Adaptive Locking Strategies in a Multi-node Data Sharing Environment

Ashok M. Joshi
Digital Equipment Corporation
Database Systems Engineering group
Nashua, New Hampshire 03062

Abstract

This paper describes some of the concurrency control algorithms used in Rdb/VMS. Rdb/VMS[1] uses the facilities provided by the VMS lock manager in order to perform locking among concurrent processes. The locking algorithms adapt to the contention among concurrent users by adjusting the number of locks required as well as the number of lock requests that are required per transaction. This makes it possible to reduce the locking overhead considerably without sacrificing correctness or concurrency in a data sharing environment. These algorithms indicate that it is possible to achieve substantial improvements in certain situations.

1. Introduction

Locking is perhaps the most widely used concurrency control technique today. Locking ensures that concurrent transactions are able to perform updates to the database correctly and consistently. There is a wealth of literature on the subject of locking and concurrency control [Berns87, Carey84, Eswar76, Gray78, Lehma81].

Locking is a pessimistic method of concurrency control; a transaction has to acquire several locks on data objects during its execution including record locks, table locks, page locks and file locks. The number of locks acquired and released by a transaction is dependent on the number of objects accessed (and/or updated) by the transaction as well as other factors such as the locking granularity used by the DBMS.

In a typical database management system, the number of locks requested by the transaction is independent of the actual number of conflicts that occur among concurrent transactions. Hence, even if the concurrent transactions access non-intersecting sets of data, they will continue to incur the same overhead of locking for each transaction. The concurrency control sub-system of the DBMS is unable to detect the degree of conflict and adjust the number of locks requested by the transaction, without sacrificing correctness and the degree of concurrency in the system. Locking can impose as much as 10% overhead for interactive transactions [Date83]. In a data sharing system that uses distributed lock management services, the locking overhead may increase substantially due to inter-node messages. It is desirable to minimize this overhead in a high performance database management system.

Rdb/VMS is a high performance relational database management system that uses locking algorithms that are sensitive to the contention in the system. The algorithms use lock de-escalation [Lehma86, Lehma89] in order to reduce the number of locks required per transaction under low contention situations. The algorithms, which first appeared in Digital's database products in 1984, are also able to detect situations of low contention and automatically reduce the number of lock requests per transaction. If contention increases, the number of locks automatically increases to the appropriate level required to maintain the correctness and consistency of the data. In this sense, the algorithms are adaptive to the contention in the system. This adaptive nature can lead to a significant reduction in the number of lock requests made by a transaction. This is particularly significant in a distributed environment, where it is critical to reduce the number of inter-node lock request messages per transaction. Rdb/VMS has achieved significant performance gains using these techniques.

[1] The following are trademarks of Digital Equipment Corporation: Rdb/VMS, VAX, VAXClusters, VMS

We start with a description of the relevant features of the VMS distributed lock manager. This is followed by a description of the use of these features in Rdb/VMS. We describe some of the record locking algorithms that use lock de-escalation techniques in order to reduce the total number of locks requested, followed by a description of the adaptive strategies that take the actual contention into account in order to reduce lock traffic. We indicate some performance improvements and conclude with a discussion and comparisons of related work on lock de-escalation.

2. The VAXCluster and VMS Distributed Lock Manager (DLM)

Rdb/VMS is a full-function relational database management system developed by Digital Equipment Corporation that provides access to the database in a VAXcluster. A VAXcluster system is a "closely coupled" structure of standard VAX computers (nodes) that has characteristics of both loosely coupled and tightly coupled systems. The system has separate processors and memories that are connected by a message-oriented interconnect, running the VAX/VMS operating system. However, the concept of VAXclusters relies on close physical proximity, a single physical and logical security domain, shared physical access to disk storage and high-speed memory-to-memory block transfers between nodes. The VAXcluster system provides a high availability, easily extendible system to customers. This model of data sharing has been referred to as the shared-disk paradigm in the the literature [Bhide87]. Refer to [Krone87] for more details on the VAXcluster concept.

The VMS distributed lock manager (DLM) is the foundation of all resource sharing in clustered as well as single node VMS systems. It provides services for naming, locking and unlocking cluster-wide resources and clusterwide synchronization. Since the DLM is heavily used, it is designed to minimize the number of messages that must be exchanged between nodes in order to manage locks. Secondly, the DLM is able to recover from failures of nodes holding locks so that surviving nodes can continue to access shared data in a consistent manner. [Renga89] discusses some of the high availability mechanisms in Digital's database products that utilize the services of the DLM. Rdb/VMS utilizes the services of the DLM in order to provide concurrency control, mutual exclusion and certain types of event notification. The decision to use the DLM was based primarily on its ability to support VAXclusters as well as its robustness and flexibility. This decision has strongly influenced the architecture of the database management system.

The DLM allows cooperating processes to define shared resources and synchronize access to these resources. The DLM does not recognize the concept of a transaction holding or requesting locks; lock conflicts are resolved on an inter-process basis. This feature of the lock manager has resulted in a process-per-user architecture for the database management system.

