line. However, we must address the issues of (1) knowing when a node failure has occurred, and (2) determining the dirty pages which the failed node had cached. Node failure must be identified by the hardware. Once this is done, if the failed node's cache directory remains intact, the cache directory of the failed node may be probed in order to determine which cached pages are dirty. However, if the contents of failed node's cache directory are lost, the dirty pages can be recovered from the logs of the other systems, as is done in [12]. In [12], every page in the database has a `Page_LSN` which contains the LSN (log sequence number) of the log record that describes the latest update to that page. During recovery, this information is used to reconstruct any dirty page which had been lost due to a system failure.

3.2.4 Summary

For an SD database system, coherency for database objects in implemented in software, while in an SM database system, database buffer coherency is performed as a side effect of the hardware cache coherency protocol. AL exploits the more efficient SM coherency protocol by acquiring locks directly from shared memory. For AL, we discussed appropriate mechanisms for ensuring the failure atomicity of transactions, and how to support fine granularity locking in the context of a hardware cache coherency protocol. The mechanisms for supporting fine granularity locking in SD and SM are similar. The most costly mechanism used to ensure a cascadeless implementation is stable logging, and is effective for both SD and SM systems. However, in both systems, ensuring a cascadeless implementation *without* stable logging is possible by storing the before image of a record in the same page as the uncommitted update.

The major differences in recovery for SD and SM involves ensuring that the lock space can be returned to a consistent state after a node crash (in order to ensure the failure atomicity of transactions). The recovery issues for a *centralized* implementation of CL have been documented in [12]. If either a GLM or LLM fails, the surviving node can be used as a source of lock reconstruction information. In AL, a node failure could (1) cause a transaction abort but not lose all the transaction's locks, or (2) lose locks of a non-aborted transaction. To cope with these cases, recovery mechanisms are necessary to ensure that the lock space can be returned to a consistent state. In order to facilitate undo and redo on the lock space, the transaction identifier is logged to stable storage prior to acquiring any locks, and, as is normally done in any lock manager implementation, when a transaction acquires a lock, the transaction identifier and the granted lock mode are stored in the lock control block. These mechanisms would also be useful for a CL implementation to handle the case when the GLM and at least one LLM fail.

Since the mechanisms to support recovery and cascadeless implementations are similar for AL and CL, thus implying similar overheads, we did not implement these support mechanisms, nor did we measure the impact of their performance of the database system. Instead, our performance studies presented in section 4 focuses on assessing the overheads involved in performing lock operations. Our performance studies show that at a concurrency level of 24, to acquire one hot lock under CL costs 1.09 ms., while it costs 0.17 ms. under AL. It would be worthwhile to assess the system's overall transaction throughput rate under AL and CL, but to do so, one must also consider the cost of performing disk I/O. This is part of our future work.

4 Performance Studies

In this section, we present performance benchmarks of prototypes of the CL and AL lock manager designs on the KSR1 multiprocessor. Both AL and CL implementations were performed for the following two reasons. First, we wanted to compare (an SM implementation of) AL with an SD implementation of CL. However, if we were to use existing communication mechanisms (such as TCP or UDP) for inter-node communication, performance of SD CL would be very poor. For example, on the KSR1, the round trip time to write one byte of data was measured at 13 ms. for TCP, and 10 ms. for UDP [4]. Second, assuming a cache coherent shared memory multiprocessor was adopted as the platform for the database, we wished to quantify the differences between AL

[4]These inter-node communication costs can be improved by using high speed networks and bypassing the TCP or UDP protocol stack. Using these methods, [11] reports a best case roundtrip inter-node message time of 500 μsecs.

and a *shared memory* implementation of CL.

The purpose of our experiments was to compare the performance of AL vs. CL under cases where control returns to the transaction without waiting. Since this is a very common lock manager data path, it is an important path to optimize [8]. For lock acquisitions, control returns without waiting when the requested lock is compatible with the current lock mode. When the lock request is compatible, the transaction identifier and requested mode are appended to the LCB and the lock is granted. Otherwise, when the lock request is not compatible, the transaction identifier is enqueued in (the wait list of) the LCB and the lock is not granted. We focused on the first case, since it covers all requests in which the requesting transaction can proceed immediately. Furthermore, since the code for lock release is similar to the lock request code, our comparisons focused on timing the code for lock requests.

