Database Locking Protocols for Large-Scale Cache-Coherent Shared Memory Multiprocessors: Design, Implementation and Performance

Lory D. Molesky and *Krithi Ramamritham*
Department of Computer Science
University of Massachusetts
Amherst Ma. 01003
e-mail: lory@cs.umass.edu, krithi@cs.umass.edu

June 6, 1995

Abstract

Significant performance advantages can be realized by implementing a database system on a cache-coherent 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 examine two approaches to the implementation of locking in a cache-coherent shared memory multiprocessor database system. The first approach, shared-memory locking (SML), allows each node (processor) of the multiprocessor to acquire and release locks directly via the use of cache-coherent shared memory. The second approach, message-passing locking (MPL), typically requires messages to be sent to a lock manager, located on a remote node.

Our empirical evaluation of these approaches on the KSR-1 multiprocessor indicates that for most database locking traffic patterns, the performance of SML is substantially superior to that of MPL. For instance, when contention is high, the performance of SML is nearly an order of magnitude better than MPL.

1 Introduction

Recent advances in multiprocessor technology have made *large-scale* cache-coherent multiprocessors possible, enabling the construction of high performance shared memory (SM) database systems. Commercial multiprocessors, such as the KSR-1, have been constructed which scale to thousands of nodes (processor/memory pairs), and offer low-latency, high bandwidth access to shared memory via a hardware based cache coherency protocol [18]. On an SM database system of this caliber, it is conceivable that thousands of transactions may execute concurrently, and, as a result, create very high contention for shared resources.

Typically, database systems employ locking to conservatively enforce transaction isolation, such as serializability or cursor stability [6]. For example, prior to performing a read or write on a database item, it is necessary to obtain a read or write lock on this database item. In order to minimize transaction execution time, these lock requests should be granted in a timely fashion. However, when concurrent lock requests are issued by transactions executing on many different nodes, the performance of transactions may be adversely affected unless the lock manager is carefully designed and implemented. These factors motivate our study of lock management for an SM database system constructed on a large-scale multiprocessor.

In any multiprocessor algorithm or protocol, the potential for concurrent execution must be weighed carefully against the overheads of communication, synchronization and obtaining exclusive access to a shared resource. With this in mind, this paper considers design and implementation strategies, and the performance of two basic SM database lock manager architectures. One architecture is based on access to the lock space (the data structure which maintains the status of locks) via updates to shared memory, and the other based on communication with a server. We refer to these competing strategies as SML (shared memory locking) and MPL (message-passing locking).

The implementation of SML and MPL exploit a novel synchronization feature of the KSR-1 multiprocessor, enabling very efficient implementations of these approaches. This synchronization primitive allows a process (transaction) to temporarily lock a cache line, excluding other processes from accessing the data in this cache line. For SML, this feature enables a very efficient implementation of critical sections, while for MPL, this feature enables a very efficient implementation of message-passing using shared memory.

We have conducted an evaluation of SML and MPL on the KSR-1. Our experiments demonstrate that under common locking scenarios, the locking cost for MPL increases linearly with contention while the locking costs for SML remain fairly constant. The SML 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 [19, 14, 11]. When multiple transactions attempt to lock the same data object in a compatible mode, e.g., read locks, (creating a *hot lock*), the lock manager becomes a performance bottleneck. In fact, under high contention for hot locks, SML's performance is better than MPL's by almost an order of magnitude.

Although providing good performance is extremely important, the requirements of database systems dictate that recovery properties also play a significant role in the choice of an SM lock management architecture. For SML and MPL, we compare crash recovery mechanisms and their associated overheads in the context of ensuring that one or more node crashes does not necessarily require the abort of active transactions running on surviving nodes.

This paper is structured as follows. Section 2 reviews some background material, including a general discussion of database locking and our assumptions about the underlying hardware. Section 3 discusses two approaches to the implementation of lock management on a cache-coherent shared memory multiprocessor, SML and MPL. Section 4 discusses our KSR-1 implementations of SML and MPL. This discussion focuses on the details of the implementation of obtaining exclusive access to the lock space in SML, and the implementation of communication in MPL. Our performance studies of these implementations are given in section 5. A comparison of crash recovery mechanisms and overheads for SML and MPL is provided in section 6. Related work is discussed in section 7, and our conclusions are presented in section 8.

2 Background

In this section, we review background material which motivate the performance and implementation considerations involved in the design of an SM database lock manager. First, we present a general discussion of database locking. Locking is the preferred method for enforcing transaction isolation in commercial relational and object-oriented database systems, and is discussed in section 2.1. Section 2.2 provides an overview of the KSR-1 multiprocessor, which we used to implement and evaluate our prototype lock managers.

2.1 Locking in Database Systems

To ensure isolated transaction executions, virtually all commercial database systems use locking. The basic lock modes are shared (S) and exclusive (X). An X lock on a 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. In addition to the basic lock modes, intention locks are also included to facilitate multigranularity locking [7]. In a multigranularity locking protocol using intention locks (i.e., IX, IS, SIX), the database is characterized by a hierarchy (on a directed acyclic graph), where 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.

