Locking and Latching in a Memory-Resident Database System

Vibby Gottemukkala
Georgia Institute of Technology
Atlanta, GA 30332
vibby@cc.gatech.edu

Tobin J. Lehman
IBM Almaden Research Center
San Jose, CA 95120-6099
toby@almaden.ibm.com

Abstract

As part of the *Starburst* extensible database project developed at the IBM Almaden Research Center, we designed and implemented a memory-resident storage component that co-exists with *Starburst*'s disk-oriented storage component. The two storage components share the same common services, such as query optimization, transaction management, *etc.* However, the memory-resident storage component is faster than the disk-oriented storage component and hence needs faster run-time services. This paper examines two run-time services, the lock manager and the latch mechanism, and investigates possible cost-cutting measures. We propose the use of of a single latch for protecting a table, all of its indexes, and all of its related lock information, in order to reduce storage component latch costs. We then show that although a table-level latch is a large granule latch, it does not significantly restrict concurrency. We also examine traditional lock manager design and suggest a different design that is appropriate for memory-resident storage components. The new design exploits direct addressing of lock data and dynamic, multi-granularity locks. Performance measurements of the new lock manager show that it outperforms the regular *Starburst* lock manager, which is of a traditional lock manager design, by as much as 60%.

Proceedings of the 18th VLDB Conference
Vancouver, British Columbia, Canada 1992

keywords: Memory-resident database, lock manager, latch, Starburst.

1 Introduction

The performance of relational database management system (DBMS) implementations has often been a problem for non-traditional database applications wishing to use the attractive features of the relational model. Applications such as language-based editors [Horwitz 85], program development environments [Linton 84], and performance monitoring tools [Snodgrass 84] could benefit from using the relational model, yet they require better performance than that typically associated with disk-oriented database systems. Thus, application designers are often frustrated in choosing between database systems that offer either function or performance, but never both.

The advent of object-oriented database systems, such as Object Store [Lamb 91], has further clouded the issue of which type of database system to use. Object Store uses a memory-based storage component and, as a result, is able to perform database operations at *memory speeds.* Furthermore, the virtues of object-oriented programming have already contributed to many success stories [Michaels 91, Lamb 91]. However, those applications that need support for relational operations, as well as high-performance, have not yet had their needs met.

One solution to the non-traditional database application designer's dilemma would be a database system that offers the performance of a memory-based storage component with the function of a relational database system. At the IBM Almaden Research Center, we are building such a system. As the base relational database system, we are using *Starburst*, an extensible database system prototype [Haas 89, Haas 90]. One of the goals of *Starburst* is to support experimentation with storage components through the use of its extensible Data Management Extension Architecture

(DMEA) [Lindsay 87]. DMEA provides database implementors with a simple interface for creating new storage methods and attachments. In *Starburst* terms, *storage methods* manage tables and their associated records, while *attachments* manage data structures related to tables, such as indexes. We will use the term *storage component* to designate a storage method and its associated attachments.

The two *Starburst* storage components of interest in this paper are the traditional disk-oriented storage component, known as the Vanilla Relation Manager (VRM), and the high-performance memory-resident storage component, known as the Main Memory Manager (MMM) [Lehman 92]. MMM is intended to be the storage component one would use when storing data for which fast response-time is crucial, while for less critical data, or for data that does not fit in memory, VRM would be used.

For MMM to achieve its full performance potential, it has to not only be stream-lined itself, but it needs to interact only with other software components that have also been optimized for performance. In *Starburst*, a set of common services is provided for storage components. This set of common services comprises such software components as a memory storage manager, an event queue manager, a default logging and recovery manager, a latch mechanism, and a default lock manager. A storage component such as MMM can use the standard software component provided by the common service, or it can use an alternative, more specialized software component, if one exists. Thus, we examined each software component in the set of *Starburst* common services to determine where improvement was possible. Since we are concerned only with high-performance software components that would work with the MMM storage component, we explored the possibility of letting these software components exploit the unique features of MMM in order to increase performance. Under this criterion, we determined that there was room for improvement in the use of the latch mechanism, the design of the lock manager, and the design of the recovery manager.

Much work has already been done in the area of efficient logging and recovery mechanisms for memory-resident database systems [DeWitt 84, Eich 87, Hagmann 86, Kumar 91, Lehman 87, Salem 86, Salem 90]. Less work has been done in the area of synchronization: latching and locking. Therefore, in this paper, we focus on efficient use of the latch mechanism and lock manager in the context of the MMM storage component.

The remainder of this paper is organized as follows: Section 2 presents a minimal-cost latching scheme along with an analysis that shows that such a scheme provides sufficient concurrency. Section 3 briefly describes the design of both the regular *Starburst* lock manager and the MMM lock manager, and then Section 4 compares the performance of the two lock managers. Finally, Section 5 presents our conclusions.

2 Reducing Latch Cost

One of the common services we have identified as potentially reducing the performance of the MMM storage component is the latch mechanism. A latch, or short-term lock, is a low level primitive that provides a cheap serialization mechanism with shared and exclusive lock modes, but no deadlock detection [Gray 78]. In *Starburst*, latches are used to, among other things, gain exclusive or shared access to buffer pool pages. Each time the VRM storage component needs access to a page, it contacts the buffer pool which then latches the page in the buffer pool in a shared or exclusive mode.

A latch operation typically involves far fewer instructions than a lock operation, as a latch's data structures are statically allocated and directly addressable. In fact, in R* [Williams 82], a distributed relational database system prototype developed at IBM research, a latch and unlatch operation used about 20 CISC (IBM 370) instructions,[1] which was roughly an order of magnitude less than a lock and unlock operation [Yost 92, Gray 89, Lehman 89]. Thus, given a latch's relatively short pathlength, one might dismiss the concern of latches imposing a significant overhead in the overall storage component pathlength.