Our performance studies also focused on comparing AL and CL under contention for hot locks. For hot locks, regardless of the CL partitioning scheme, all lock requests will be sent to the same global lock manager. For hot locks under AL, accesses to the LCB storing the hot lock are serialized via a critical section. For non-hot locks (concurrent requests for *different* locks), under most request patterns, the performance results for both AL and CL would improve. For non-hot locks under CL with multiple GLMs, a good partitioning scheme would reduce the ill-effects of contention. For non-hot locks under AL, the ill-effects of contention should be negligible, since contention for page locks would be reduced, leaving only the overheads of bus contention.

Section 4.1 discusses the KSR1 implementation platform. Section 4.2 discusses details of the AL and CL implementation on the KSR1. Section 4.3 discusses our experimental methodology, and section 4.4 discusses the results of our experiments.

4.1 KSR1 Multiprocessor Overview

The KSR1 architecture can support up to 1088 nodes. All memory in the system is cache memory, thus the KSR architecture is called *AllCache*. The node interconnect is a 2-level hierarchy where the first level, called a *ring*, implements coherency among groups of 32 nodes. Up to 34 rings can be connected together, yielding a 1,088 node configuration. Each node has a *local cache* which is 32 MB, plus a smaller .5 MB *subcache*. Half of the subcache is an instruction subcache, the other half is a data subcache. Coherency is performed between local caches, and all data and instruction references are made through the subcache, using the local cache as an intermediary (if necessary). Pages in the local cache are 16K bytes, while pages in the subcache,

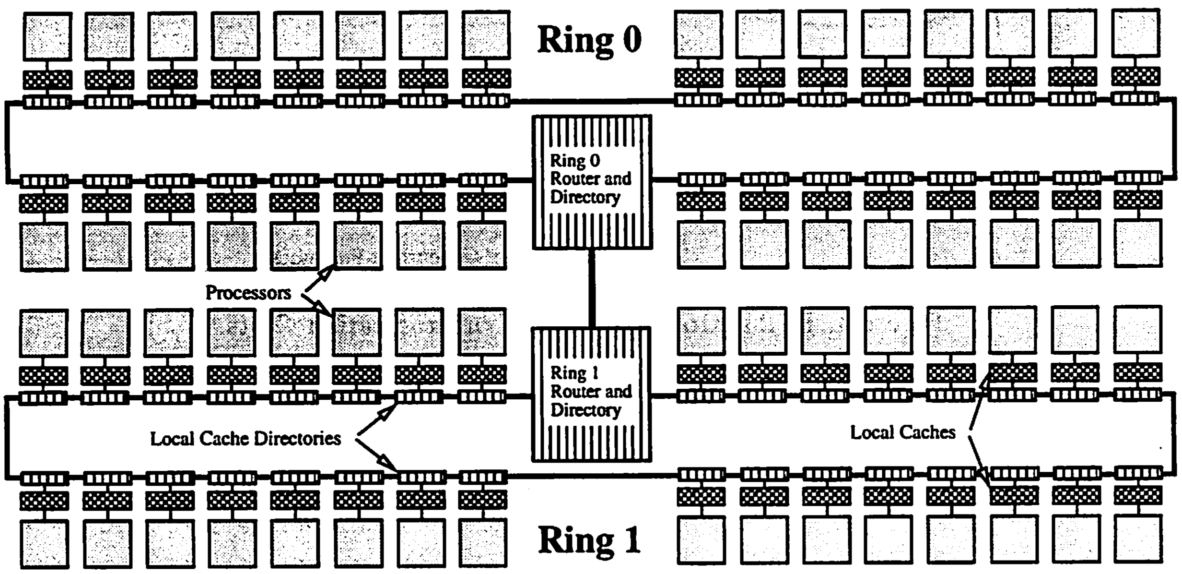

Figure 3: A 64-node KSR1

called *subpages*, are 128 bytes. Our experiments were run on a 64-node, 2-ring KSR1, illustrated in figure 3.