There are many cases where locks may be held concurrently by many different transactions. Clearly, many transactions may hold locks on *different* database items without conflict. In addition, transactions may concurrently hold locks on the *same* data item or index node. While *hot spots* [12, 14] 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.

Hot locks may arise under many situations in multi-node database systems. For example, consider multiple query transactions, each executing on a separate node, which simultaneously attempt to lock a record r in S mode. In this case, since there are many concurrent compatible requests, the lock on r is hot. As another example, 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 factors, hot locks, and the likelihood of concurrent requests for locks on different database objects, motivate our study of lock manager performance.

2.2 KSR-1 Multiprocessor Overview

In a cache-coherent shared memory multiprocessor, a coherency protocol, typically implemented in hardware, ensures that any read operation sees the most recently written value for any data item [9].. 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, operation 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. 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.

The KSR-1 is an example of a large-scale cache-coherent multiprocessor, and can be configured to support up to 1088 nodes. All memory in the system is cache memory, and thus this machine has been referred to as a COMA (Cache-Only Memory Architecture). Low latency access to cache memory is achieved with a hierarchy of slotted rings [18]. The first level of the hierarchy, called Ring 0, implements coherency among groups of 32 nodes. The second level, Ring 1, inter-connects up to 34 Ring 0's, yielding up to 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 maintained between local caches, and all data and instruction references are made through the subcache, using the local cache as an intermediary (if necessary). The unit of coherency between nodes is 128 bytes, and is called a *subpage*.

On the KSR-1, the two level cache interconnect implements a five 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 0, remote Ring 0's 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 [1], local cache access 20 - 24 cycles, local ring access 175 cycles, remote ring access 600 cycles [18].

These two subsections, 2.1 and 2.2, indicate that (1) in many cases, concurrency of database lock management operations is possible, and (2), a large-scale cache-coherent multiprocessor, such as the KSR-1, could potentially support a large number transactions which concurrently execute on multiple nodes. With these factors in mind, next, in section 3, we consider the two basic approaches for implementing a lock manager – SML (shared memory locking) and MPL (message passing locking).

3 Lock Management Options

On a cache-coherent SM multiprocessor, the simplest approach to implementing database locking is to allow any transaction to acquire and release locks on database objects via operations on a (global) lock table stored in shared memory. We call this strategy *Shared Memory Locking*, or SML. An alternative strategy is to designate one node to manage each database object, and, when the needed database object is managed on a remote node, message passing is used to communicate lock requests[2]. This strategy is called *Message Passing Locking*, or MPL.

The MPL approach has been taken in many shared-disk (SD) database systems [11, 15, 17, 22]. SD systems differ from SM systems in that although both have access to shared disks, SD systems do not have shared memory. In SD systems, there are two basic models of MPL – *centralized* and *distributed* [15]. In a centralized MPL architecture, a single node is designated to serve as the lock manager for all database objects. A centralized MPL architecture imposes an inherent sequential bottleneck on the lock acquisition/release process. This has been recognized,

[1] A clock cycle on the KSR-1 is 50 ns.

[2] Note that when the database object is managed at the local node, message passing is not required.

and to mitigate this problem, distributed MPL architectures have been proposed. In distributed MPL, multiple lock managers can be used, where each manages a distinct partition of the lock space [15].

Of course, the absence of shared memory in an SD system precludes SML. However, since SML is possible in a SM database system, our objective is to quantitatively assess the performance differences between SML and MPL. In general, we would expect that SML would exhibit better performance than MPL for the following reasons:

- SML eliminates inter-process communication (IPC) overheads.
- SML reduces sequential bottlenecks, thus increasing potential concurrency.

By eliminating the lock manager processes, SML eliminates a number of inherent inefficiencies associated with it. Since all lock acquisitions and requests operate directly on shared memory, no IPC is required in SML. In MPL, even 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 SML, concurrent lock requests for different locks are not subject to this single server type of bottleneck. Furthermore, in SML, concurrent requests for the same lock will incur sequential bottlenecks only at the lowest level of granularity – that is, only during reference to the actual shared data.

Additional implementation considerations may also significantly affect the performance of lock management. If not carefully designed, overheads associated with obtaining exclusive access to the lock space in SML and MPL can substantially degrade performance. Ensuring exclusive access to a particular data structure is an efficient and practical method to serialize groups of operations on the data structure, avoiding potential anomalies due to the arbitrary interleaving of operations issued by concurrent processes.

Thus, the performance of SM database lock management is influenced by a number of factors, including (1) overheads associated with obtaining exclusive access to the lock space, (2) IPC overheads, and (3) sequential bottlenecks. We consider these factors next in section 4, where we examine the details of efficient implementations of SML and MPL.

4 Multiprocessor Implementation of MPL and SML

In this section, we describe the implementation of SML and MPL. SML and MPL consist of two components, (1) obtaining exclusive access to the lock space, and (2) updating the lock space (to reflect a lock acquisition or release). Updating the lock space is basically identical for both SML and MPL, and is discussed in section 4.1.