2.1 Design Alternatives

One could imagine different MMM designs depending on latch cost. If latch cost were indeed insignificant, as some numbers indicate, one might consider a design of the MMM storage component that mimicked the VRM design, *i.e.* where individual latches control a table's memory pages (referred to in MMM as *partitions*[2]), the nodes in a (T Tree [Lehman 86]) index, and the lock manager data structures. The many fine-grained latches would ensure that sharing was not limited. If, on the other hand, latch cost turned out to be more substantial, one might consider an alternative MMM design where all latches were removed, except one: the table latch. In this design, latches would not appear in the partitions that comprise an MMM table, they would not appear in any table control struc-

[1] Complex Instruction Set Computer (CISC) instructions may correspond to several Reduced Instruction Set Computer (RISC) instructions.

[2] In MMM, a table is stored in a variable-length *segment*. A segment is composed of a number of fixed size *partitions*.

tures, *e.g.* table statistics, they would not appear in the nodes of any MMM index, and they would not appear in the data structures of the MMM lock manager. Such a design is feasible in a memory-resident database environment, as the lack of disk I/O reduces the chances of a transaction being pre-empted while holding the large-grained table latch.

To compare the alternative designs, we performed a number of experiments on *Starburst*, measuring the amount of time spent in individual components, such as VRM, MMM, the lock component and the latch component.[3] Table 1 shows the latch overhead, expressed as a percentage of the overall storage component cost, for the VRM storage component, and for two design configurations of the MMM storage component: table-only latches, and node/partition/lock latches. Various scan operations were performed on a 10,000 tuple table, using a 16-field, 208-byte tuple, as defined by the Wisconsin Benchmark database [Bitton 83]. In Table 1, STC is the total storage component time, including the latch overhead. The VRM table scan and the MMM table/index scans using table-latches were measured. The MMM table/index scans (using node/partition/lock latches) were computed using the measured table latch times.

As a reference point for latch costs in MMM, we measured latch overhead in a VRM table scan. To our surprise, and contrary to the idea that latches are cheap, latch overhead was significant even in the VRM table scan case — 19% of the overall VRM pathlength was due to latching. Further examination reveals some of the reasons: First, each call to VRM results in at least three latches being set (two to fix a page in the buffer pool, one to release it). In a 10,000 tuple table, this alone is 30,000 latch calls. Second, although some database systems such as R* might seem to have inexpensive latch operations, it is not guaranteed that all systems will have cheap latch calls. As we mentioned earlier, an R* latch/unlatch operation used about 20 IBM 370 instructions. However, R* was able to exploit the powerful IBM 370 *compare and swap* instruction, which greatly simplified the latch implementation.

Some machines, such as our test machine — an IBM Risc System 6000 Model 530 running AIX 3.1, do not have an equivalent compare-and-swap instruction. In fact, our test machine doesn't even have a *test and set* instruction. For synchronization, we were forced to use an SVC (a kernel supervisor) call to perform the compare and swap function. On our Risc System 6000, we measured the *Starburst* latch/unlatch operation using a hardware clock that has a resolution of 256 nanoseconds. When the latch operation was called repeatedly, suggesting that the instructions and data for the latch operation were in the processor cache, the latch/unlatch operation took about 7.3 microseconds. For more random, isolated latch calls, the operation took as much as 18 microseconds. Estimating the performance of our Model 530 at 10 MIPS, we calculate that the latch/unlatch pair uses about 73 RISC instructions.

In the case of the MMM table scan (Table 1), there is little difference between table latching and partition latching. In this particular example, the table-latch case sets a latch per MMM storage component call (10,000 latch calls). The partition-latch case sets a latch per lock (one table lock), a latch per partition (60 partitions in this table), and a latch per MMM call (10,000), for a total of 10,061 latch calls. The disparity between the two latching paradigms is more apparent when we examine the index case. MMM indexes contain only pointers to table data [Lehman 86, Lehman 92], so table data must be latched whenever an index is used. Thus, if our MMM design used individual latches in each table partition, each index node, and each lock data structure, an index scan of the entire table would require at least three latches per call: an index node latch, a table partition latch, and a lock data structure latch, plus a latch call for each new index node encountered (240 index nodes), for a total of (((3 x 10,000) + 240) = 30,240 latch calls). This would result in a latch overhead of 37% of the MMM storage component cost. In contrast, the table-latch case incurs one latch call per MMM invocation, which is at most a 15% overhead.

When only two tuples are fetched from an index, the differences in the two latch organizations become more apparent. The table-latch design requires only three latches, one for each call to MMM (the third call returns end-of-scan), whereas the other latch design requires a latch call for each index node touched during the search (8), plus a partition latch for every reference to the table (16), plus a partition and index node latch for three scan calls (6), plus two latch calls for the tuple locks for a total of 30 latch calls. In this case the table-latch is overhead is 6% of the MMM cost, whereas the other method imposes a 41% overhead. A similar argument applies to the insert case.

2.2 What About Reduced Concurrency?

It appears that the use of table latches over index node, partition and lock latches can significantly improve the performance of MMM, but will the use of coarse-grained table latches reduce concurrency? If one latch controls all access to a table, including lock information, then both read and write operations on

[3] Section 4 gives some more detail on how the tests were conducted and what hardware was used.