This 2-level cache interconnect implements a 5-level memory hierarchy, consisting of the subcache, local cache, local ring's cache access, remote ring cache access, and disk. If a memory request is not satisfied by the node's subcache or local cache, the local ring is first searched. If the requested memory address is not found on the local ring, remote rings are then searched. If the address is not located in any of the caches, the disk will finally be accessed. A subcache access requires 2 cycles [5], local cache access 20 - 24 cycles, local ring access 175 cycles, remote ring access 600 cycles [19].

The node datapaths are 64 bits wide, it has 64 floating point registers, 32 integer registers, and 32 address registers. On each clock cycle, an instruction pair, consisting of one address calculation, branch, or memory instruction, and one IPU/FPU operation can be initiated.

4.1.1 Synchronization Primitives

The KSR1 provides simple yet very effective primitives for ensuring that sections of code can be executed indivisibly and in isolation. The *get subpage* (gsp) and *release subpage* (rsp) can be used to implement critical sections. The gsp(p) instruction requests that the invoker obtains a

[5]A clock cycle on the KSR1 is 50 ns.

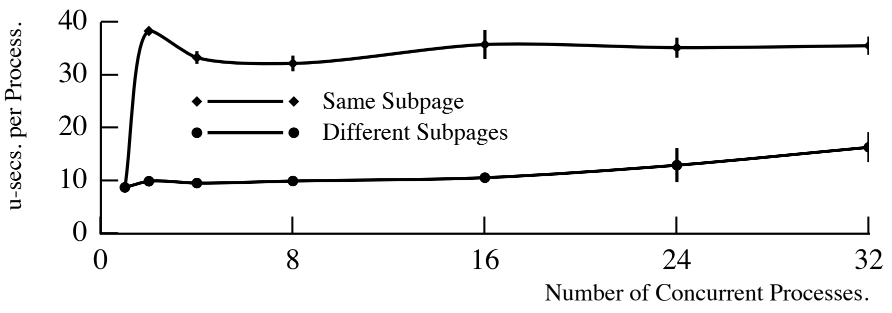

Figure 4: KSR1 Atomic Subpage Synchronization Costs.

subpage p in a mutually exclusive (ME) state, while rsp(p) removes a subpage from ME state. The semantics of the gsp primitive are such that, if the subpage is not already in ME state in any cache, the local cache acquires it in ME state [6]. Thus, once a subpage p is locked by gsp(p), no other process, whether it is on the same or a different node, can acquire p until it is released.

When the data that a critical section accesses can fit on a single subpage, the implementation is extremely efficient. One reason for this is that, as a result of synchronization, the critical section data is also brought into the requesting node's cache. Furthermore, the overheads for page synchronization on the KSR1 are small, even under high contention. This is illustrated in figure 4, which plots the overheads involved in acquiring a subpage in ME state. The bottom curve plots the cost of N nodes simultaneously acquiring N different subpages, while the top curve plots the cost of N nodes simultaneously acquiring the same subpage. Thus, the bottom curve illustrates just the effects of bus contention, while the top curve illustrates the effects of contention for a subpage [7]. As the figure illustrates, acquiring a subpage in ME state takes less than 10 μ-secs., and this overhead scales well under high contention.

4.2 Implementations of Autonomous and Conventional Locking

AL and CL can be decomposed into two components, a synchronization component, and the component which updates the lock space. When a transaction requests a lock under CL, process synchronization is used to communicate this request to a lock manager. In contrast, when a transaction requests a lock under AL, data synchronization is used to process the request. Section 4.2.1 discusses the implementation of synchronization for CL, while section 4.2.2 discusses the

[6]There are two versions of gsp, gsp.nwt (no-wait) and gsp.wt (wait). We use gsp.wt, which causes the cache to stall the CPU until the subpage can be obtained in ME state.

[7]The slight bump in the top curve for two nodes results from the allocation of requestors to different rings of the multiprocessor.

implementation of synchronization for AL. The component which updates the lock space is discussed in section 4.2.3.