However, in SML and MPL, the techniques used to obtain exclusive access to the lock space are much different. In SML, exclusive access is obtained through a simple application of the KSR-1 synchronization primitives. In MPL, a single server is responsible for updating a specific database item, and thus exclusive access is implicit. However, when the server in question is located on a remote node, it is necessary to perform IPC to communicate the request and the reply. Our implementation of MPL exploits cache-coherent shared memory and the KSR-1 synchronization primitives in order to perform IPC efficiently.

Given the KSR-1's synchronization primitives are used for both these locking approaches, we discuss these synchronization primitives in section 4.2, then the implementations of SML and MPL are discussed in sections 4.3 and 4.4.

4.1 Lock Acquisition and Release

Consider acquiring a lock on a database object. 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, a data structure which stores locking information specific to one database object. 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. If the requested mode is compatible with the mode stored in the LCB, and there are no conflicting waiters (for fairness), a tuple containing the requesting transaction's identifier 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 per transaction list of acquired locks is another lock management data structure which is maintained the transaction management system. This data structure enables an efficient implementation of transaction aborts and commits. However, to simplify our performance experiments, our experiments did not measure this component of the transaction management system.

Next we discuss the synchronization primitives available on the KSR-1. These synchronization primitives are used in SML to obtain exclusive access to LCB's, and are used in MPL to implement IPC.

4.2 KSR-1 Synchronization Primitives

The KSR-1 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 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 [3]. 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. We will discuss shortly how critical sections needing multiple subpages can be handled, but first we consider the efficiency of the KSR-1's subpage synchronization mechanism under high contention.

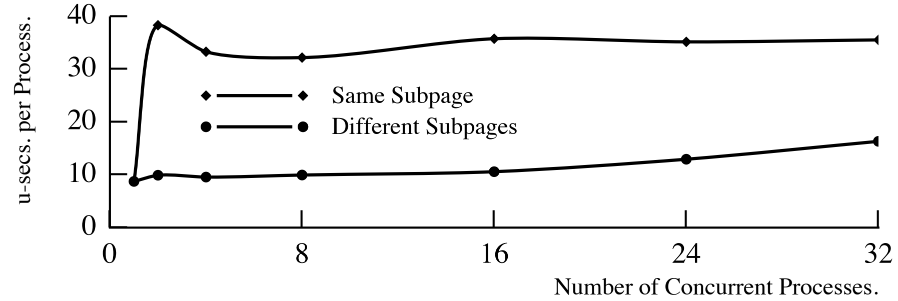


Figure 1: KSR-1 Atomic Subpage Synchronization Costs.

This is illustrated in figure 1, which plots the overheads involved in acquiring a subpage in ME state. The bottom curve plots the cost of N nodes each simultaneously requesting different subpages, while the top curve plots the cost of N nodes simultaneously requesting 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 specific subpage[4]. As the figure illustrates, acquiring a subpage in ME state takes less than 10 μ-secs., and this overhead scales well under high contention.

[3] 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.

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

4.3 Implementation of Exclusion in SML

Using the KSR-1's synchronization primitives, it is possible to construct simple and efficient implementation of SML. In order to ensure that potentially incorrect interleavings of operations on the same LCB do not occur, each transaction obtains exclusive access to a specific LCB (or LCB's) before an update is performed. This is done by placing the LCB's in shared memory, then access to a specific LCB is performed inside a critical section.

Figure 2 illustrates the pseudo-code (using the syntax of the C programming language) used to obtain exclusive access to the `acquire_lock()` procedure. First the appropriate LCB is determined by hashing (line 1). Prior to accessing a specific LCB, `gsp(&LCB)` is executed, ensuring that no other transaction, whether it be on the same or different node, can obtain read or write access to this LCB (line 2). Then the code for lock acquisition is executed (`acquire_lock()`), and the subpage storing the LCB is released with `rsp(&LCB)` (line 4).

In our experiments, a single subpage was sufficient to store an LCB. However, if a single LCB spanned multiple subpages, the same locking mechanism can be employed to obtain exclusive access to this LCB. That is, by convention, we can require that transactions execute `gsp` on a (single) designated subpage among the multiple pages.

Since our objective was to examine the performance of independent transactions running on different nodes, we implemented transactions as independent processes. To accurately model these independent transactions, shared data structures were 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. Note that this implementation strategy offers a great deal of flexibility – once this data is cached, disk I/O is not required, but can be forced with the appropriated system calls.

For SML, data structures such as the hash table and the LCB's were implemented via

SML Lock Acquisition

```
1. Hash record name to obtain LCB;
2. gsp(&LCB);
3. acquire_lock(...);
4. rsp(&LCB);
```

Figure 2: Lock Acquisition Using Shared Memory.

shared files, while for MPL, as we will see next, shared files were used to implement the FIFO communication buffer.