Description	STC time	Latch overhead	Latch call count	Latch percentage of STC time
VRM Full Table Scan, using (buffer) page latches	1293 ms	248 ms	34,028	19%
MMM Full Table Scan (node/index/lock latches)	450 ms	73 ms	10,061	16%
MMM Full Table Scan (table latches)	450 ms	73 ms	10,000	16%
MMM Full Index Scan (node/index/lock latches)	598 ms	221 ms	30,240	37%
MMM Full Index Scan (table latches)	450 ms	73 ms	10,000	16%
MMM Index Scan: Fetch 2 tuples (node/index/lock latches)	0.536 ms	0.219 ms	30	41%
MMM Index Scan: Fetch 2 tuples (table latches)	0.329 ms	0.022 ms	3	6%
MMM table and Index Insert: (node/index/lock latches)	0.552 ms	0.182 ms	25	33%
MMM table and Index Insert: (table latches)	0.377 ms	0.0073 ms	1	2%

Table 1: Latch costs in MMM and VRM.

the table must use an *exclusive* table latch. One danger of using an exclusive table latch is that a process could acquire the latch and then get pre-empted, thus making the table inaccessible to any other process until the latch-holding process awakens and releases the table latch. This is similar to the convoy phenomenon [Blasgen 79]. One way to avoid this problem would be to suspend process pre-emption for the duration of the table latch.[4]Thus, in a uni-processor system, any table-latch contention problems could be avoided. Of course, for fairness reasons, we would bound the latch hold time and periodically force MMM to "come up for air" and give up the table latch. Also, as table latches must not be held by a transaction while it is waiting on a lock, the MMM lock manager must release the transaction's table latch before suspending the transaction.

Although the table-latch case is easily argued for a uni-processor system, what about a multi-processor (MP) environment where transactions running on other processors could potentially be made to wait while one process "hogged" the latch of a frequently accessed table? It is necessary to determine the amount of concurrency, or processor overlap, that is possible using table-level latches. For this purpose we projected processor usage for a multi-processor using uni-processor execution times obtained by measuring the performance of queries run on the MMM storage component.

Table 2 shows the performance measurements of some queries run on *Starburst* using the MMM storage component. We chose a representative set, including long transactions (an index scan that touched every tuple) and short transactions (an insert or fetch). For simplicity, we grouped inserts, deletes and fetches together and gave them a common cost.[5] For the purposes of these tests, a single transaction performed a single operation, such as a table scan, or an insert. A more realistic transaction, one that performs a set of these operations, would be a combination of these low level operations.

The execution times presented in Table 2 represent query execution time only; other costs such as parsing, query optimization, and query compilation, have been excluded from this measurement. Furthermore, the scan-type transactions do not include the time to do anything with the tuple once it has been retrieved, such as printing; tuples returned up to *Starburst*'s Application Programming Interface were discarded.

For the following discussion, we'll use these terms:

λ_i = the arrival rate of type i transactions

α_{ij} = the proportion of the time transaction i is holding table latch j.

S_i = the total CPU time, or service time, of transaction i.

[4]By modifying the AIX Version 3 kernel, we would have the ability to use a fast SVC service that would make a process temporarily exempt from pre-emption by other processes of the same, or lower, priority. Such a call would add approximately 20 instructions to the pathlength of the latch operation. However, we might be able to reclaim some of this pathlength to switching from *Starburst*'s general multi-node latch to a more efficient, single (X) mode latch.

[5]As logging has not yet been implemented for MMM, these figures do not include the cost of writing log records for update transactions.

Trans Type T_i	Service time (sec) S_i	Latch hold time (sec) $S_i * \alpha_{ij}$	Latch hold percentage α_{ij}	Transaction description
T1	0.864	0.450	52%	Table Scan, count 100%
T2	0.794	0.383	48%	Table Scan, return 10%
T3	1.006	0.586	58%	Index Scan, count 100%
T4	0.445	0.107	24%	Index Scan, return 10%
T5	0.007	0.0003	4%	Fetch, Insert or delete

Table 2: Results of benchmarks run on *Starburst*.

Given the query execution time and the percentage of that time spent holding the table latch, we can compute the maximum number of transactions that can be active in the database system. The fundamental limit of resources in the system implies that no table latch (j) can be held for more than one second for each second of real time. The total demand for table latch (j) by transactions of type (i) is $\lambda_i \alpha_{ij} S_i \leq 1$, and for all transaction types, $\sum_i \lambda_i \alpha_{ij} S_i \leq 1$. Thus, we can use the formula

$$\sum_i \lambda_i \alpha_{ij} S_i \leq 1 \; for \; each \; j$$

to compute λ_i, the maximum number of (simple) transactions per second that we could process with an *unlimited* number of processors, for a given mix of transaction types (T_i). In this analysis, it is assumed that all limiting effects (such as multiprogramming level limitations or contention for other resources) have been removed.

Then, given λ_i, Little's Law [Kleinrock 75]

$$N = \sum_i \lambda_i S_i$$

gives us the number of transactions that can be run in parallel, which equates to the processors that can be kept busy *simultaneously*. We refer to this quantity as the amount of *processor overlap*.

For example, suppose we have only one transaction type T, and one relation R. A type T transaction runs for one second and holds the table latch on relation R for 10% of the time. Then, for an unlimited number of processors, the arrival rate is $\lambda * .10 * 1s \leq 1$, or $\lambda \leq \frac{1}{.10 * 1s}$ or $\lambda \leq 10/s$, or 10 transactions per second. Then, given λ, the processor overlap, or number of transactions that can operate simultaneously, is $N = \lambda * T$. Substituting terms $(\lambda = 10\ /s)$ and $(T = 1s)$ gives us $(N = 10)$, 10 transactions being processed in parallel.

So, given the transaction service times and transaction latch hold times from table 2, we can examine various transaction mixes using a specific database and observe both the processor overlap and the rate at which transactions are processed. Our example database is made up of 100 tables, each 10,000 tuples in size. Each tuple contains 16 fields and is 208 bytes long, as described by the Wisconsin Benchmark Database [Bitton 83]. To mimic a realistic distribution of database references, we use the 80–20/80-20 rule. That is, 80% of the references go to 20% of the tables (warm tables). Furthermore, 80% of the references to warm tables go to 20% of the tables (hot tables). Thus, for a 100 table database:

64% of the database references go to 4 tables (hot).
16% of the database references go to 16 tables (warm).
20% of the database references go to 80 tables (cool).