4.2.1 Synchronization in Conventional Locking

We considered a number of alternatives for implementing process synchronization. After benchmarking various UNIX facilities for communication, we concluded that on our implementation platform, a FIFO queue is the most efficient method for implementing inter-process communication. The standard (UDP and TCP) socket interface was also considered, but the performance was unacceptably slow.

We took advantage of the cache coherency protocol to implement a highly concurrent (almost wait-free) FIFO queue. A circular FIFO is implemented by maintaining two variables in the header, `next−available` and `next−service`, which denote the next free slot in the queue, and the slot which is currently being serviced. The only time any part of the FIFO data structure is locked is when the enqueueing transaction "reserves" a slot. Specifically, each enqueueing process (transaction) first locks the FIFO header, reads and increments `next−available`, then unlocks the FIFO header. To obtain maximal concurrency from the cache coherency protocol, the header is stored on a separate subpage from the remainder of the FIFO, and each slot is allocated on a separate subpage. Locking the header is implemented using the atomic subpage lock mechanism described in section 4.1.1. Once a slot is acquired, the requestor knows which slot to place its request into. Once the request slot has been filled, a synchronization variable `enqueue−complete` is set, indicating to the lock manager that the enqueue is complete. The requestor then waits for the service to complete by busy-waiting on the `service−complete` slot variable.

The LM waits for the `next−service` slot to be filled by busy-waiting on `enqueue−complete`. Once `enqueue−complete` is set, the LM processes the request (see section 4.2.3), then places the return value of the request into the same slot, and finally sets the `service−complete` slot variable, indicating to the requestor that the service is complete.

4.2.2 Synchronization in Autonomous Locking

The strategy for synchronization in AL is much simpler than in CL. In AL, once the appropriate LCB is determined by hashing, an atomic subpage lock is acquired on the LCB. Once the lock acquisition or release is processed, the subpage lock on the LCB is released.

4.2.3 Lock Acquisition and Release

Identical lock acquisition and release code is used by both AL and CL. The lock acquisition procedure requires three arguments, the address of the LCB of the database object, the requested lock mode, and the TID of the requesting transaction. A lock is granted if the requested mode does not conflict with the strongest mode of any holder (one comparison), and the strongest mode of any waiter (another comparison). Prior to granting the lock, the TID and requested mode tuple are added to the list of lock holders. If the lock is not granted, this tuple is added to the list of waiters. Since our experiments focused on the case where concurrency was possible, we do not discuss the details of granting locks to waiting transactions.

4.3 Experimental Methodology

In order to best simulate independent transactions in an SM database system, our testbed implements each transaction as a separate process. Furthermore, in order to isolate the computational load of one transaction from another, each transaction was allocated to a separate, dedicated node on the KSR1, and prevented from being swapped out by pinning the process into memory. Shared variables are implemented by requiring all processes to open a shared file, then the `mmap(2)` system call is used to map the file into the address space of each process. Once this mapping is performed, access to shared memory is performed like any other data reference, i.e., through pointers or structure access.

In the interest of a fair comparison and potential scalability, all our workloads were run on a "mixed" ring. In a mixed ring workload, half the transactions execute on Ring 1, and the other half execute on Ring 2. In order to obtain consistent timing measures, prior to actually timing code segments, we "preloaded" the cache by executing these segments without timing them. This scenario is consistent with the high probability of cache hits which would arise in the case of frequently requested locks. The KSR timer routines used are of the "user_timer" variety, and measure *elapsed* time (as opposed to *system* time). This routine reports time in units of 8 machine cycles, where each cycle is 50 ns.

In order to create contention for locks, the standard barrier synchronization method is used. For example, to create the conditions under which N processes will concurrently request a lock, a shared variable (`sem`), initialized to zero, is incremented by each process. Then, each process waits, by busy-waiting on sem, until sem is equal to N.