Resource names are user-defined. In addition, the DLM allows users to define a resource hierarchy like database, table, records etc. The DLM provides a variety of lock modes. These lock modes are a superset of the intention lock modes described in [Date83].

The DLM provides for synchronous as well as asynchronous completion of lock requests. This allows the caller to post a request for a lock and continue processing without having to wait for the request to complete. This mechanism is used for event notification by Rdb/VMS. This is described in more detail later.

2.1 Blocking ASTs

A very useful feature of the DLM is the notion of blocking asynchronous system traps (ASTs). ASTs are software traps that are generated by the operating system to notify processes of the occurrence of asynchronous events. Whenever a lock request is made for a resource, the caller can optionally specify the address of a routine (blocking AST routine) that is to be invoked whenever there is a conflicting request for this lock. When another process requests a conflicting lock on the same resource anywhere in the cluster, the DLM notifies the process holding the lock by transferring control to the blocking AST routine associated with the lock. The blocking AST routine that is thus invoked has the option of either giving up the lock immediately, ignoring the conflict, or deferring handling of this request till some later time. The DLM guarantees that the blocking ASTs will be delivered reliably to all the processes that have blocking ASTs enabled.

If a lock request cannot be granted immediately, the process requesting the lock may (optionally) choose not to enqueue the request (bounce locks). This fea-

ture is useful if a process wishes to determine whether there is contention for a resource before attempting to acquire a lock. This is most often used in situations where users do not wish to wait for a high contention resource, but may prefer to try the request later. Refer to [Snama87] for additional details of the internals of the VMS distributed lock manager.

3. Use of the VMS Distributed Lock Manager by Rdb/VMS

The decision to provide transparent access to the database in a VAXcluster environment influenced the design of the concurrency control algorithms. In particular, it was decided to use the DLM for almost all types of synchronization, including page locks, table locks and record locks. In addition, we decided to support a shared data model (instead of a partitioned data model), due to the lack of fast inter-node communication primitives. Figure 1 shows the architecture of Rdb/VMS running in a two node VAXcluster. The DLM provided several advantages since conflicts can be detected and resolved on a cluster-wide basis. This permits access to the data from any node in a VAXcluster, regardless of how many other nodes are accessing the database. However, there are high costs associated with performing cluster-wide locking. As a crude approximation, inter-node lock requests are about eight to ten times more expensive (in terms of the number of instructions) than an intra-node lock request. This has several implications. Firstly, the database system has to minimize the use of locks as much as possible. This is achieved in Rdb/VMS by using lock de-escalation techniques, similar to those proposed in [Lehma86], [Lehma89], for page locks and record locks.