4.4 Implementation of IPC in MPL

We considered a number of alternatives for implementing IPC between the client and the server in MPL. After benchmarking various UNIX facilities for communication, we concluded that on the KSR-1, 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. Thus, instead of using a standard UNIX interprocess communication protocol, we implemented message passing by using shared memory.

By exploiting cache-coherent shared memory, we constructed a highly concurrent circular FIFO queue used to transmit messages between MPL clients and the MPL server. This queue permits concurrent enqueues (client requests), and allows concurrent enqueues and dequeues (done by the server) on different entries in the queue. Maximal concurrency is achieved by exploiting the technique of “slot reservation” discussed in [4, 8].

The data structure for this FIFO queue consists of a few bytes of header information and `N` slots. Each slot is used to transmit messages between one MPL client and the MPL server. The 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.

An MPL client makes a request to an MPL server by reserving a slot, formatting a request, then setting a synchronization flag (indicating to the MPL server that the request is ready for service). The code for the MPL server and MPL client is shown in figure 3, and a detailed explanation of this code follows. First we discuss the MPL client.

- Slot Reservation
  The reservation of a slot is performed inside a critical section, as shown in lines 1-4 of the MPL client. The critical section is enclosed between the `gsp` and `rsp` calls. The argument to these calls is the base address of the FIFO (&Fifo), ensuring that at most one process will update the `next_available` slot (stored in the header). Inside the critical section, a local variable, `i`, is first set to the next available slot, then `next_available` is incremented by 1 modulo `N` (the number of slots).

- Formatting the Request and Synchronizing with the Server
  Once a slot (`i`) is reserved, the `service_complete` synchronization variable associated with

MPL Client

```
1. gsp(&Fifo);
2. i = Fifo.next_available;
3. Fifo.next_available =
       (Fifo.next_available + 1) % N;
4. rsp(&Fifo);
5. Fifo[i].service_complete = 0;
6. Format Request;
7. Fifo[i].enqueue_complete = 1;
8. while (Fifo[i].service_complete == 0) ;
```

MPL Server

```
1. while (1) {
2.     i = Fifo.next_service;
3.     while (Fifo[i].enqueue_complete == 0) ;
4.     Service Request;
5.     Fifo[i].service_complete = 1;
6.     Fifo[i].next_service =
           (Fifo[i].next_service + 1) % N;
7. }
```

Figure 3: Client and Server Code for MPL.

slot `i` is set to 0 (line 5). The request is then formatted by entering information in the slot such as the transaction identifier, the operation requested (acquire lock or release lock), and the requested mode of the lock (S, X, etc.) (line 6). Once the request is formatted, the slot variable `enqueue_complete` is set to 1, indicating to the server that the request is ready for service (line 7). Finally, the client waits for the server to complete the request by busy-waiting on the service_complete slot synchronization variable (line 8).

Next, we discuss the details of the MPL server.

- MPL Server Operation:
  The MPL server waits for the `next_service` slot to be filled by busy-waiting on `enqueue_complete`. Once `enqueue_complete` is set, the request, such as a lock acquisition or release, is serviced. The return value of the request is then written into the slot, then the `service_complete` slot variable is set to 1, indicating to the requestor that the service is complete.

At the client end, a high degree of concurrency is possible. To eliminate unnecessary subpage conflicts, the header is stored on a separate subpage from the remainder of the FIFO, and each slot is allocated on a separate subpage. Thus, slot reservations require temporarily locking only the header, and a reservation can proceed concurrently with enqueues, without subpage synchronization conflicts. Furthermore, concurrent enqueues can also occur without subpage synchronization conflicts.

To summarize, we have attempted to construct as efficient an implementation of SML and MPL as possible. To this end, wherever possible, we have utilized the KSR-1's synchronization primitives. In the interests of a fair comparison, we have not used the TCP/IP to implement MPL, instead, we exploited shared memory in the MPL implementation. Moreover, in order to avoid any potential synchronization conflicts, a separate subpage was allocated for each slot of the MPL FIFO queue. Next, we discuss the performance of these implementations.

5 Performance Studies

In this section, we present performance benchmarks of prototypes of the SML and MPL lock manager designs on the KSR-1 multiprocessor. Our experiments were run on a 64-node (2 rings of 32 processors) KSR-1. Up to 32 of the 64 processors were simultaneously used[5]. A hardware timer was used to determine the execution time of the SML and MPL implementations.

The main objective of our experiments was to assess the performance of SML and MPL under various degrees of contention in 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 [6]. For lock acquisitions, control returns without waiting when the requested lock is compatible with the current lock mode or the lock is not currently held by any transaction (thus granting the lock request). Recall that the significant operations which need to be performed after compatibility is determined are that the transaction identifier and requested mode are appended to the LCB. Since the code for lock release is similar to the lock request code, our comparisons focused on timing the code for lock requests.

To meet these experimental objectives, we measured the performance of two basic lock request patterns under varying degrees of contention.

- Performance of Hot Locks.

  Under high contention, requests for hot locks will cause the most significant performance degradation for both SML and MPL. For hot locks under SML, accesses to the LCB storing the hot lock are serialized via a critical section. For hot locks, regardless of the MPL partitioning scheme, all lock requests will be sent to the same global lock manager.