4.4 Experiments

We conducted a number of experiments which compared CL to AL. To quantitatively assess how each strategy performs under contention, our experiments measured the average time it took for a transaction (running on a dedicated node) to perform a lock request under different degrees of contention. Our comparisons focus on the cases where concurrency is possible (i.e., the acquisition of different locks, or the acquisition of the same lock in shared mode). The same lock acquisition and release code was used for CL and AL, but different synchronization mechanisms were used to ensure mutually exclusive access the LCB. In CL, ensuring a mutual exclusion is implicit, since, for a given LCB, only one GLM is allowed to update it. To ensure mutual exclusion in AL, subpage locks are held on the LCB for the duration of any reading and updating by a lock operation.

In all the graphs, a solid line represents costs associated with AL, while the dotted line represents costs associated with CL. The x axis represents the degree of concurrency, i.e., how many processes are concurrently performing the measured operation. The y axis represents the cost, per process (also interpreted as or per transaction), to perform the measured operation. Along the x axis, data points were sampled for 1, 2, 4, 8, 16, 24 and 32 concurrent processes (transactions) [8]. Based on multiple runs of the same experiment, 95 percent confidence intervals (where we are 95 percent confident that the measurement error falls within this interval) were computed for each sample point, and are plotted in each graph.

Acquiring Hot Locks

For AL and CL, the cost of acquiring a hot lock, along with the synchronization overheads, are plotted in figure 5. Figures 5a and 5b graph the same experiments, figure 5a plots contention up to 8 processes while 5b plots contention up to 32 processes.

The bottom dotted line plots *process* synchronization costs, used in CL, while the bottom solid line plots *data* synchronization costs, used in AL. For these curves, the y axis indicates the average cost of performing a null critical section with the two synchronization mechanisms. Data synchronization is measured as the overhead required to obtain and release a subpage lock in ME state. Process synchronization measures the overhead to use the FIFO queue mechanism when

[8] Note that, in order for an experiment to compare the *same number of concurrent transaction processes* in AL and CL, an *additional* (dedicated) node serves as the global lock manager in CL.

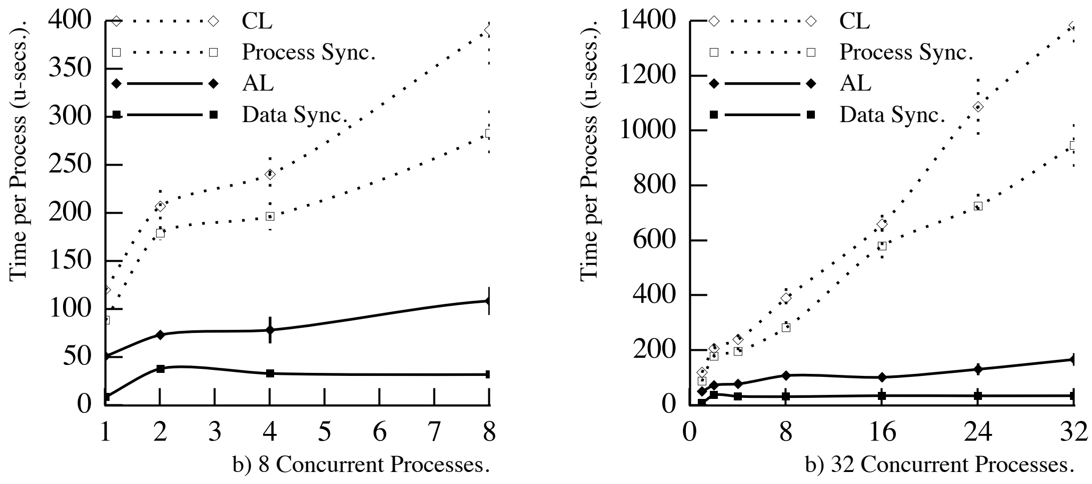

Figure 5: Hot Lock Acquisition and Associated Process and Data Synchronization Costs.

only a null message is sent. These two curves indicate that under any degree of contention, the inherent synchronization overheads for CL are significantly greater than for AL.