Secondly, it is desirable to reduce the inter-node lock

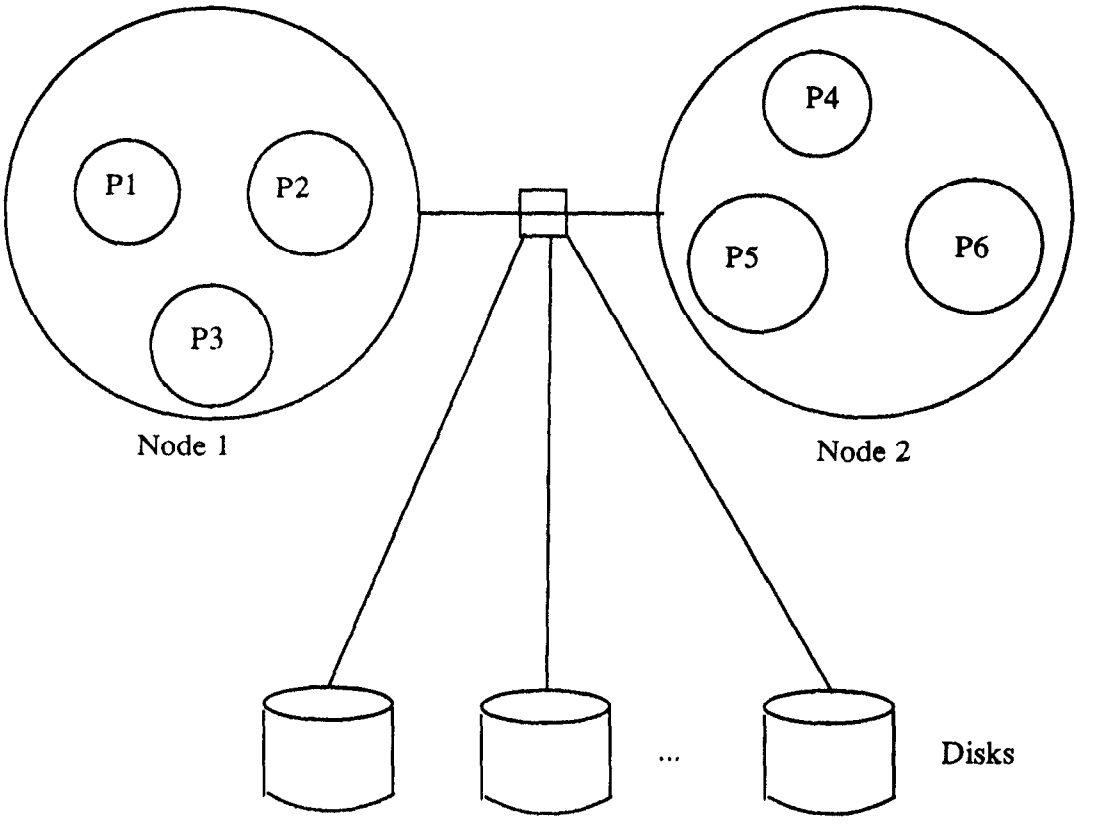


Figure 1: Process architecture in a two node VAXcluster environment.
Processes P1, P2, P3 are accessing the database from node 1
Processes P4, P5, P6 are accessing the same database from node 2

traffic. This can be done by ensuring that data are partitioned on a per-node basis, in order to minimize inter-node contention. Since Rdb/VMS supports a shared data model, this has to be done at the application level by carefully partitioning the data sets that the programs access. Note that this partitioning can be dynamic and completely at the discretion of the application developer. In order to improve performance in such situations, we have developed the adaptive algorithms described below.

The algorithms are intended to exploit the partitioned nature of application programs. In other words, the enhancements most benefit those applications that work against disjoint partitions of the database and have little sharing. In addition, the algorithms do not penalize applications that do not use these enhancements. In other words, applications that do not exhibit a high degree of data partitioning will run at about the same (or slightly improved) performance levels as without the optimization.

3.1 Rdb/VMS Lock De-Escalation

In order to explain the concept of lock de-escalation, it is necessary to distinguish between strong and weak lock modes. A strong read (write) lock on an object only allows readers (a single writer) to access (write) the object. Weak locks are similar to intention locks; however, an important distinction is that intention locks are typically held in a two-phase manner, whereas we permit strong locks to be demoted to weak locks within a transaction. When a set of objects can be organized into a tree-structured granularity hierarchy, the weak locks are usually held at the coarse granularity level to indicate read or write intentions of the transaction.

The de-escalation algorithm works as follows. Acquire a strong lock at the root of the resource hierarchy. This lock dominates all the objects in the hierarchy and hence, all the descendants of the root are implicitly locked. No further explicit locking is required. It is also necessary to remember the leaf level entities that are accessed, should it become necessary to perform de-escalation at a later time.

When there is conflict at the root of the hierarchy, the transaction acquires a strong lock on the appropriate nodes at the next level in the hierarchy and demotes the parent lock to a weak lock. This pairwise acquiring/demotion of locks continues towards the leaves of the tree as long as there is contention or we reach the leaf level, at which point, no further refinement is possible.

Lock de-escalation is most useful when an application accesses several objects that belong to the same parent in the resource hierarchy because a small number of locks is sufficient to implicitly lock a large number of objects. We now describe how this technique is used in Rdb/VMS, beginning with a description of the lock de-escalation feature for page locks.

3.2 Buffer Locking

An Rdb/VMS database is made up of a number of files containing database pages. Each page is an integral number of 512 byte blocks; the actual size of a page is DBA-defined and can vary from one database file to the other. Files are allocated as contiguous groups of blocks. Contiguous pages are grouped into sets; a set is the unit of data transfer when reading from the disk into database buffers. The number of pages per set is determined at database definition time based on the size of buffers in main memory. For example, if a buffer is defined to be 12 disk blocks and each page is 4 disk blocks, then each set contains three pages. It is possible to determine the set boundaries by knowing the page size and buffer size. In this example, the set boundaries are page 1, page 4, page 7, ... as shown in Figure 2.

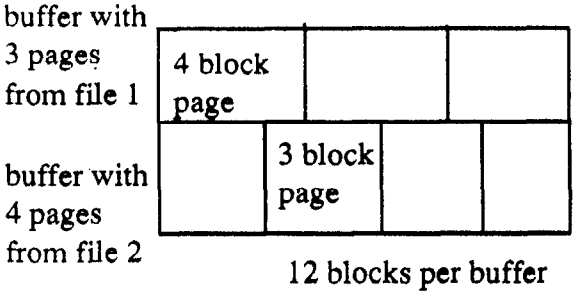


Figure 2: Buffer organization showing 4 block pages and 3 block pages

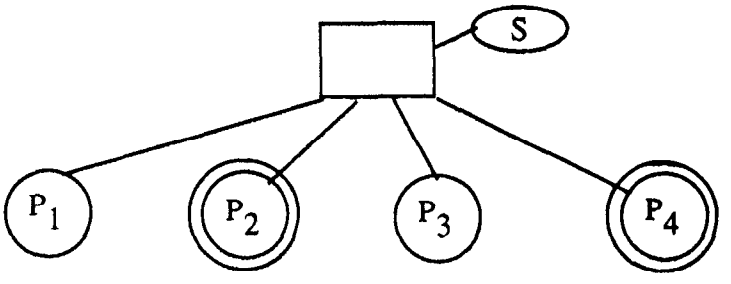
As mentioned before, a read IO is performed on a set of pages. By defining a large buffer size (and consequently a large set), the buffer manager provides some read-ahead capability, by fetching the requested page as well as the neighboring pages that belong to

the set. This has influenced some of the algorithms for storing related records on near-by pages to achieve clustering.