- Performance of Batched Locks.

  In the MPL implementation, one way to reduce the per-lock overhead of lock acquisition is to request many locks (in *batches*) at a time. We compare the performance of batched locks in MPL to that of SML.

[5] Regrettably, an operating system bug (later fixed on the KSR-2) prevented us from reliably using a greater number of processors.

For both of these traffic patterns, for each experiment, we report two measurements. The primary measure is the elapsed time to acquire a lock, and the secondary measure, a component of the first, is the time spent for obtaining exclusive access or performing IPC.

Section 5.1 discusses our experimental methodology, and section 5.2 discusses the results of our experiments.

5.1 Experimental Methodology

In order to accurately model 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 KSR-1, and prevented from being swapped out by pinning the process into memory.

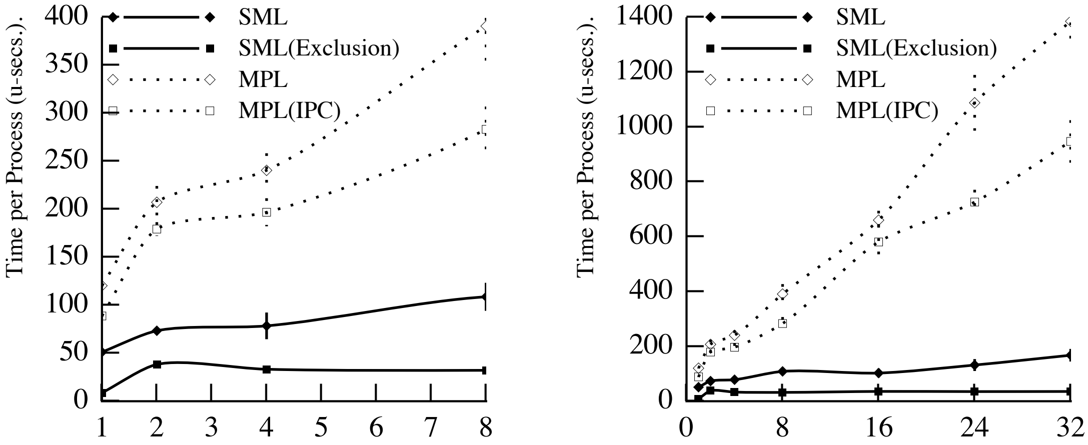
In the interest of a fair comparison and potential scalability, all our workloads were run on two ring 0's. In our experiments, half the transactions execute on one Ring 0, and the other half execute on the other Ring 0. In order to obtain consistent timing measures, prior to actually timing code segments, we "preloaded" the cache by executing these segments without timing them. This scenario models that of a "warm" database, one where a substantial part of the database management code is resident in the instruction cache. 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.

5.2 Experiments

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. The same lock acquisition and release code was used for MPL and SML, but, as discussed in sections 4.3 and 4.4, different methods were used to obtain exclusive access to LCB's.

In all the graphs, a solid line represents costs associated with SML, while the dotted line represents costs associated with MPL. 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 per transaction), to perform the measured operation.



a) 8 Concurrent Processes.

b) 32 Concurrent Processes.

Figure 4: Hot Lock Acquisition and Associated Process and Data Synchronization Costs.

Along the x axis, data points were sampled for 1, 2, 4, 8, 16, 24 and 32 concurrent processes (transactions) [6]. 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.

5.2.1 Acquiring Hot Locks

For SML and MPL, the cost of acquiring a hot lock, along with the overheads of obtaining exclusive access and IPC are plotted in figure 4. Figures 4a and 4b graph the same experiments, figure 4a plots contention up to 8 processes while 4b plots contention up to 32 processes.

The bottom dotted line plots the overheads associated with IPC in MPL (labeled MPL(IPC)), while the bottom solid line plots the overheads associated with obtaining exclusive access in SML (labeled SML(Exclusion). For these curves, the y axis indicates the average cost of performing a null critical section under the two approaches. For example, the overheads associated with obtaining exclusive access in SML were determined by measuring the time required to obtain and release a subpage lock in ME state. Similarly, the overheads associated with IPC in MPL were measured (using the FIFO queue mechanism) by sending a null message to the server and waiting for the acknowledgement. These two curves indicate that under any degree of contention, the inherent synchronization overheads for MPL are significantly greater than for SML.