The top dotted line and top solid line plot the cost of acquiring a hot lock for CL and AL respectively. The cost of acquiring a hot lock was measured by having N transactions (each running on a separate node) concurrently attempt to acquire a shared lock on a single record. For hot lock acquisition, the y axis indicates the average lock acquisition time per transaction, in μ-secs. As is evident from the figure, under any level of contention, using AL to acquire a hotlock offers significant performance advantages over using CL. The high overheads of synchronization under contention in CL are reflected in the total cost to acquire a lock for CL. For CL, the overheads of synchronization dominate the costs of actually performing the lock acquisition. However, for AL, the overheads of synchronization do not overwhelm the costs of performing the lock acquisition. The difference between the top and bottom lines for AL and CL indicate the cost of actually executing the critical section, and are about the same for AL and CL.

Acquiring Batched Locks

Given that batched locks (sending more than one lock request in a single message) will mitigate the negative performance impact of process synchronization, we compared the performance of AL against a CL implementation in which multiple lock requests were requested per message. For CL, sending N requests for N different locks to the lock manager in a single message constitutes a batch of N. Since the notion of batch locks does not apply to AL, AL acquired the N different locks one at a time. Figure 6 compares the performance of AL and CL for batches of 10 and 100 locks. For batches of 10 locks, the lock acquisition cost of AL and CL are about

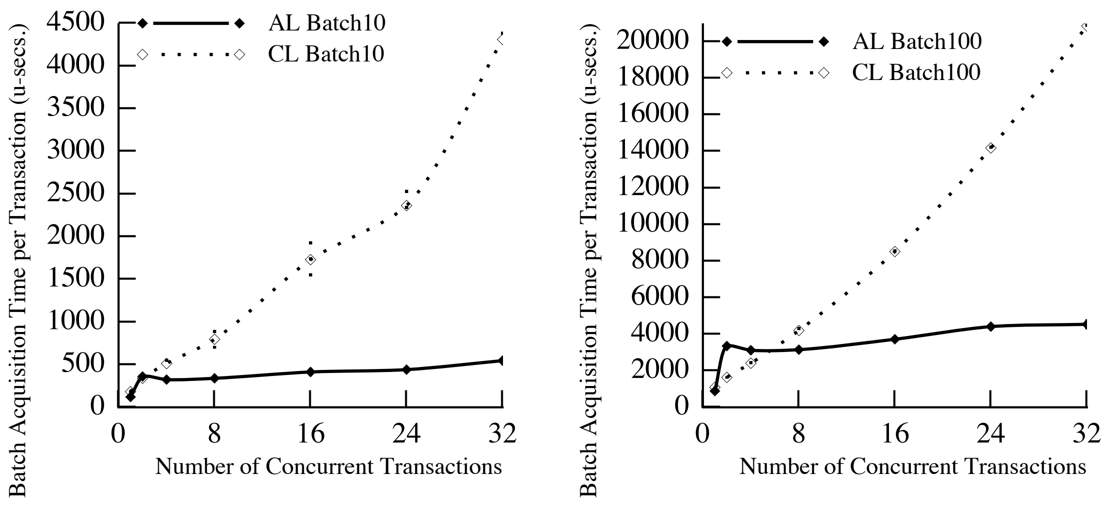

Figure 6: Acquiring 10 and 100 Batched Locks

equal for up to four concurrent processes, and for more than four concurrent processes, AL again outperforms CL. For batches of 100 locks, CL outperforms AL for concurrency levels of two and four, but once concurrency level reaches eight, AL outperforms CL.

4.5 Summary

In order to quantitatively compare AL and CL, we implemented both locking strategies on a large scale cache coherent shared memory multiprocessor. For both these implementations, we exploited two features of the multiprocessor to achieve low latency: the cache coherency protocol, and the atomic subpage lock facility. We emulated a database environment consisting of independent concurrent transactions by assigning each transaction to a separate, dedicated node. Contention for hot locks was assessed by measuring the case where multiple transactions concurrently request the same lock in a compatible mode. Our experiments also measured the synchronization component of lock acquisition.

