Highly Concurrent Locking in Shared Memory Database Systems

Christian Jacobi and Cédric Lichtenau

University of Saarland — Computer Science Department
PO Box 151150, 66041 Saarbruecken, Germany
{cj,cls}@cs.uni-sb.de,
http://www-wjp.cs.uni-sb.de/

Abstract. In parallel database systems, conflicts for accesses to objects are solved through object locking. In order to acquire and release locks, in the standard implementation of a lock manager small sections of the code may be executed only by a single thread. On massively parallel shared memory machines (SMM) the serialization of these critical sections leads to serious performance degradation. We eleminate the serialization by decomposing the complex database lock needed for granular locking into basic lock primitives. By doing so, we measured a speedup of a factor 200 on the SB-PRAM. Our method can be ported to any architecture supporting the used lock primitives, as most SMMs do.

1 Introduction

In parallel database systems (DBS), the access to objects (e.g., relation, page, record) must be regulated to ensure transaction isolation. This is done by locking objects before using them. Standard locking implementations rely on a complex database lock control structure. Small sections of the code accessing this structure must be serialized to prevent race conditions between concurrent threads. In massively parallel systems like the TERA MTA or the SB-PRAM running a very large number of threads (in the order of thousand), not only the avoidance of long lock duration becomes crucial. When many short transactions requests a lock on the same object, the serialization in the lock manager leads to heavy performance degradation even if virtually all lock requests are compatible. This situation arises frequently on relation granularity if intentional locks are used. We present a novel approach to the locking problem that prevents this unnecessary serialization. We replace the complex lock control structure by a combination of efficient lock primitives that are typically supported in SMMs. We show that this approach dramatically speeds up the lock manager on massively parallel computers. A speedup factor of 200 was measured on the SB-PRAM, a 64 processors SMM. Our implementation of the lock manager can easily be ported to other SMMs since it requires only standard locks available on most SMMs.

In the next section we introduce the shared memory model and basic lock primitives used later on. In section 3 we shortly describe multi-granular locking. In section 4 we outline the standard implementation of the database lock and

P. Amestoy et al. (Eds.): Euro-Par'99, LNCS 1685, pp. 477–481, 1999.

its drawbacks. Section 5 presents our new approach to DBS locking. Finally we compare the performance of both implementations.

2 Hardware Model

In the shared memory model, an arbitrary large number of processors can concurrently access a shared memory. There exist several implementation of this model such as the NYU Ultracomputer [3], the SB-PRAM [2] and the TERA Multithreaded Architecture [1]. These computers have between 8 and 64 processors and are capable of running up to over thousand threads in parallel. These machines also feature hardware support for basic lock primitives in oder to efficiently enable parallel work and fast synchronization of so many threads. We introduce three basic locks that will be used later on:

Binary Lock At any point of time, only a single thread can hold this lock.

Group Lock An n-group lock, $n \in \mathbb{N}$, manages n groups. A thread may request a lock for a group i, $1 \leq i \leq n$. At any time, multiple locks for the same group may be granted, but never for two different groups (cf. Table 1).

Reader/Writer Lock (R/W-lock) The R/W-lock is a variation of the group lock. There are two groups, namely readers and writers. At any time, an R/W-lock can be helt either by multiple readers or by a single writer.

Our locking code was developed and benchmarked on the SB-PRAM, an PRAM emulation with 64 processors running 2048 threads built at University of Saarbruecken. The concurrent locking by an arbitrary number of threads on the SB-PRAM is executed in parallel without any kind of serialization, if the locks are compatible. Other systems like the NYU Ultracomputer or the TERA MTA have little serialization in the memory modules.