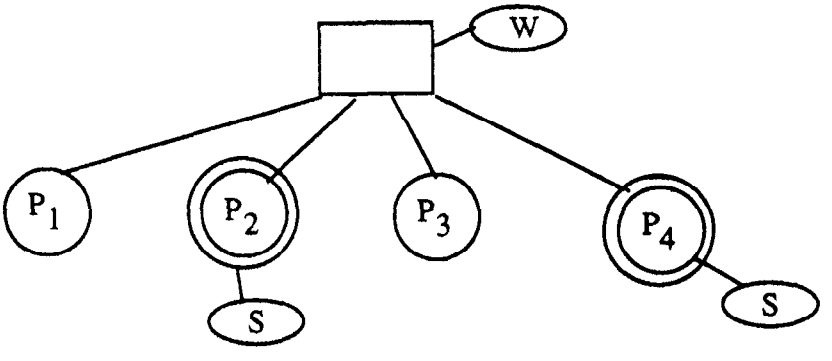
Whenever a page needs to be fetched from disk, the buffer manager determines the set it belongs to. It then acquires a strong lock on the set and reads the set of pages into an available buffer. Regardless, of the mode (read or write) of the requested page, the lock manager always requests a strong buffer lock on the set (exclusive mode). The strong buffer lock dominates all the pages in the set; in other words, the strong lock on the set implies a lock on all the pages in the set. Associated with this lock is a blocking AST routine that is set up to handle conflicting requests for any page in this set of pages. Figure 3 shows an example of the de-escalation technique applied to page locks.

If the buffer manager succeeds in getting this lock, then this single lock will be sufficient to lock all the pages in the set. This can lead to significant savings in the number of lock requests if there is no contention for these pages and more than one page in the set is accessed. Note that page locks are treated as short-duration latches (semaphores or non-two phase locks).

As each page is accessed, the buffer manager records



Case a: Strong buffer lock indicates implicit locks on pages P1, ... P4.
Pages P2 and P4 have been accessed by the transaction.



Case b: Buffer lock weakened because of conflicting request for a page in the same set. Strong locks acquired on pages P2 and P4.

Page Pn has been accessed/remembered by the transaction

Figure 3: Lock de-escalation for page locks showing the state before and after conflict

the activity for the page (whether read or write) in the buffer. This information is used when there is a conflicting request for the buffer lock by another transaction. The conflicting request for the buffer lock invokes the blocking AST routine for the process holding the strong buffer lock. The blocking AST routine first acquires appropriate locks (read or write) on the pages that have been remembered. Then it downgrades the buffer lock to a weak mode. The other transaction's request for an exclusive lock fails, and it is forced to acquire a weak lock on the buffer and a strong lock on the page that it needs to access. This request is granted if it is compatible with the existing lock on the page, otherwise the request is forced to block. This mechanism has been useful in reducing the number of page locks (latches) that are necessary.

Note that the second transaction needs to get a weak buffer lock in order to prevent future requests for an exclusive buffer lock (from other users) from being satisfied.

3.3 Record Locking

Rdb/VMS supports two-phase locking at the record level using lock de-escalation in order to reduce the number of locks that may be required for accessing disjoint sets of records. Records within a table are grouped into a tree structure called the *adjustable lock granularity tree* (ALG tree). This tree organizes the records into varying levels of granularity starting with the root of the tree being the entire table and the leaves being individual records. The number of levels in the tree as well as the successive refinements of the granularity at each intermediate level can be defined by the DBA. In the simplest case, it is possible to have a two-level tree where the root represents records in the entire table and the leaf level the individual records. Figure 4 indicates the resource hierarchy for record locks. The term *logical area lock* is explained below.

The lock de-escalation protocol for record locks is similar to the page locking described above. Whenever a record lock is requested, Rdb/VMS attempts to acquire a strong lock on the highest ancestor of the record (in the ALG tree). If it succeeds in obtaining the strong lock, all descendants of that node are implicitly locked. When individual records are accessed, it is necessary to remember each record that has been accessed so that it is possible to later de-escalate the high level lock to the leaf level if necessary. The cost of remembering implicitly locked records is proportional to the number of records accessed.

