A Per-object Based Hybrid Concurrency Control

Kwak, Tae-Yeong, Lee, Yoon-Joon, Kim, MyoungHo
CS department, Korea Advanced Institute of Science and Technology
{tykwak, yjlee, mhkim}@dbserver.kaist.ac.kr

Abstract

Existing concurrency control algorithms do not well conform to various environments, in the performance perspective. Each algorithm has some assumption on the conflict characteristic of its execution environment, and its performance degrades when the assumption fails.

A per-object hybrid scheme integrating two-phase locking algorithm with parallel validation technique is presented to solve this problem. Read and write accesses on each database object are controlled by one algorithm at a time. This controlling algorithm can alter during transactions are running, in order to enhance the overall system performance. In the simulation study, proposed scheme with the proposed algorithm alteration strategy is shown to well conform to various execution environments, better than both two-phase locking algorithm and parallel validation technique.

1. Introduction

For obtatining high performance, multiple transactions are running concurrently in most database systems. Such concurrent executions may bring inconsistency into the database, unless the interaction between transactions are carefully controlled. Serializable execution of transactions is proven to preserve the database consistency.

So far, several algorithms have been proposed in order to control concurrent transactions. The major assumption on transaction conflict classifies these algorithms into two groups. Pessimistic algorithms including 2-phase locking algorithm [7] assume that a transaction tends to conflict with others. Conversely, optimistic algorithms such as validation techniques [8] assume that conflicts between transactions seldom occur.

The characteristics of concurrency control algorithms have been evaluated in various ways [1,5,6,9]. It is widely accepted that potential deadlock and lock management overhead are major defects of 2-phase locking algorithm. It, however, performs better than validation techniques when transactions frequently conflict with others. Recent studies showed that the concurrency control performance relies heavily on the significance of resource utilization.

Database system is utilized for various appication area. Concurrency control algorithm embedded in a database system thus should be highly adaptable. In this respect, integration of several algorithms has been tried. Most proposals concentrate on using different algorithms per transaction [4,11,12]. The use of algorithms per data object was also suggested [10]. In [10], They pointed out that the per-transaction based strategies fail to sufficiently exploit the advantages of each basic algorithm. The idea proposed in [10] does not, however, give direct solution for the centralized databases. Furthermore, most of these previous works did not give any quantitative evaluation of the proposed algorithm's performance characteristics in various execution environments.

We propose a per-object based integration strategy that couples 2-phase locking algorithm and parallel validation technique. Generally, each database object has only one controlling algorithm at a time. The system dynamically determines the appropriate algorithm for each object to enhance performance. Transactions do not have to care the controlling algorithm for each object.

Transactions have their private workspaces [2]. They execute in four phases: the normal execution phase, two check phases, and the update phase between check phases. After doing all of its operations in the normal phase, every transaction checks its validity in check phases. If it fails to validate itself, it dies or kills other transactions to keep the consistency of database. Validated transactions transfer their execution results to the persistent database in their update phases.

The criterion of choosing algorithm for each object is that what algorithm would waste less transaction time. Every access on a data object causes the re-estimation of mean wasted time for a transaction in accessing that object. If the current algorithm is assumed to waste too much time, the other algorithm begins to control the accesses on that object. Statistical information such as the mean transaction execution time, the aborted transaction

0-8186-8147-0/97 $10.00

count, the mean lock duration time, and the blocked transaction count are maintained per each object for this purpose.

The remainder of the paper is organized as follows. Section 2 describes basic concurrency control algorithms used in this paper. Section 3 presents the proposed integration strategy. Correctness of our work is shown in section 4. Section 5 compares our proposed algorithm with the basic algorithms. Section 6 describes related works on the integration of heterogeneous algorithms. In section 7, we conclude with a discussion.

2. Basic Algorithms

Most of existing concurrency control algorithms deal with every operation in the same manner. We can classify these *homogeneous* algorithms by their assumption on transaction conflict. Pessimistic and optimistic algorithms are major classes in this classification. Algorithms in the former class assume that transactions are likely to conflict with each other, and that these conflicts are serious; they should be resolved immediately. Pessimistic algorithms block the execution of conflicting transactions or roll them back as soon as any conflict occurs, to prevent or avoid non-serializable executions. Conversely, optimistic algorithms assume that conflicts seldom occur. This class of algorithms does not consider conflicts seriously. In general, it lazily detects and recovers executions that are non-serializable; it verifies transactions' validity at the end of their execution, then terminates and restarts the transactions that cause conflicts. Well-known examples of each class are strict two-phase locking algorithm [7] and parallel validation technique [8], which we adopt as basic algorithms into our work.