The four hot tables are the bottleneck, so we can use the transaction mix and a single hot table to determine the maximum value of λ. Then, given λ for a single hot table, we can derive the relative values of all the λ_i for each transaction type.

Table 3 shows the total number of (simple) Transactions Per Second (TPS) that could be processed by an unlimited number of processors, and then the actual number of processors that could be kept busy. It also shows the breakdown of TPS over the hot (64%), warm (16%), and cool (20%) tables. The first "# processors busy" column in Table 3 gives a slightly unrealistic estimation of processor overlap, as it does not include any application time, nor does it include any time to process the data returned by the database system. The second "# processors busy" column in Table 3 gives a more realistic estimate, as we doubled the transaction time to allow for application and tuple processing. Recall that the TPS numbers presented here are for relative comparison only, as the transactions used here do not represent real-world transactions. To get real TPS numbers, we would have to evaluate transactions that performed multiple steps involving some combination of scans, inserts, updates, and deletes.

We ran three sets of mixes: first, a completely uniform distribution of all the transaction types from Ta-

Arrival Rate Mix					Hot	Warm	Cool	Total	# procs busy	# procs busy
T1	T2	T3	T4	T5	TPS	TPS	TPS	TPS	(no appl code)	(with appl code)
20	20	20	20	20	13.1	3.3	4.1	20.4	12.7	25.4
60	10	10	10	10	10.6	2.6	3.3	16.5	12.3	24.6
10	60	10	10	10	11.6	2.9	3.6	18.1	12.8	25.6
10	10	60	10	10	9.0	2.2	2.8	14.0	11.4	22.8
10	10	10	60	10	19.4	4.9	6.0	30.3	16.2	32.4
10	10	10	10	60	28.1	6.5	8.1	40.8	12.9	25.8
100	0	0	0	0	8.9	2.2	2.8	13.9	12.0	24.0
0	0	0	100	0	37.4	9.3	11.6	58.3	26.0	52.0
0	0	0	0	100	13,333	3,333	4,133	20,799	145	290

Table 3: Processor overlap and TPS.

ble 2; second, a somewhat skewed mix where 60% of the transaction mix consisted of one transaction type and the remaining transaction types each received 10%; finally, a skewed mix where some of the transaction types accounted for 100% of the mix.

The numbers in Table 3 show that, for a variety of transaction types and a variety of transaction mixes, a reasonable amount of processor overlap is possible using coarse-grained table latches. Thus, we conclude that the use of table latches does not significantly reduce concurrency.

3 Reducing Lock Cost

Besides the latch mechanism, another run-time service that could potentially reduce the performance of the MMM storage component is the lock manager. Compared to the large body of work in the locking family of concurrency control methods (for example, [Agrawal 85, Bernstein 81, Carey 83, Carey 84, Eswaran 76]), there is little or no published work in the area of changing the locking mechanism itself. The System R lock manager described in [Gray 78] appears to be the basic design choice of most database systems, including *Starburst*.

Even though the regular *Starburst* lock manager keeps all of its data in memory, there are features in MMM that a lock manager could exploit to improve performance. Exploiting the extensibility feature of *Starburst*, we implemented a second lock manager, one that was tailor-made for the MMM storage component. We briefly describe the two lock managers to contrast their designs before we compare their relative performance in Section 4. Readers interested in more details of the MMM lock manager or related work in the area of reducing lock cost should consult [Gottemukkala 92].

3.1 The *Starburst* Lock Manager

The control structure of the "regular" *Starburst* lock manager (SB LM) is shown in Figure 1. The SB LM uses a fixed-size hash table to speed lookups to Lock Control Blocks (LCB's), which contain information about locks, such as the name of the lock, the group mode of the lock, and a queue of Lock Request Blocks (LRB's). Each requestor of a lock is assigned an LRB, which contains information about the requestor, such as the requested lock mode, the held lock mode, the status of the lock (held, waiting), and other transaction information. The SB LM maintains a pre-allocated free pool of LCB's per hash table slot and a pre-allocated free-pool of LRB's per transaction, which allow it to set only a single latch per lock operation.

To set a lock on a named object, the *Starburst* lock manager does the following: It computes a hash value for the name of the object to be locked, and then sets a latch on the hash class (or hash slot) corresponding to the hash value. If an LCB for the object is not present, it initializes a pre-allocated LCB for the new lock and attaches it to the hash class chain. If an LRB for the lock requestor is not present, it initializes a pre-allocated LRB, and marks the process status as "running" if there is no conflict with other locks, or marks the process status as "blocked" and suspends the process.

The SB LM supports hierarchical locking with two locking granularity levels: table and tuple. When locking a tuple, an intention lock must first be placed on the table, to synchronize with any other table-level operations that could be active on the same table. Hence, setting a tuple lock results in two lock calls to set the table-level intention lock and the actual tuple lock.