In the low contention case, it is possible to have one strong lock on an ancestor that implicitly locks its descendants. If the amount of conflict increases, it is possible to perform de-escalation and acquire explicit record locks. The blocking ASTs permit de-escalation to occur while the transaction is in progress. Figure 5 shows an example of the de-escalation technique applied to record locks.

4.0 Lock Caching: Area and File Locks

Rdb/VMS uses the term *logical areas* to refer to tables (or horizontal partitions of tables). Whenever a transaction needs to access a logical area, it is necessary to acquire an intention lock [Date83] on that area in the requested mode. At the end of the transaction, the logical area lock (intention lock) is demoted to allow other users to access the area. In other words, logical area locks are managed in a two-phase manner.

In addition, Rdb/VMS provides a wide variety of on-line operations that allow a DBA to perform physical restructuring of the database (e.g. move a database file from one device to another). In order to support this on-line restructuring, it is necessary for transactions to maintain locks on physical files as well. These locks are acquired at the start of the transaction

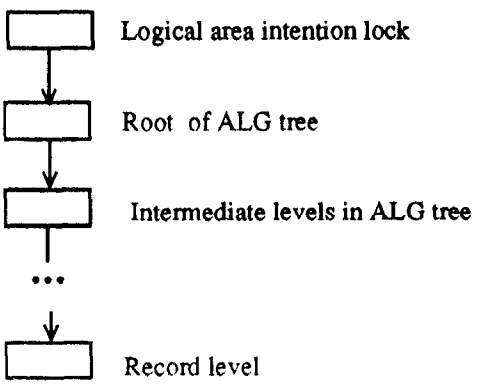
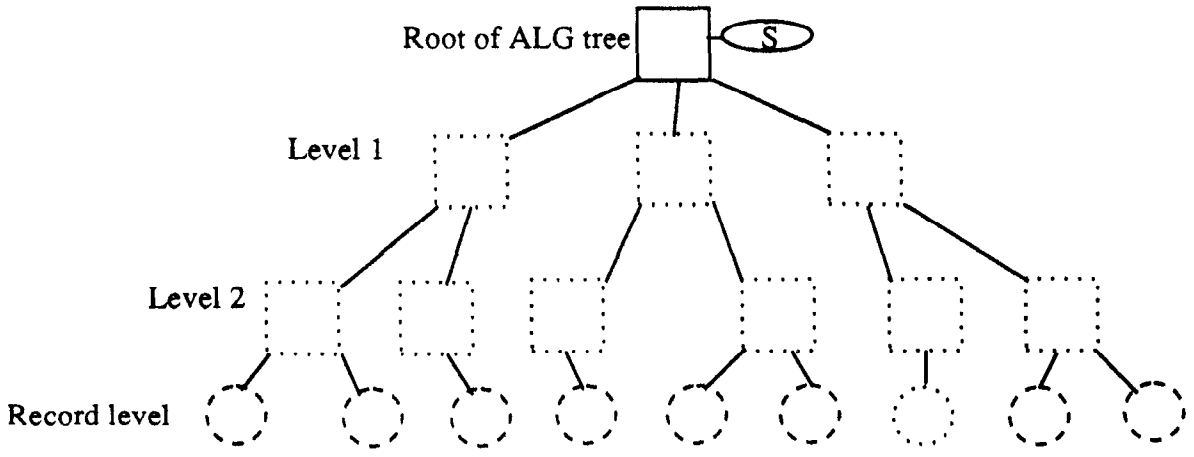
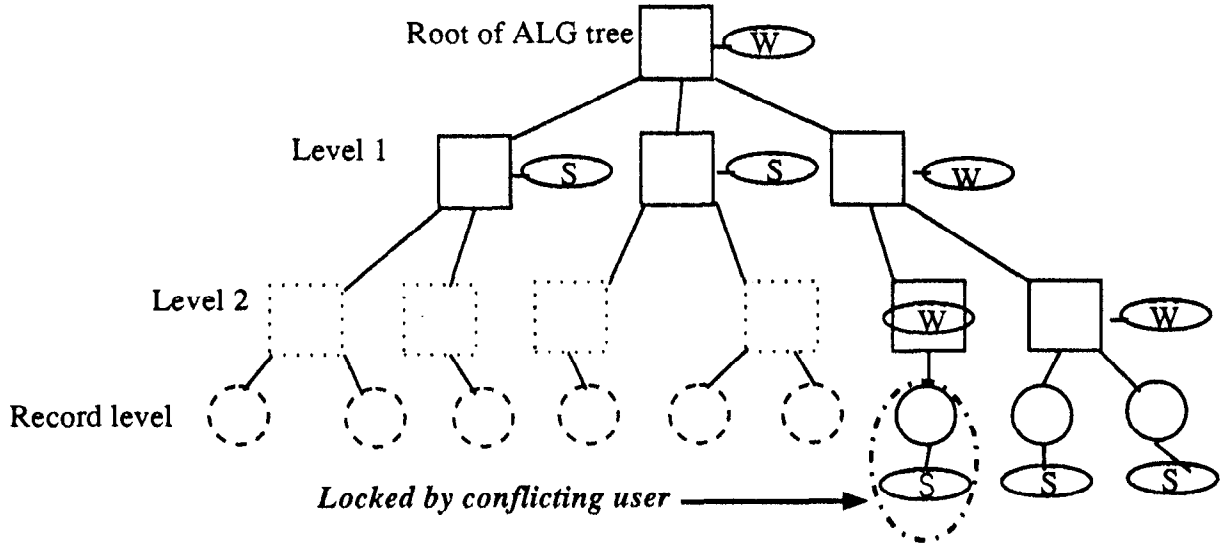