Performance studies have been done to investigate the operational characteristics of each algorithm. Empirical study with real-life database reference strings has shown that optimistic algorithms outperform locking algorithms if the I/O costs can be neglected by large buffers [9]. Analytic evaluation of concurrency control algorithms has claimed that overhead of validation techniques is bigger than that of 2-phase locking algorithm [5]. Overheads of deadlock detection and resolution are neglected in this study, however. Thus, we can conclude that overheads of each algorithm are quite comparable, as assumed in [1]. Simulation studies [1,6] have shown a surprising result. When the conflict rate or the degree of concurrency is low, two-phase locking algorithm and validation techniques perform similarly. If the degree of concurrency becomes high, two-phase locking algorithm works better, when resource contention is significant. Validation techniques work better with plenty of resources.

3. Integration Strategy

Our approach to integrate strict two-phase locking with parallel validation technique is described here. The database system model, the data object access procedures, the serializability validation procedures, and the algorithm alteration procedures are presented in turn.

3.1 Database system model

A database consists of independent data objects. Thus, implicit accesses to a data object by accessing another are prohibited. If a transaction T accesses a data object x, T must contain some operation p[x] on x.

Each data object has a type which represents its control algorithm for the operations on that object. Allowed types are L and P that indicates strict 2-phase locking algorithm and parallel validation technique respectively. In general, Every data object has exactly one type at any given time.

Private workspace model [2] is used in our work; every transaction has its private workspace, and there are three types of operations: read, pre-write, and write.

read (x) returns the value of x.

pre-write (x, v) updates the value of x in the workspace to be v.

write (x) transfers the value of x from the workspace to the database.

Transactions run in four phases: the normal phase, the first check phase, the update phase, and the second check phase (Table 1). In the normal phase of a transaction T, all read and pre-write operations of T on P-typed data objects go with no restriction. The read-set and the write-set are maintained for T to keep track of the P-typed data objects T has accessed during its execution. T's operations on L-typed data objects generally follow 2-phase locking protocol; every read or pre-write operation on an L-typed data object x must be preceded by the appropriate lock acquisition, unless T has already accessed x under parallel validation. The lock table is maintained to manage locks.

The first check phase begins after all operations of T have done in the normal phase. In this phase, the validity of T with transactions in their update phases is verified. T aborts and restarts its execution if the validation fails; the update phase begins otherwise. In the update phase, the

Execution phases	Type of data object	
	L	P
normal	**lock, read, pre-write**	**read, pre-write**
1st check		Validation with transactions in their update phase
update	**write**	
2nd check	**unlock**	Validation with transactions in their normal phase

Table 1. Operations and procedures in each phases

```
function read ( x );
  if ( x is in read-set or in write-set or x is P-typed )
    then insert ( x, read-set );
    else lock ( x, shared );
  endif;
  if ( a copy of x exists in the private workspace )
    then return the value of the copy of x;
    else return the value of the original x;
  endif;
end;

procedure pre-write ( x, v );
  if ( x is in read-set or in write-set or x is P-typed )
    then insert ( x, write-set );
    else lock ( x, exclusive );
  endif;
  if ( no copy of x exists in the private workspace ) then
    create a copy of x;
  endif;
  value of the copy of x ← v;
end;
```

Figure 1. Read and pre-write

values of updated copies in T's workspace are transferred to the persistent database by write operations. In the last phase, which is the second check phase, the validity of T with transactions in their normal phases is verified. When the validation fails, all transactions that conflict with T are forced to restart. T completes its execution after the validation finishes.

3.2 Accessing Data Objects

The read and pre-write operations are shown in Figure 1. The lock operation is slightly modified so that '*what transaction is blocked when*' information is recorded for the object that the lock request is not granted.

During the unlock operation, this recorded information is used to update the mean lock duration. '*Wasted time by block*' information is recalculated then, to decide whether it needs to change the control algorithm of the data object.

The write operation only transfers the value of the data object to the persistent database from the local workspace.

3.3 Validation in Check Phases

As in the validation techniques, our scheme assigns the increasing *transaction number* to each transaction when it starts its first check phase. For each transaction T_j and for all transaction T_is that have smaller transaction numbers than that of T_j, one of the following conditions must hold:

1. T_i completes its update phase before T_j starts its normal phase.
2. T_i's write-set does not intersect T_j's read-set, and T_i completes its update phase before T_j starts its update phase.
3. T_i's write-set does not intersect both T_j's read-set and write-set, and T_i completes its normal phase before T_j completes its normal phase.

For concurrently running transactions, the condition 1 does not hold. Thus, the condition 2 or 3 must hold for them. In the first check phase, the condition 3 is verified; the condition 2 is verified in the second check phase.

Check phases are synchronized; transactions should enter a critical section to perform each check phase.

When the validation fails in any check phase, restarts of one or more transactions occur. Transactions do book-keeping jobs before they abort; for each data object that cause a restart, '*when transaction aborts*' information is recorded. '*Wasted time by abort*' information is again calculated to determine whether it needs to change the control algorithm.