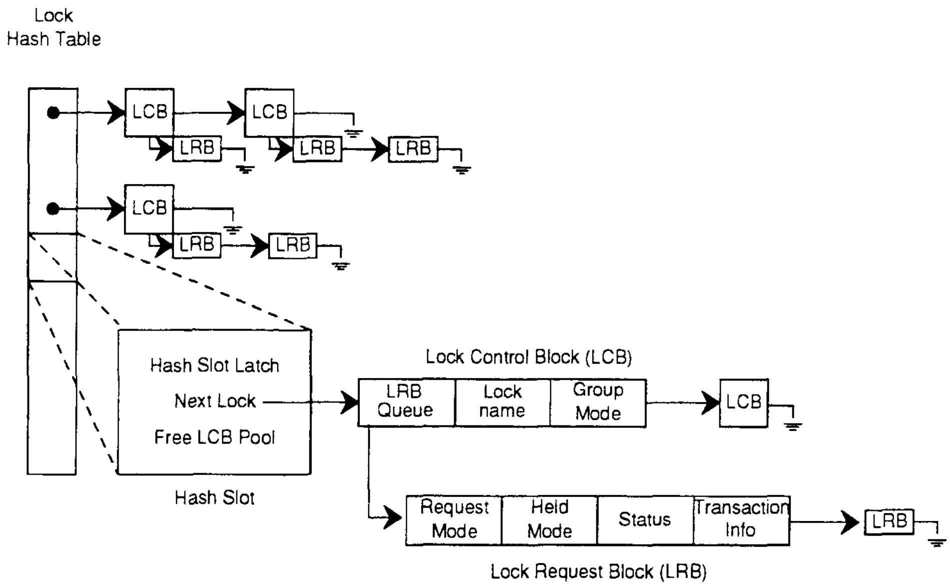

Figure 1: Control structure of the *Starburst* lock manager.

3.2 The MMM Lock Manager

Figure 2 shows the basic data structures used by MMM and the MMM lock manager. MMM manages tables in variable-length segments, which are collections of fixed-length partitions that contain records [Lehman 92]. Each segment has a segment control block, which contains control information, and lock information. As we mentioned in Section 2, a single table latch protects the table and all of its related structures, such as index and lock information. Thus, once MMM has acquired the table latch to reference table data, it has also implicitly latched all of the table's lock data as well.

The MMM Lock Manager (MMM LM) exploits two features of the MMM storage component:

1. MMM Table data is fixed in memory, so MMM LM lock data is attached directly to table and record data, thus eliminating the need for a hash-table lookup operation to locate lock data.

2. MMM Table control data is also fixed in memory, allowing MMM LM to maintain a "locking granularity level" flag for each table, which is used to eliminate the need for locks at multiple levels while maintaining the semantics of hierarchical locking.

The MMM Lock Manager uses two locking granularities: "table" and "tuple." A *locking granularity level* flag, kept in the table's control information, designates the current locking granule size. The MMM LM changes the locking granularity level flag for each table dynamically, depending on the level of sharing required for that table. Locking at the table level is cheaper than locking at the tuple level, so it is the preferred method when fine granularity sharing is not needed. When one or more transactions are blocked while trying to access a table that is locked with another transaction's table lock, the table lock is *de-escalated* into a collection of tuple locks; the higher cost for tuple-level locking is then paid, but the level of sharing is increased. To allow for the possibility of table lock de-escalation, tuple locks that would have been set are "remembered," so that they may be converted into real tuple locks if the need arises. When fine granularity locks are no longer needed, tuple-level locks are escalated into table-level locks. Certain operations that require the use of an entire table can *force* lock escalation to the table level and disable lock de-escalation until they have completed.

Each transaction that sets a lock on a table gets a Table Lock Control Block (LCB) which tracks the aggregate mode of the transaction's locks on that table. The Table LCB also maintains the list of remembered locks that a transaction acquires while a table lock is in effect. A transaction's remembered lock that is compatible with the aggregate lock modes of all the other running transactions with LCB's on a table is referred to as a *granted lock*. On the other hand, a transaction's remembered lock that is not compatible with the aggregate lock modes of the table's other LCB's, and thus causes the transaction to block, is referred to

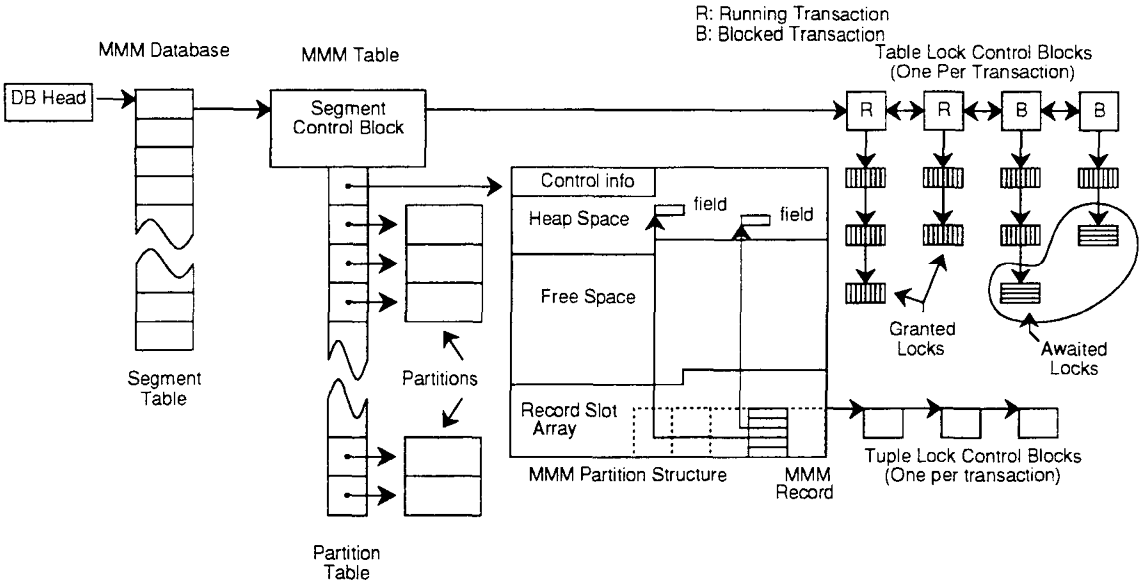

Figure 2: Structure of the MMM Lock Manager.

as an *awaited lock*.