The top dotted line and top solid line plot the cost of acquiring a hot lock for MPL and SML respectively. The cost of acquiring a hot lock was measured by having N transactions (each

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

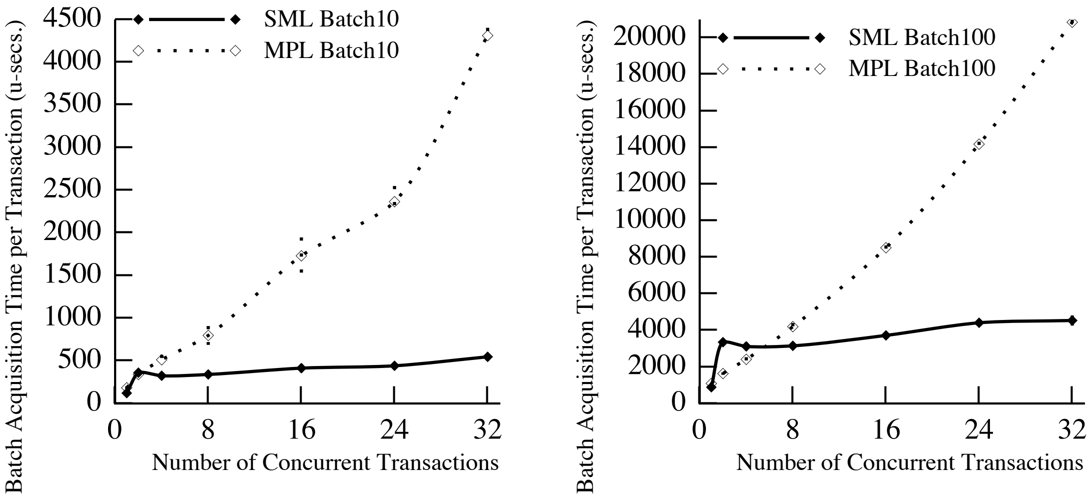


Figure 5: Acquiring 10 and 100 Batched Locks

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 SML to acquire a hotlock offers significant performance advantages over using MPL. The high overheads of IPC under contention in MPL are reflected in the total cost to acquire a lock for MPL. For MPL, the overheads of IPC dominate the costs of performing the lock acquisition. However, for SML, the overheads of obtaining exclusive access do not overwhelm the costs of performing the lock acquisition. The difference between the top and bottom lines for SML and MPL indicate the cost of actually executing the critical section, and are about the same for SML and MPL.

5.2.2 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 SML against a MPL implementation in which multiple lock requests were requested per message. For MPL, a batch of N corresponds to the client sending, in a single message, a request for N different locks. Since the notion of batch locks does not apply to SML, SML acquired the N different locks one at a time. Figure 5 compares the performance of SML and MPL for batches of 10 and 100 locks. For batches of 10 locks, the lock acquisition cost of SML and MPL are about equal for up to four concurrent processes, and for more than four concurrent processes, SML again outperforms MPL. For batches of 100 locks, MPL outperforms SML for concurrency levels of two and four, but once concurrency level reaches eight, SML outperforms MPL.

Note that the noticeable spike at two transactions in the SML curve is due to our usage of two rings. When two or more processors are used in our SML lock acquisition experiments, at least one reference will be to a remote ring. As mentioned in section 2.2, a remote ring reference is much more expensive (600 cycles) than a local ring reference (175 cycles). We speculate that, as the number of processors in the two ring experiment increases, local ring references are more likely to be satisfied before remote ring references, thus amortizing the subpage access cost in cases of very high concurrency.

5.3 Summary

In order to quantitatively compare SML and MPL, 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 overheads associated with obtaining exclusive access and IPC.

These experiments reveal the overheads inherent in using message passing vs. direct access to shared memory to implement lock operations. Consider N processes which simultaneously attempt to access a single critical section. The *average* cost, per process, will be at least:

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

Where CS is the execution time of the critical section, which corresponds to the cost of updating the lock space code for both SML and MPL. In addition to the cost of actually executing the critical section, the cost of a lock operation also includes overheads associated with IPC in MPL and obtaining exclusive access to the lock space in SML. It is not easy to accurately model these overheads analytically, since these are dependent on subtle interactions between the software and the hardware of the implementation platform. In an ideal situation, these overheads would be negligible. However, our experiments show that the overheads of IPC to a single server in MPL overwhelm the cost of executing the code which updates the lock space (the critical section). In contrast, the overheads associated with SML are negligible, and this translates into better overall performance for SML, especially under high contention.

Although our experiments were limited to measuring contention up to 32 nodes, the multiple ring architecture of the KSR-1 suggests that SML will scale up to hundreds of nodes. In our performance tests on a 2-ring KSR-1, processes were evenly divided between the two rings. For SML, 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, SML was shown to dominate MPL for a wide range of parameters.

For concurrent requests for *different* locks, under most request patterns, the performance results for both SML and MPL would improve. For non-hot locks under MPL with multiple lock servers, a good partitioning scheme would reduce the ill-effects of contention. For non-hot locks under SML, the ill-effects of contention should be negligible, since contention for page locks would be reduced, leaving only the overheads of bus contention.

6 Recovery Issues

In addition to performance issues, the suitability of a specific lock management strategy is also strongly influenced by recovery issues. Recovery issues are particularly important in multi-node database systems (including SD and SM database systems), where additional recovery issues beyond those found in uniprocessor database systems may surface. More specifically, if the crash of a single node (processor) of multi-node requires a total reboot because of the lack of support for detecting and isolating individual node failures, then multi-node database recovery is essentially the same as uniprocessor database recovery. However, these additional recovery issues arise in multi-node systems where *it is possible* to detect and isolate individual node failures.