These experiments reveal the overheads inherent in using data and process synchronization to implement lock operations as critical sections. Consider N processes which simultaneously attempt to access a single critical section. The *average* cost, per process, will be at least:

$$\sum_{i=0}^{N-1} \left(\frac{i}{2} * CS\right)$$

Where CS is the execution time of the critical section, the cost to execute the lock acquisition code for both AL and CL. In addition to the cost of actually executing the critical section, the cost of a lock operation also includes synchronization overheads. The synchronization component is not so easy to model analytically, since it depends on subtle interactions between the software and the hardware of the implementation platform. In an ideal situation, the synchronization overhead would be negligible. However, our experiments show that the synchronization overheads for CL overwhelm the cost of executing the lock acquisition code (the critical section). In contrast, the synchronization component for AL is negligible, and this translates into better overall performance for AL, especially under high contention.

Although our experiments were limited to measuring contention up to 32 nodes [9], the multiple ring architecture of the KSR1 suggests that our autonomous lock manager will scale up to hundreds of nodes. In our performance tests on a 2-ring KSR1, processes were evenly divided between the two rings. For autonomous locking, this allocation maximized the more costly inter-ring cache-to-cache traffic. By maximizing the inter-ring traffic, our performance measures reflect the worst case type of traffic which would occur in a system with more than two rings. Yet, AL was shown to dominate CL for a wide range of parameters.

5 Conclusion

It is well known in the multiprocessor literature that data synchronization is more efficient than process synchronization. Our study demonstrates the suitability of using data synchronization for implementing a critical component of a database system, the lock manager. Our performance studies quantify the performance of autonomous locking (which uses data synchronization) vs. conventional locking (which uses process synchronization).

For an SM database system, autonomous locking offers significant performance advantages over conventional locking. AL eliminates the notion of a global lock manager, allowing all transaction managers to acquire locks directly from shared memory. By eliminating the GLM, the overheads inherent in inter-process communication are eliminated, thus providing much better performance from lock acquisition and release operations. Furthermore, with AL, no explicit partitioning of the database is necessary to allow the concurrent acquisition of database locks. Under any degree of contention, AL outperforms CL for requests for a single lock. When contention is high (i.e., for hot lock operations), the performance of AL is nearly an order of

[9]Due to configuration parameters restricting the allocation of processors, our experiments used *only* 32 of the 64 processors

magnitude better than CL.

There has been considerable debate about whether a SM implementation platform is suitable for database systems. Based on a TP1 benchmark performed on a Sequent Symmetry shared memory multiprocessor, [21] concludes that a SM database system can deliver very high performance. In [22], an analytical and simulation study compares SN (shared nothing), SD, and SIM (shared intermediate memory [10]). This comparison concludes that the data sharing architectures, especially SIM, are more resilient to transaction load surges. In [3, 4], simulation studies compare SN, SD, and SM (called SE (shared everything) in this reference), and concludes that SM outperforms SN and SD by a fairly wide margin. However, none of these studies have measured the performance of the lock manager component of the database in isolation.

Other researchers have argued that SM database systems are not well suited for high performance database implementations, claiming that the platforms that these database systems are implemented on are not scalable and do not have good failure properties [5]. We have shown that this need not be the case:

- Our prototype lock manager implementation and performance studies indicate that *it is possible* to construct a scalable implementation of a lock manager. For the KSR1 implementation platform, our measurements show that the impact of high data sharing contention on shared system resources is marginal. This indicates that, in addition to efficiently supporting a lock manager, any concurrent operation on shared data will be scalable.
- For autonomous locking, we have also described recovery mechanisms which are sufficient to ensure a failure atomic lock space, and mechanisms which ensure a cascadeless system in the context of fine granularity locking.

Current research in multiprocessor architectures and in cache coherent DSM architectures promise to enhance the failure properties of shared memory platforms. In anticipation of these features, we have described mechanisms for supporting a failure atomic lock space for autonomous locking. Two basic mechanisms enable the reconstruction of a failure atomic lock space after a node crash has occurred. First, prior to acquiring any locks, a transaction must

[10]The SIM model is slightly different than the shared memory model. In SIM, a shared intermediate memory serves as a global shared buffer for all nodes.

stable log its transaction identifier. Secondly, as is normally done in any lock manager implementation, the transaction identifier and the granted lock mode are stored in the lock control block. Under any page migration and node crash scenario, these two mechanisms enable a failure atomic lock space by providing the recovery procedure sufficient information to remove all locks held by aborted transactions and reestablish any locks (which were lost due to a node crash) held by non-aborted transactions.