Figure 4: Resource hierarchy for record locks



Case a. No other users interested in records in this table. Transaction only holds strong root level lock and implicit locks on all the records in the table.



Case b: Conflicting request for a record lock from another user results in the root lock being weakened and acquiring strong locks at intermediate levels. The right side of the resource tree has been weakened all the way down to the record level.

W indicates a weak lock on this object

S indicates a strong lock on this object

..... Implicitly locked objects

- - - - - Accessed and remembered objects

-·-·-·- Object locked by conflicting user

Figure 5: Lock de-escalation for record locks showing resource tree before and after conflicting request.

and held until end-of-transaction. In an unoptimized scenario, the logical area locks and file locks can add considerably to the overhead of locking.

4.1 Adapting to Low-Conflict Situations

Assume that application programs are designed to access disjoint partitions of the database. Note that multiple copies of the program may be run from multiple computer nodes in the VAXcluster; each invocation will most often access its own set of data, and occasionally access data in another partition. An important point to note is that the partitioning need not be statically defined; rather, it may change dynamically, depending on the number of invocations of the application program, number of computer nodes available to access the database etc. This kind of disjoint partition access is not uncommon in real-life applications.

In such a scenario, the same logical area lock and file lock is requested and released by every transaction. In addition, the mode of the requested lock is usually the same as in the prior transaction. Hence, it makes sense to avoid the unnecessary overhead of lock request (at begin transaction) and demotion (at end-of-transaction) for each transaction: rather, the process can simply hold the logical area locks and file locks (since they will be immediately requested in the subsequent transaction). This optimization is based on the fact that locks are process-owned in Rdb/VMS. Hence, locks can simply be carried over from one transaction to the next. Note that in systems where locks are held by transactions, it should be possible to modify the lock manager to incorporate this concept of lock carry-over.

The term *carry-over* locks is used to denote those locks that are capable of being transferred within a process from one transaction to the next. If the transactions in a process access several different logical areas and files, it may happen that the process acquires a large number of carry-over locks over a period of time. Furthermore, any given transaction will only need some subset of those logical areas and files. Hence, it is useful to distinguish carry-over locks as those being used in the current transaction from those that are not. This is done as follows. Every process maintains data structures that contain information about the locks that the process owns. Each carry-over lock has an IN-USE flag associated with it. Before using a carry-over lock, a transaction can set the IN-USE flag indicating that this lock is being used in the current transaction. At end-of-transaction, it is necessary to clear the IN-USE flag for every logical area and file lock.

Carry-over locks have a blocking AST routine associated with them. Whenever there is a conflicting request for such a lock, the blocking AST routine is invoked. It works as follows. If the lock is marked as in-use, then it is not possible to give it up at this time. The blocking AST routine sets another flag indicating that this lock should be given up at end-of-transaction. The conflicting transaction has to wait. However, if the lock is not in use, it can be released, thus allowing the conflicting transaction to continue.

4.2 Carry-over optimization for record locks

We applied a similar optimization to record locks. Note that there are some significant differences between logical area locks and record locks with respect to the notion of carry-over. First, there is almost no temporal locality of reference for record locks. Secondly, the number of records is typically so large that it is not economical to carry-over record locks.

The carry-over optimization is applied to the root of the ALG tree for records within a table. It works as follows. When a transaction is ready to commit, it checks the root-level lock for each ALG tree. If that lock is a weak lock, the transaction gives up all the locks in that ALG tree. If however, the root-level lock is a strong lock, it does not demote the lock, but carries it over to the next transaction. Note that the strong root-level lock means that there are only implicit locks on the internal and leaf (record level) nodes of the ALG tree.

The reason for only carrying over strong locks is that record locking is based on lock de-escalation. If we also carried over weak root-level locks, that would disable the benefits of the ALG de-escalation algorithms, because it would prevent any transaction from ever acquiring a strong root-level lock. Hence, the algorithms use the heuristic that if the root level lock is weak, it is best to give it up completely. Note that the conflicting transactions will also release all the (weak) root-level ALG locks at transaction end. This makes it possible for subsequent transactions to try for a strong ALG root-level lock.