In this context, it is possible to enforce more robust recovery strategies than a brute force approach of "aborting all active transactions in the entire system". To minimize lost work in the event of the crash of one or more nodes, we consider the implications of adopting the following multi-node recovery objective:

- Abort active transactions running on crashed nodes.
- Avoid aborting active transactions running on surviving nodes.

This recovery objective has direct implications on lock management. With respect to these active transactions, all locks held by aborting transactions should be released, while no locks held by non-aborting transactions should be released. We consider these multi-node lock management recovery issues for SML in section 6.1 and for MPL in section 6.2.

6.1 Recovery in SML

The recovery properties of any data structure stored in shared memory may be adversely affected by cache-coherency. To address this problem for SM database systems implemented on cache-coherent architectures, recovery protocols have been proposed [13]. These protocols ensure the multi-node recovery objective mentioned earlier in this section.

Consider the most popular and practical cache coherency protocol, the *write-invalidate* protocol. In a write-invalidate protocol, the last write to a cache line (subpage) invalidates all other copies of the cache line which may be stored in the caches of other nodes. In the case where a cache line stores entries which correspond to multiple lock acquisitions, it is possible that the only copy of an entry made by some node x will migrate to node y. For example, this may occur after independent transactions running on nodes x and y both acquire a compatible lock (i.e. a read lock) on one specific data item. This may lead to two problems related to the failure atomicity of active transactions: (due to the migration of lock acquisition information to other nodes (y)):

- *Lost Locks* - the crash of node y may destroy lock acquisitions made by node x.
- *Spurious Locks* - the crash of node x aborts transactions running on node x, but locks acquired by transactions running on node x remain cached on other nodes (y).

In such situations, certain runtime and restart recovery mechanisms can be employed to restore the failure atomicity of active transactions while adhering to the multi-node recovery objective. Moreover, these recovery mechanisms exhibit low overheads. For example, consider a low overhead runtime mechanism for restoring lost locks. Whenever a lock is acquired (or released), a log record written to volatile memory. This will ensure that sufficient information can be maintained on surviving nodes, enabling the restart recovery mechanism to restore any locks lost due to the crash of another node.

Consider the removal of spurious lock acquisitions during restart recovery. Recall our previous discussion of lock acquisition in section 4.1, where we noted that a lock acquisition LCB entry includes the transaction ID, and assume that this transaction ID also encodes the ID of the node where this transaction was executing. After each surviving nodes determines which node(s) has crashed, the surviving node's restart recovery procedure merely needs to examine all LCB's on its node for any ID's of crashed nodes, and release these locks. Further details of these SM recovery mechanisms may be found in [13].

6.2 Recovery in MPL

Recovery strategies originally designed for SD systems can be employed for MPL. For example, consider recovery strategies discussed in the SD system proposed in [11]. In [11], locking is managed by a single server, called the GLM (Global Lock Manager) and a per node LLM (local lock manager). Initially, a lock is obtained from the GLM, however, since the requesting node caches lock acquisitions, subsequent requests made by the same node may be satisfied locally. Thus, the transaction system first attempts to obtain locks from the LLM. If the lock is currently held by the LLM, no IPC will be necessary. 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 restart recovery, no surviving node will be granted any of l's locks by the backup GLM until all nodes recover completely.

This discussion has shown that the runtime overheads for SML and MPL are very similar, and that restart recovery can be performed fairly easily and cheaply in both lock management strategies. For example, the runtime overheads (in terms of space and time), associated with normal runtime operation of the volatile logging strategies in SML [13] are comparable to the caching strategies implemented in the SD variant of MPL in [11]. Also, for both SML and MPL, ensuring that spurious locks can be identified can most effectively be done with the same runtime mechanism – storing the node ID in the lock table. Given these similar recovery mechanisms and overheads, choice of SM lock management architectures should be determined based on performance criterion.

7 Related Work

A number of studies have been conducted on the performance of database lock management, to name a few see [3, 24, 25]. These studies are based on analytical and simulation models, and consider uniprocessor and SD database systems. In contrast, our study has considered implementation and performance issues in SM database systems, and has focused on empirical measurements.

In [23], the performance of on-line transaction processing applications are examined on a Sequent shared memory multiprocessor. This work examined the performance effects of the interaction between many system components, including the caching subsystem, process migration, and disk I/O. However, in this paper, high contention database locking was only mentioned as one possible source of system performance degradation.

Other recent work on SM database implementations have considered parallelism in query processing [5, 10]. However, these studies have not addressed the performance of SM database lock management. Other general performance studies of the KSR-1 multiprocessor are reported in [21, 20], but these studies do not assess database related activities.

In [16] a technique for augmenting an SD system with shared memory is suggested. Instead of using a GLM per object, a *global lock table* is 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, no quantitative performance results are reported. In contrast, we provide quantitative performance measurements for MPL.