3 Locking in DBS

To ensure transaction isolation, virtually all database systems use locking. The basic lock modes are shared (S) and exclusive (X). An S lock on an object r, which is either a relation, a page, or a tupel, allows reading of r. An X lock allows reading and writing on r. To enable granular locking, intentional lock modes IX, IS and SIX are supported in addition to X and S locks. Before requesting an X, IX

requested	already granted lock		
lock	g1	g2	g3
g1	+	−	−
g2	−	+	−
g3	−	−	+

requested	already granted lock	
lock	read	write
read	+	−
write	−	−

Table 1. Three group lock and R/W-lock matrix. + compatible, − incompatible

or SIX lock on fine granuls, there must be IX or SIX locks set on coarser granules. Similarly, before requesting IS or S locks, the coarser granules must be locked IS or IX. Table 2 illustrates the compatibility matrix of the five lock modes X, S, IX, IS and SIX. A survey of isolation concepts is given in [4].

We define a **CC-lock** to be a data structure with operations cc-lock() and cc-unlock() that works as a lock according to this compatibility matrix. In the next sections we will describe two implementations of this CC-lock. The first implementation is well known, but suffers poor performance on massive parallel shared memory architectures. The second implementation eliminates this disadvantage. Both implementations acquire and release locks via operations on the shared memory. In contrast to most distributed database systems, no designated lock manager is used. Molesky and Ramamritham call this *autonomous locking* and showed its superior performance on SM systems [6].

4 Standard CC-Lock Implementation

On the standard implementation of the CC-lock, each CC-lock consists of two lists, one holding granted lock requests, the other one holding waiting lock requests. A new lock for mode φ is granted immediately iff φ is compatible to all granted locks and to all waiting lock request. The new lock request is added to the waiters list, otherwise.

On a lock release, waiting lock requests (if any) are checked one by one for compatibility with the active locks in the same order as they where appended to the waiters list (FIFO). Waiting locks get activated until the first incompatible waiting request is encountered. Activation can for example be done by interrupts or by polling of the waiting threads on a shared memory variable.

To the best of our knowledge, in all implementations following this basic concept, concurrent accesses of multiple threads to the same CC-lock get serialized by a binary lock. The SB-PRAM implementation of this CC-lock has a critical section of 60 instructions [8].

The SB-PRAM supports 2048 hardware threads, about 2000 threads are availabe for transaction processing, the others are used by the operating system. Suppose the 2000 threads concurrently request compatible locks on the same object. The last thread then has to wait $2000 \cdot 60 = 120$K instructions until it can acquire the requested lock, although all requested locks are compatible. This situation arises frequently with IS/IX locks on coarse granule. For example, each access to a page or tupel needs a previously set IS or IX lock set on relation granularity. This results in a 120K instructions lower bound for transaction length, since the first thread cannot enter the release-lock critical section before the last thread has finished its request-lock critical section. A lower bound of 120K instructions is far too high for short transactions, like debit/credit transactions.

The claimed 120K instructions lower bound only holds, if all threads lock the same object in a round-robin fashion. If the object is not locked round-robin, then 120K instructions is only the expectation.

requested	already granted mode					
lock mode	free	X	S	IS	IX	SIX
X	+	−	−	−	−	−
S	+	−	+	+	−	−
IS	+	−	+	+	+	+
IX	+	−	−	+	+	−
SIX	+	−	−	+	−	−

Table 2. Compatibility matrix of the CC-lock

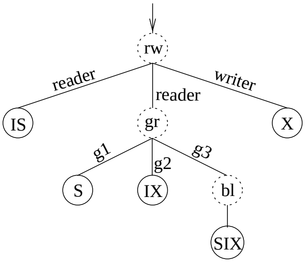

Figure 1. lock tree

5 A New Approach to the CC-Lock

As seen in the previous section, the standard implementation of the CC-lock tends to unnecessarily serialize lock requests. We now describe a new approach for the implementation of CC-locks, which does not serialize compatible lock requests. The main idea is to split the complex CC-lock into basic locks (section 2). A combination of three basic locks will then act as a CC-lock.