4 Performance Experiments

In this section we compare the performance of the MMM Lock Manager (MMM LM) with that of the *Starburst* Lock Manager (SB LM). The hardware used for our experiments was a model 530 IBM Risc System 6000 workstation with a 25 MHz processor and a 64 Kbyte I&D cache. The peak integer performance of a model 530 Risc System 6000 is nominally 25 MIPS, but in practice we have estimated that 10 MIPS is closer to the mark for database workloads because of poor instruction and data locality, which causes processor cache misses. Our machine was configured with 128 megabytes of memory, which was more than enough to keep all data cached in memory during test runs. Furthermore, to ensure that we were getting real memory-resident performance numbers, we added special sanity checks to *Starburst* to verify that no I/O occurred in any of the test runs. All our tests were run under AIX Version 3.1.

A real-time hardware clock on the Risc System 6000 with a resolution of 256 nanoseconds was used to measure query execution time during test runs. We instrumented *Starburst* with assembly-routine macros that made use of the Risc System 6000's hardware clock facility to time key code segments in *Starburst*. Thus, we were able to not only obtain the overall execution time of a query, but we were also able to obtain an accurate breakdown of where the time was being spent in the query's execution.

4.1 The Results

As we were interested in the most common case where there was no lock contention, we measured lock cost by setting non-conflicting read locks. The execution times in Table 4 are presented in four groups. The first group shows the execution times of the lock, unlock, and combined (lock/unlock) operations performed by the regular *Starburst* lock manager (SB LM). The second group shows the execution times of the lock, unlock and combined operations performed by the MMM lock manager (MMM LM) for setting "remembered" locks. A remembered lock is set instead of a real tuple lock when the lock granularity level of a table is at the table level. The third group shows execution times of the same operations performed by the MMM lock manager for setting real tuple locks, as would be the case when the table granularity flag is at the tuple level. The last group shows the cost of de-escalating a table lock (essentially, a remembered lock) into a real tuple lock.

The execution times presented in Table 4 represent the *minimum* execution times for both SB LM and MMM LM. Since the SB LM is a "traditional" lock manager which stores lock data in a chained-bucket hash table, its performance is sensitive to the number of locks set. *Starburst* memory constraints prevented us from creating a lock hash table larger than 1,000 slots, so lock numbers greater than 500 caused SB LM's performance to decrease.[6] The best perfor-

[6] By using a dynamic hashing algorithm, such as linear hashing [Litwin 80] or modified linear hashing [Lehman 86], we could

mance was obtained from SB LM when setting and releasing 100 locks, 25 microseconds and 12 microseconds, respectively.

For a sanity check, we compare the performance of the *Starburst* lock manager with two other tuned lock managers: the R* lock manager and the GAMMA database machine lock manager. We estimate the 37 microseconds for the SB LM lock/unlock call at 370 RISC instructions, and compare that with the 150 IBM 370 (CISC) instructions that were used by a lock/unlock call in R* [Gray 89], and the 235 DEC VAX (CISC) instructions that were used by a lock/unlock call in the GAMMA database machine [DeWitt 90, Ghandeharizadeh 89]. It is difficult to compare RISC and CISC instructions at this level and draw any conclusions, but we can make some observations. The 150 instructions used by R* actually contained 5 occurrences of the *compare and swap* instruction, which correspond roughly to 20 or 25 RISC level instructions. Furthermore, it is not uncommon to estimate one CISC instruction at 1.5 to 2 RISC instructions. From these statements, and our own knowledge of the SB LM, we feel that the SB LM is a reasonable implementation of a lock manager.

As expected, the MMM lock manager tuple lock cost is less than that for the *Starburst* lock manager, as the direct access to the Lock Control Block (LCB) avoids the cost of the hash-table lookup step. Additionally, because of the direct access, MMM LM is not affected by the number of locks set on a table. Overall tuple locking cost for the MMM lock manager is derived from a combination of the two lock granularity modes: table and tuple. When there is no contention for the table, all of the locks set are table-level, and the tuple locks are remembered. In this mode, the unlock cost reflects processing the table-level LCB (checking for waiting transactions, *etc.*) and then throwing away (recycling) the remembered tuple-lock data structures. Thus the cost of unlock for remembered tuple locks is relatively small (5 μs), as no checking is needed. When there is contention for the table, some of the locks set are real tuple-level locks, although, recall that only one lock is set per tuple, as table-level intention locks are not needed. The tuple-unlock cost of the MMM lock manager and the *Starburst* lock manager are similar, as they perform similar functions. Both lock managers traverse the LCB chains, set a latch on each LCB, and check for waiting transactions. In fact, the MMM LM is slightly slower when unlocking a tuple because of the extra checking done to test for possible lock granularity escalation.

Not shown in Table 4 is the time used by the MMM lock manager to set a fixed table lock, which is 10 microseconds for lock and 10 microseconds for unlock (20 combined). The first lock call in a table, for either a remembered or real tuple lock, also incurs the cost of setting the initial table lock. Hence the *first* tuple lock (and subsequent unlock) operation costs approximately 31 (20 + 11) and 44 (20 + 24) microseconds, for remembered and tuple locks respectively.[7] Thus, using regular locking and intention locks, the *Starburst* lock manager would require 74 microseconds to lock and unlock the first tuple, whereas the MMM lock manager would require either 31 or 44 microseconds. Subsequent tuple-lock calls to the *Starburst* lock manager would still cost 74 microseconds, whereas subsequent calls to MMM LM would cost 11 or 24 microseconds, for remembered or real tuple locks, respectively. If we were to move the intention lock checking logic into the *Starburst* storage components that use the regular *Starburst* lock manager, then we would be able reduce the lock cost somewhat, although the amount is difficult to quantify. Notice also that a table-scan operation does not have the problem of repeatedly resetting the table-level intention lock, as the logic of the table-scan code is such that the intention lock is set exactly once. However, repeated probes with an index, or any modifying operation, such as insert, delete, or update, do repeatedly set table-level intention locks in *Starburst.*

Finally, we computed the cost of de-escalating remembered locks into tuple locks. At first glance, the cost of de-escalation might appear to be the cost of setting a regular tuple-lock for each remembered lock, spending 24 + 11 = 35 microseconds total per tuple lock. Fortunately this is not the case. Consider the following:

- All *running* transactions have remembered locks that have a compatible lock mode.
- All remembered locks of *blocked* transactions have compatible lock modes, with the exception of the last one (the request which caused the transaction to block).