In order to asses the performance of lock manager implementations in SM database systems, our performance studies have focused exclusively on the lock management component. However, other components of a database management system may also profoundly influence performance – especially the commit process of systems which require disk I/O to implement the WAL protocol. Technological trends, such as non-volatile RAM [1] suggest that costly disk I/O can be avoided. For such systems, the performance of lock management will have a larger impact on the performance of the entire database management system and our design and implementation strategies will prove to be very useful.

Some researchers have argued that SM database systems are not well suited for high performance database implementations, arguing that the platforms that these database systems are implemented on are not scalable [2]. 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.

8 Conclusion

On a large-scale multiprocessor, it is particularly important to optimize the performance of lock management, since many transactions, running on different nodes, may concurrently invoke database locking operations. For this reason, this paper considered the design, implementation and performance of two basic locking strategies on a large-scale cache-coherent shared memory multiprocessor, SML and MPL. In SML, database locks are acquired by updating shared data structures stored in coherent shared memory, while in MPL, communication with a server is typically required.

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

Our implementations of SML and MPL exploit a novel synchronization feature of the KSR-1, one which enables processing nodes to obtain and hold exclusive locks on subpages. This feature enables a very efficient implementation of exclusive access to the lock space in SML and enables an also enables an efficient implementation IPC in MPL. For the KSR-1 implementation platform, our performance evaluation of SML show that the impact of high data sharing contention on shared system resources is marginal.

Although it would be possible to implement these locking methods with other multiprocessor synchronization primitives (such as semaphores) such approaches are not likely to be as efficient. No other multiprocessor synchronization primitive both caches a data segment *and* guarantees exclusive access to a data segment. The ease of use and the efficiency of this synchronization primitive suggests that other multiprocessor vendors should adopt the KSR-1's approach to synchronization.

Finally, we considered the implications crash recovery and its relation to database lock management architectures. Given sufficient low-level failure detection and isolation mechanisms, for both SML and MPL, it is feasible to avoid aborting active transactions running on surviving nodes. Moreover, in both SML and MPL, this can be supported with low runtime overheads. Given the similarity of this recovery support, based on its superior performance, it is clear that the SML should be the SM lock management architecture of choice.

References

[1] G. Copeland, T. Keller, R. Krishnamurthy, and M. Smith. The Case for Safe RAM. *Proceedings of the 15th International Conference on Very Large Data Bases*, pages 327–335, 1989.

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

[3] P. Franaszek, J. Robinson, and A. Thomasian. Concurrency Control for High Contention Environments. *ACM Transactions on Database Systems*, pages 304–345, June 1992.

[4] A. Gottliieb, B. Lubachevsky, and L. Rudolph. Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors. *ACM Transactions on Programming Languages and Systems*, 5(2):164–189, April 1983.

[5] G. Graefe. Volcano – an extensible and parallel query evaluation system. *IEEE Transactions on Knowledge and Data Engineering*, 6(1):120–135, February 1994.

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

[7] 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.

[8] M. Herlihy and J. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[9] D. Lilja. Cache Coherence in Large-Scale Shared-Memory Multiprocessors: Issues and Comparisons. *ACM Computing Surveys*, 25(3):303–338, September 1993.

[10] T. Martin, P. Larson, and V. Deshpande. Parallel Hash-Based Join Algorithms for a Shared-Everthing Environment. *IEEE Transactions on Knowledge and Data Engineering*, 6(5):750–763, October 1994.

[11] 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. *Proceedings of the 17th International Conference on Very Large Data Bases*, 17:193–207, 1991.

[12] 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.

[13] L. Molesky and K. Ramamritham. Recovery Protocols for Shared Memory Database Systems. *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 11–22, May 1995.

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

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

[16] 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.

[17] T. Rengarajan, P. Spiro, and W. Wright. High Availability Mechanisms of VAX DBMS Software. *Digital Technical Journal*, (8):88–98, February 1989.

[18] Kendall Square Research. *KSR1 Principles of Operation.* KSR Research, Waltham, Mass., 1992.

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

[20] R. Saavendra, R. Gaines, and M. Carlton. Micro Benchmark Analysis of the KSR1. *Proceedings of Supercomputing '93*, pages 202–213, November 1993.

[21] J. Singh, T. Joe, A. Gupta, and J. Hennessy. An Empirical Comparison of the Kendall Square Research KSR-1 and Stanford DASH Multiprocessors. *Proceedings of Supercomputing '93*, pages 214–225, November 1993.

[22] W. Snaman and D. Thiel. The VAX/VMS Distributed Lock Manager. *Digital Technical Journal*, (5):29–44, September 1987.

[23] S. Thakkar and M. Sweiger. Performance of an OLTP Application on Symmetry Multiprocessor System. *Proceedings of the 17th International Symposium on Computer Architecture*, pages 228–238, May 1990.

[24] A. Thomasian. Two-Phase Locking and its Trashing Behavior. *ACM Transactions on Database Systems*, pages 579–625, December 1993.

[25] A. Thomasian. On a More Realistic Lock Contention Model and its Analysis. *Proceedings of the 10'th International Conference on Data Engineering*, pages 2–9, 1994.