According to the compatibility matrix (Table 2), a requested X lock is incompatible to any other granted lock, including granted X locks, and vice versa. Nevertheless, multiple non-X locks may be granted concurrently (e.g., IX and IS locks). If we, in thought, combine the non-X lock modes of the compatibility matrix to one lock mode compatible to itself, we get the compatibility matrix of the R/W-lock. Thus, we can use a R/W-lock (rw) to detach non-X locks from X locks. There remains the problem of testing compatibility of the non-X locks.

According to the compatibility matrix of the four remaining lock modes S, IX, IS and SIX, a requested IS lock is compatible to any other granted lock. Vice versa, any requested lock is compatible to an already granted IS lock. Thus, an IS lock request is granted as soon as the r/w lock rw is succesfully obtained.

Consequently, we just have to look at the three remaining lock modes S, IX, SIX. Suppose that requested SIX locks would be compatible to granted SIX locks, then, the remaining compatibility matrix of these three modes would be a diagonal — the same matrix as of the group lock. We can therefore separate the three remaining modes from each other by using a 3-group lock (gr) and assigning a group number to each mode (e.g., S=g1, IX=g2, SIX=g3). The remaining problem is to serialize concurrent SIX locks. This can be done by using a binary lock (bl).

The described combination of basic locks yields a lock tree (Figure 1). The leaves are labeled with the lock modes of intentional locking. The inner nodes are labeled with basic locks, and the edges are labeled with lock modes of the basic locks. On a lock request, the thread goes from the root to the requested leaf, thereby acquiring the basic locks in the appropriate lock modes. In order to release a lock, the thread goes upward from the leaf to the root releasing the basic locks on the path.

Correctness Since the proof of correctness and fairness of the CC-lock is too long, it is omitted here, but it can be found in [5].

6 Comparisons and Conclusion

We have described a fast and highly concurrent lock manager. The basic idea was to split up the complex database lock structure (CC-lock) into three basic lock primitives. On the SB-PRAM, these lock primitives do not serialize compatible lock requests, and therefore the CC-lock does not serialize, either. Our implementation can easily be ported to any parallel architecture supporting the used lock primitives.

As measurments on the SB-PRAM showed, the new CC-lock has a run time of less than 70 instructions if no waits due to incompatibility to other locks are necessary. The total run time of a lock request thru the lock manager is between 400 and 600 instructions. On relation granularity, where mostly mutually compatible intention locks (IS/IX) are used, nearly all requests can be handled with 600 instructions. In contrast, due to serialization, the standard implementation needs 120K instructions if 2000 threads are running. This results in a speedup of factor 200 for locking on relation granularity for high speed shared memory systems, if our new implementation is used.

We currenctly implement of a rudementary database system on the SB-PRAM. For this, we also developed and implemented a serialization free deadlock detection algorithm [5]. Detailed results are expected for fall 1999.

Acknowledgments: The authors would like to thank Michael Bosch, Arno Formella, Silvia M. Mueller, and Jochen Preiss for their help and valuable discussions.

References

[1] G. Alverson, et al.: *The Tera Computer System*, Supercomputing, 1990.

[2] F. Abolhassan, J. Keller and W.J. Paul: *On the Cost-Effectiveness and Realization of the theoretical PRAM Model*, SFB-Report 09/91, March 1991.

[3] A. Gottlieb, et al.: *The NYU Ultracomputer — Designing an MIMD Shared-Memory Parallel Computer*, IEEE Trans. on Computers, pp. 175-189, Feb 1983.

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

[5] C. Jacobi: *A relational Database System for the SB-PRAM. TPC–B and Concurrency-Control*, Master Thesis (german), University of Saarland, 1999.

[6] L.D. Molesky and K. Ramamritham: *Efficient Locking for Shared Memory Database Systems*, University of Massachusetts Technical Report, March 1994.

[7] J. Roehrig: *Implementing P4 on the SB-PRAM*, Master Thesis (german), University of Saarland, 1996.

[8] B. Spengler: *A relational Database System for the SB-PRAM. Basic Components*, Master Thesis (german), University of Saarland, 1997.