So the process of conversion of remembered locks to "real" locks involves:

1. Converting all remembered locks of running transactions into real locks.
2. Converting all non-blocking (granted) remembered locks of blocked transactions into real locks.

eliminate this problem for any number of locks.

[7]The first lock call is actually a bit less than this, as both operations are done in a single lock call — however it's easier to explain it this way.

Operation	Cost (microseconds)
SB LM Lock	25 μs
SB LM Unlock	12 μs
SB LM Combined	37 μs
MMM LM (seg + Remembered) Lock	6 μs
MMM LM (seg + Remembered) Unlock	5 μs
MMM LM (seg + Remembered) Combined	11 μs
MMM LM (tuple) Lock	9 μs
MMM LM (tuple) Unlock	15 μs
MMM LM (tuple) Combined	24 μs
MMM LM De-escalate Op	2.3 μs

Table 4: Execution times of a Read Lock/Unlock call for *Starburst* and MMM

3. Converting all blocking (awaited) remembered locks of blocked transactions into real locks. This is almost equivalent to a tuple lock call. However, the number of these will be low — equal to the number of blocked transactions.

Since all non-blocking (granted) remembered locks have compatible lock modes, steps 1 and 2 only involve adding the lcbs to the tuple lock chains; no compatibility checks are needed. Also, no new tuple LCBs need to be allocated because they have already been allocated as remembered locks (remembered and real tuple locks use the same data structure).

In fact, we found that the cost of remembering a lock and then de-escalating it later appears to be slightly less than the cost of setting the tuple lock in the first place. We've seen this effect before in our benchmark experiments. When an operation is repeatedly done in "batch" mode, it runs faster — probably due to a better hit ratio in the processor instruction and data cache.

4.2 The Bottom Line

Consider the transaction execution numbers presented earlier in Table 2. The latch-hold times represent the amount of time spent in the MMM storage component. If we compute the number of locks set by these transactions and then multiply that by the lock cost, we'll be able to compare the total time spent locking with the total time spent in the MMM storage component. The interesting cases are ones in which locking cost plays a major role, such as index scan or update transactions. To keep the numbers consistent, the MMM storage component and transaction times in Table 5 do not include any lock manager time. The lock times measured during the runs to generate Table 2 were subtracted from the MMM and transaction times.

Table 5 shows the bottom line. For the three lock costs (SB LM lock/unlock, MMM LM real tuple lock/unlock, and MMM LM remembered lock/unlock) we display the cost of the lock call in milliseconds, and we also display the lock overhead as a percentage of the total MMM time used for locking ($\frac{lock\ time}{MMM\ time\ +\ lock\ time}$). The benefits vary depending on transaction type, the specific lock mode needed by the MMM LM (remembered or real tuple lock), and the amount of unnecessary intention-mode locking performed by SB LM. However, except for the case of table scans where locking cost is not significant, the MMM lock manager can reduce SB LM locking cost by 30% to 60%, as a result of using more efficient data structures and setting cheaper coarse-grained locks when fine-grained sharing is not needed. These savings equate to a 15% to 30% reduction in MMM time and a 5% to 20% reduction in overall transaction time. Additional savings can be produced by MMM LM in those cases where SB LM is called to set redundant table-level intention locks.

5 Conclusion

We have described our attempts to reduce the lock and latch costs in the *Starburst* MMM storage component. We have shown that the use of table-level latches in MMM provides provides up to a 35% improvement in storage component performance, while not significantly reducing concurrency.

As an example of *Starburst* extensibility, the MMM lock manager exists in *Starburst* side-by-side with the regular *Starburst* lock manager. It supports the standard *Starburst* lock manager interface, and it communicates with the global deadlock detector that interfaces to all lock managers (currently, there are only

Trans Type	Transaction time	MMM time	Number of Locks set	SB LM cost	MMM LM hi cost	MMM LM lo cost
T1	864 ms	450 ms	1	0.074 ms (.02%)	.024 ms (.01%)	.011 ms (0%)
T2	794 ms	383 ms	1	0.074 ms (.02%)	.024 ms (.01%)	.011 ms (0%)
T3	910 ms	496 ms	10,000	370 ms (42%)	240 ms (33%)	107 ms (17%)
T4	433 ms	85 ms	1,000	37 ms (30%)	24 ms (22%)	11 ms (11%)
T5	7 ms	0.3 ms	1	0.074 ms (20%)	0.044 ms (12%)	0.031 ms (9%)

Table 5: Comparing lock costs with MMM storage component costs.

two). We described the design and implementation of the MMM lock manager. By attaching Lock Control Blocks directly to the data and using dynamic multi-granularity locking, we can achieve locking costs that are up to 60% less than that of the *Starburst* lock manager. This lock cost reduction translates into an MMM storage component performance improvement of up to 30%, and an overal transaction response-time improvement of up to 20%.

We have shown that common services, such as the latch mechanism and the lock manager, are important to the performance of a database system storage component, and to the database system overall. In the case of the *Starburst* MMM storage component, reducing lock and latch costs has improved MMM performance by up to 65%, which corresponds to an improved transaction response time of up to 30%.

6 Acknowledgments

Thanks to Bill Cody, Robert Morris, Eugene Shekita, Kurt Shoens, Jim Stamos, Joel Richardson, Laura Haas, and the VLDB referees for their assistance with this paper.

7 References

[Agrawal 85] R. Agrawal, M. Carey, and M. Livny, "Models for Studying Concurrency Control Performance: Alternatives and Implications," *Proc. ACM SIGMOD Conf.*, May 1985.