Autonomous locking can also be integrated with fine-granularity sharing environment. By logging redo information and stable logging undo information for updated data objects prior to page migration, inter-node cascading aborts can be avoided in a fine-granularity sharing environment. AL can coexist in a network of systems which use CL. To integrate AL into networks of systems which use CL, a single process on the AL system could be used as an intermediary to acquire and release locks under AL, but to remote CL systems, it will serve as a global lock manager.

References

[1] J. Archibald and J. Baer. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. *ACM Transactions on Computer Systems*, 4(4):273–298, November 1986.

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

[3] A. Bhide. An Analysis of Three Transaction Processing Architectures. *VLDB*, 14:339–350, September 1988.

[4] A. Bhide and M. Stonebraker. A Performance Comparison of Two Architectures for fast Transaction Processing. *IEEE Proc. 4th Intl. Conference on Data Engineering*, pages 536–545, February 1988.

[5] D. DeWitt and J. Gray. Parallel Database Systems: The Future of High Performance Database Systems. *CACM*, 35(6):85–98, 1992.

[6] R. Foster. Private communication. *KSR Research*, March 1994.

[7] J. Garza and W. Kim. Transaction Management in an Object-oriented Database System. *Proc. 1988 ACM SIGMOD International Conference on Management of Data*, pages 37–45, June 1988.

[8] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1993.

[9] J. N. Gray, R. A. Lorie, G. R. Putzulo, and I. L. Traiger. Granularity of Locks and Degrees of Consistency in a Shared Database. In Michael Stonebraker, editor, *readings in Database Systems*, pages 94–121. Morgan Kaufmann, 1988.

[10] J.Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH Multiprocessor. *To Appear in the 21th International Conference on Computer Architecture*, April 1994.

[11] P. Keleher, A. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. *To appear in the 1994 Winter Usenix Conference*, 1994.

[12] C. Mohan and I. Narang. Recovery and Coherency-control Protocols for Fast Intersystem Page Transfer and Fine-Granularity Locking in a Shared-Disk Transaction Environment. *VLDB*, 17:193–207, 1991.

[13] C. Mohan, I. Narang, and S. Silen. Solutions to Hot Spot Problems in a Shared Disks Transaction Environment. *IBM Research Report, IBM Almaden Research Center*, December 1990.

[14] L. D. Molesky and K. Ramamritham. Multi-Node Multi-Level Transactions. *Submitted to Proc. 1994 VLDB*, Feb 1994.

[15] J. Moss. *Nested Transactions: An Introduction.* Bharat Bhargava, ed. Van Nostrand Reinhold, 1987.

[16] P. Peinl, A. Reuter, and H. Sammer. High Contention in a Stock Trading Database: A Case Study. *Proc. 1988 ACM SIGMOD International Conference on Management of Data*, pages 260–268, June 1988.

[17] E. Rahm. Concurrency and Coherency Control in Database Sharing Systems. *Technical Report, University of Kaiserslautern, Germany*, December 1991.

[18] E. Rahm. Use of Global Extended Memory for Distributed Transaction Processing. *Proceedings of the 4th Int. Workshop on High Performance Transaction Systems, Asilomar, CA.*, September 1991.

[19] Kendal Square Research. *KSR1 Principles of Operation.* KSR Research, Waltham, Mass., 1992.

[20] A. Reuter. Concurrency on High-Traffic Data Elements. *Proc. 1982 ACM Symposium on Principles of Database Systems*, pages 83–92, March 1982.

[21] S. Thakkar and M. Sweiger. Performance of an OLTP Application on Symmetry Multiprocessor System. *Proceedings of the Seventeenth Annual International Symposium on Computer Architecture*, pages 228–238, May 1990.

[22] P. Yu and A. Dan. Performance Evaluation of Transaction Processing Coupling Architecutres for Handling System Dynamics. *IEEE Transactions on Parallel and Distributed Systems*, 5(2):139–153, June 1994.