During periods of high contention for record locks, the blocking ASTs (associated with the locks in the internal nodes of the ALG tree) will ensure that the

locks are demoted to the appropriate compatibility mode on demand. Hence, it is not necessary to associate any additional information with the locks in the ALG tree. If there is a conflicting request for a carried over lock, the lock will be weakened by the blocking AST routine, and at commit time, will be released completely.

The carry-over optimization can yield substantial reductions in the number of lock requests per transaction. However, this optimization can lead to starvation of certain types of transactions. A notification protocol was designed to handle this case.

4.3 NOWAIT transactions

Rdb/VMS supports the notion of NOWAIT transactions. A user can optionally specify a NOWAIT clause on the SET TRANSACTION statement to indicate that if the transaction cannot immediately acquire locks on the resources that it requests, it should return to the user with a lock conflict message. The user can retry the request later. This feature is most useful in interactive applications.

If the locks that a NOWAIT transaction needs are being held by other processes as carry-over locks, it can lead to starvation of the NOWAIT transaction. In order to handle this case, the presence of a NOWAIT transaction must notify all the users in the database system that they cannot hold area and record locks across transactions.

This problem was solved by using the concept of blocking ASTs and two of the lock modes, concurrent write (CW) and protected read (PR), provided by the DLM. The interesting point about these lock modes is that CW is compatible with CW, PR is compatible with PR, but PR is not compatible with CW.

A NOWAIT transaction broadcasts its presence to all the users in the system. On receiving this broadcast, every user gives up all carry-over locks (in order to be fair to the NOWAIT transaction), and continues as usual.

During commit processing, every user determines whether it is allowed to perform the carry over optimization. The presence of a NOWAIT transaction in the database indicates that the optimization is not possible; otherwise, it is safe to carry over locks to the next transaction.

4.3.1 Implementation Details

Every NOWAIT transaction requests a lock (called the NOTIFICATION lock) in CW mode at transaction start. The purpose of this lock is to broadcast the presence of a NOWAIT transaction in the system. Once the lock is granted to the NOWAIT transaction, it is assured that every other user is aware of its presence and has given up all carry-over locks.

At end-of-transaction, every transaction tries to acquire the NOTIFICATION lock in PR mode (note that PR and CW are incompatible). If the PR request is granted, then there are no NOWAIT transactions in the system; hence, it is safe to perform lock carry-over. If the PR request for the NOTIFICATION lock is not granted (because another user has it in CW mode), then it is necessary to give up all the locks at end-of-transaction. In short, whether to carry over locks or not is determined by whether the NOTIFICATION lock can be acquired in PR mode or not.

The NOTIFICATION lock in PR mode has a blocking AST associated with it. If another user requests the lock in CW mode, then the blocking AST is invoked; the blocking AST demotes all the carry over locks belonging to this process and and then releases the NOTIFICATION lock. Thus, the NOWAIT transaction is able to broadcast its presence to other processes in the system.

5. Performance Improvements Due To The Carry-over Optimization

The enhancements discussed above do not penalize applications that do not access disjoint partitions of data. This is because conflicting requests will result in forcing processes to give up locks at the end of the transaction which is exactly the behavior without the optimization.

We now present some performance data based on the Debit/Credit benchmark [Anon85] that indicates the performance benefits that were achieved using the carry-over optimization for intention locks, file locks and top-level ALG locks. These numbers are only preliminary and suggestive of the kinds of performance improvements that may be obtained. Due to large disk and processor requirements of the benchmark, we are unable to report performance numbers for large numbers of nodes.
In terms of the number of lock requests, the optimization was responsible for reducing more than half the

lock/unlock requests per Debit/Credit transaction. The performance numbers were obtained by running the Debit/Credit benchmark with and without the optimization for each of the configurations in a single node, two-node and three-node VAXcluster. Since we were interested in measuring the relative performance gains due to the carry-over optimization, the TPS numbers reported here may not exactly match the official reported numbers for the Debit/Credit benchmark.

The numbers clearly indicate the performance benefits of using these optimizations in a multi-node environment. Further studies are necessary to understand performance improvements in a VAXcluster of several nodes. The relative performance gain for the two-node case is significantly higher than the one-node case. This is due to the fact that multi-node tests generate inter-node lock manager messages. Hence, a reduction in the number of messages significantly impacts the performance.

Table 1: Rdb/VMS Version 4.0 performance improvement with lock optimization (TPS and percent)

	Number of VAXcluster nodes		
	One	Two	Three
Rdb/VMS 4.0 TPS with optimization	31.8	48.6	65
Estimated percentage improvement due to optimization	14%	61%	67%

6.0 Related Work

The lock de-escalation algorithms described above have been in Digital's database products since 1984. The work reported in [Lehma86], [Lehma89] is most closely related to these techniques; however, there are some differences.