[Bernstein 81] P. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems," *ACM Computing Surveys 13, 2*, June 1981.

[Bitton 83] D. Bitton, D. DeWitt, and C. Turbyfill, "Benchmarking Simple Database Operations," *Proc. 9th VLDB Conf.*, November 1983.

[Blasgen 79] M. Blasgen, J. Gray, M. Mitoma, and T. Price, "The Convoy Phenomenon," *Operating Systems Review 13, 2* April 1979.

[Carey 83] M. Carey, "An Abstract Model of Database Concurrency Control Algorithms," *Proc. ACM SIGMOD Conf.*, May 1983.

[Carey 84] M. Carey and M. Stonebraker, "The Performance of Concurrency Control Algorithms for Database Management Systems," *Proc. 10th VLDB Conf.*, August 1984.

[DeWitt 90] D. DeWitt et al., "The Gamma Database Machine Project," *IEEE TKDE 2, 1*, March 1990.

[DeWitt 84] D.J. DeWitt et al, "Implementation Techniques for Main Memory Database Systems," *Proc. ACM SIGMOD Conf.*, June 1984.

[Eich 87] M. Eich, "A Classification and Comparison of Main Memory Database Recovery Techniques," Proc. of the 3rd Int. Conf. on Data Engineering, February 1987, pp 332-339.

[Eswaran 76] K. Eswaran, J. Gray, R. Lorie, and I. Traiger, "The Notions of Consistency and Predicate Locks in a Database System," *CACM Vol 19, No 11* , November 1976.

[Ghandeharizadeh 89] S. Ghandeharizadeh, implementor of the GAMMA database machine lock manager, Personal communication, February 1989.

[Gottemukkala 92] V. Gottemukkala, T. Lehman, "The Design and Performance Evaluation of a Lock Manager for a Memory-Resident Database System," submitted for publication, Feb 1992.

[Gray 78] J. Gray, "Notes on Database Operating Systems," *Operating Systems, An Advanced Course,* vol. 60, Springer-Verlag, New York, 1978.

[Gray 89] J. Gray, Implementor of the R* lock manager, Personal communication, February 1989.

[Haas 89] L. Haas et al., "Extensible Query Processing in Starburst," *Proc. ACM SIGMOD Conf.*, June 1989.

[Haas 90] L. Haas et al., "Starburst Mid-Flight: As the Dust Clears," *IEEE TKDE 2, 1*, March 1990.

[Hagmann 86] R. Hagmann, "A Crash Recovery Scheme for a Memory-Resident Database System," *IEEE Trans. on Computers C-35, 9*, September 1986.

[Horwitz 85] S. Horwitz and T. Teitelbaum, "Relations and Attributes: A Symbiotic Basis for Editing Environments," *Proc. ACM SIGPLAN Conf. on Lang. Issues in Prog. Env.*, June 1985.

[Kleinrock 75] L. Kleinrock, *Queuing Systems Vol I: Theory*, Wiley, 1975.

[Kumar 91] V. Kumar and A. Berger, "Performance Measurement of Some Main Memory Database Recovery Algorithms," *Proc. of the 7th Int. Conf. on Data Engineering*, April 1991.

[Lamb 91] C. Lamb, G. Landis, J. Orenstein, Dan Weinreb, "The ObjectStore Database System," *CACM Vol 34, No 10* , October 1991.

[Lehman 86] T. Lehman and M. Carey, "A Study of Index Structures for Main Memory Database Management Systems," *Proc. 12th Conf. Very Large Data Bases,* August 1986.

[Lehman 87] T. Lehman and M. Carey, "A Recovery Algorithm for a High-Performance Memory-Resident Database System," *Proc. ACM SIGMOD Conf.*, May 1987.

[Lehman 89] T. Lehman and M. Carey, "A Concurrency Control Algorithm for Memory-Resident Database System," *Foundations of Data Organization and Algorithms*, FODO 1989.

[Lehman 92] T. Lehman, E. Shekita, and L.F. Cabrera, *An Evaluation of the Starburst Memory-Resident Storage Component*, Submitted for publication, Feb 1992.

[Lindsay 89] B. Lindsay, R* implementor, personal communication, Feb 89.

[Lindsay 87] B. Lindsay, J. McPherson, and H. Pirahesh, "A Data Management Extension Architecture," *Proc. ACM SIGMOD Conf.*, June 1987.

[Linton 84] M. Linton, "Implementing Relational Views of Programs," *Proc. ACM SIGSOFT-SIGPLAN Symp. on Practical Software Development Environments*, April 1984.

[Litwin 80] W. Litwin, "Linear Hashing: A New Tool for File and Table Addressing," *Proc. 6th Conf. Very Large Data Bases,* October 1980.

[Michaels 91] J. Michaels, "Managing Money with Objects," *Wall Street Computer Review*, Vol 9, No. 3, December 1991.

[Salem 90] K. Salem and H. Garcia-Molina, "System M: A Transaction Processing Testbed for Memory-Resident Data," *IEEE TKDE 2, 1*, March 1990.

[Salem 86] K. Salem and H. Garcia-Molina, *Crash Recovery Mechanisms for Main Storage Database Systems*, Tech. Rep. No. CS-TR-0340-86, CS Dept., Princeton Univ., April 1986.

[Snodgrass 84] R. Snodgrass, "Monitoring in a Software Development Environment: A Relational Approach," *Proc. ACM SIGSOFT-SIGPLAN Symp. on Practical Soft. Dev. Env.*, April 1984.

[Williams 82] R. Williams *et al*, "R*: An Overview of the Architecture," Proc. of the Int. Conf. on Database Systems, Jerusalem, Israel, June 1982. Published in Improving Database Usability and Responsiveness, P. Scheuermann, ed. Academic Press, N.Y.

[Yost 92] R. Yost, Implementor of the R* latch mechanism, personal communication, February 1992.