[Lehma89] uses a two level resource hierarchy; the relation level and the tuple level. Each transaction starts with a relation level lock. In addition, it is necessary to keep track of the number of transactions that are waiting on the relation-level lock, in order to trigger de-escalation. Once the count of waiting transactions exceeds a certain threshold, de-escalation is performed and explicit tuple level locks are acquired, based on the transaction's tuple-level write set and read predicates. This approach has the disadvantage of blocking transactions until de-escalation is performed. Note that these transactions will have to wait even if they would have accessed disjoint sets of records.

Our algorithms do not have this "reduced concurrency" problem since we permit a multi-level hierarchy and de-escalate immediately on detecting a conflict at the coarse granularity level.

7. Conclusions

The DLM is a highly optimized, flexible lock manager that has provided the foundation for the locking algorithms in Digital's database management systems. Its fault-tolerant and distributed characteristics are invaluable in building a database system based on the shared data model.

We have described the lock de-escalation algorithms that are used for buffer locking and record locking. Lock de-escalation is extremely useful when the transaction accesses several objects that belong to the same parent in the resource hierarchy.

Finally, we have described some of the concurrency control techniques that have been implemented in Rdb/VMS. The locking mechanisms are unique in the manner they adapt to varying degrees of conflicts among concurrent transactions. This adaptability to conflicts can result in significant reduction in the number of lock requests per transaction, which in turn, results in significant performance gains.

The reduction in lock requests is extremely signifi-

cant in a multi-node shared data environment where it is critical to minimize the number of inter-node messages while permitting access to data from any node. Our experiments indicate that the adaptive algorithms are beneficial in a multi-computer environment.

It is important to point out that the locking techniques mentioned here do not interfere with commit processing and availability in a VAX cluster. If one of the nodes in a VAXcluster fails while the database is being accessed from that node, the recovery manager is able to distinguish between carry-over locks and in-use locks. This permits recovery to proceed even when the failed process holds carry-over locks.

Finally, in the absence of a mechanism like blocking ASTs, the algorithms have to be modified to handle transaction aborts due to deadlocks. This can be done by releasing all carry-over locks at end-of-transaction, if the abort was caused due to deadlocks.

7.1 Acknowledgments

The performance numbers could not have been reported without the help of Rabah Mediouni, who spent countless hours tuning the database and running the tests. Jay Banerjee helped in acquiring the data for Table 1.

Steve Klein developed the lock carryover concept. Dave Lomet, Peter Spiro, T.K. Rengarajan, Ananth Raghavan and others have contributed to the development of these ideas at various stages.

Finally, we would like to acknowledge the comments of the referees and Jim Gray, which helped significantly in improving the clarity of the presentation.

7.2 References:

Anon85: Anon et.al., *A Measure of Transaction Processing Power*, Datamation, Cahners Publishing Co. April 1985

Berns87: Bernstien, B.A., et.al., *Concurrency Control and Recovery in Database Systems*, Addison Wesley, 1987.

Bhide87: Bhide, A., Stonebraker, M., *Performance Issues in High Performance Transaction Processing Architectures*, Lecture Notes in Computer Science, Springer Verlag, September 1987, pp 277

Carey84: Carey, M.J., Stonebraker, M. *The Performance of Concurrency Control Algorithms for DBMSs*, Proc. 10th Intl. Conf. on Very Large Data Bases, Singapore, August 1984, pp 107

Date83: Date C.J., *An Introduction to Database Systems*, Vol 2., Addison Wesley, 1983

Eswar76: Eswaran, K. et.al., *The Notions of Consistency and Predicate Locks in a Database System*, CACM, Vol 19, No, 11, November 1976.

Gray78: Gray, J.N., *Notes on Database Operating Systems, Operating Systems*: An Advanced Course, Lecture Notes in Computer Science, Springer Verlag, Berlin, 1978, pp 393

Joshi89: Joshi, A.M., Rodwell, K., *A Relational Database Management System for Production Applications*, Digital Technical Journal, No 8, February 1989, pp 99

Krone87: Kronenberg, N.P., et.al., *The VAXcluster Concept: An Overview of a Distributed System*, Digital Technical Journal, No. 5, September 1987 pp 7

Lehma81: Lehman, P.L., Yao, S.B., *Efficient Locking for Concurrent Operations on B-trees*, ACM Trans. on Database systems, Vol 6, No 4, December 1981, pp 650

Lehma86: Lehman, T.L., *Design and Performance Evaluation of a Main Memory Relational Database System*, Ph.D. dissertation, University of Wisconsin-Madison, Aug 1986.

Lehma89: Lehman, T.J., Carey, M.J., *A Concurrency Control Algorithm for Memory-Resident Database Systems*, Proceedings of Third International Conference on Foundations of Data Organization and Algorithms, June 1989.

Renga89: Rengarajan, T.K., et. al., *High Availability Mechanisms of VAX DBMS Software*, Digital Technical Journal, No 8, February 1989, pp 88

Snama87: Snaman, W., Thiel, D.W., *The VAX/VMS Distributed Lock Manager*, Digital Technical Journal, No. 5, September 1987, pp 29