3.4 Control Algorithm Alteration Strategy

```
procedure first-check-phase ( T );
  valid ← true;
  conflict-set ← null;
  for each transaction U that is in its update phase do
    rs ← intersection ( read-set of T, write-set of U );
    ws ← intersection ( write-set of T, write-set of U );
    if ( either rs or ws is not null ) then
      valid ← false;
      conflict-set ← conflict-set ∪ rs ∪ ws;
    endif;
  done;
  if ( valid is false ) then
    for each data object x in conflict-set do
      record transaction abort time for x;
      recalculate the wasted time by abort;
      make decision of algorithm change;
    done;
    unlock all locks possessed by T;
    T aborts;
  endif;
end;

procedure second-check-phase ( T );
  for each transaction U that is in its normal phase do
    valid ← true;
    rs ← intersection ( read-set of U, write-set of T );
    if ( rs is not null ) then
      valid ← false;
      for each data object x in rs do
        record transaction abort time for x;
        recalculate the wasted time by abort;
        make decision of algorithm change;
      done;
    endif;
    if ( valid is false ) then
      unlock all locks possessed by U;
      U aborts;
    endif;
  done;
  unlock all locks possessed by T;
  update mean transaction execution time;
  T commits;
end;
```

Figure 2. Check phases

```
procedure change-from-L-to-P ( x );
  for each transaction T which possesses the lock on x do
    case lock mode of x of
      shared:      insert ( x, read-set of T );
      exclusive:   insert ( x, read-set of T );
                   insert ( x, write-set of T );
    end;
  done;
  lock mode of x ← none;
  clear the information of x about lock-possessing T's;
  type of x ← P;
end;

procedure change-from-P-to-L ( x );
  clear the information of x about lock-possessing T's;
  for each transaction T which have accessed x do
    give T an exclusive lock on x;
  done;
  lock mode of x ← exclusive;
  type of x ← L;
end;
```

Figure 3. Algorithm alteration procedures

Concurrency control algorithms degrade the system throughput by wasting execution time of transactions. For example, transactions waste their execution time when they are blocked under strict 2-phase locking algorithm, while transactions waste their execution time when they are aborted under validation techniques. We adopt these wasted time measures to evaluate the system status and to select an appropriate algorithm for each data object.

Let τ_L be the wasted time by strict 2-phase locking, and τ_P be the wasted time by parallel validation technique. Then, these values can be estimated like this:

$\tau_L \cong$ mean lock duration × # blocked T's

$\tau_P \cong$ mean transaction execution × # aborted T's

If both values are known, it is quite natural to select the control algorithm for a data object by comparing these two measures. However, it is impossible to know the both simultaneously, as only one algorithm works at a time.

Thus, some threshold value is used in the selection of algorithm. When the magnitude of the measure currently considered exceeds a predefined threshold value, control algorithm change occurs.

Algorithm alteration procedures are shown in Figure 3. *Note that* information for data objects already accessed under the parallel validation technique, contained in read-set and write-set of transactions, remains unchanged after the algorithm change. This trick allows transactions see the control algorithm for each data object invariantly.

4. Correctness of the proposed strategy

We describe the proof that our scheme guarantees the serializable transaction execution. The proof is based on the serialization graph (SG) for an execution history. It is proven that a history H is serializable if and only if its serialization graph SG(H) is acyclic [3]. We present a descriptive proof.

Lemma 1. Let H be an execution history of committed transactions generated by our algorithm, and suppose that $T_i \rightarrow T_j$ is in SG(H). Then, the first check phase of T_i precedes the first check phase of T_j.

Proof: Since $T_i \rightarrow T_j$, there must exist conflicting operations $p[x] \in T_i$ and $q[x] \in T_j$ respectively, such that $p[x]$ precedes $q[x]$. There are 4 different cases.

Case 1: Suppose that $p[x]$ and $q[x]$ are all under strict 2-phase locking algorithm. As under the 2-phase locking algorithm, the unlock on x in T_i precedes the lock on x in T_j. As the first check phase of every transaction precedes all unlock operations and succeeds all lock operations, the first check phase of T_i precedes that of T_j.

Case 2: Suppose that $p[x]$ and $q[x]$ are under parallel validation technique. Assume, by way of contradiction, that the first check phase of T_j precedes the first check phase of T_i. Then, the transaction number of T_j precedes that of T_i, and one of the following conditions must hold:

1. T_j completes its execution before T_i begins.
2. T_j's write-set does not intersect T_i's read-set, and T_j's all operations precede T_i's write operations.
3. T_j's write-set does not intersect both T_i's read-set and write-set, and T_j's read operations precede T_i's write operations.

This means that, for conflicting operations $s \in T_j$ and $t \in T_i$, s must precede t. It contradicts that $p[x]$ precedes $q[x]$. Thus, the first check phase of T_i precedes that of T_j.

Case 3: Suppose that $q[x]$ is under parallel validation technique, and that $p[x]$ is under strict 2-phase locking algorithm. If T_i and T_j are concurrent, this case cannot happen. Thus, T_i should precede T_j, resulting that the first check phase of T_i precedes that of T_j.

Case 4: Suppose that $p[x]$ is under parallel validation technique, and that $q[x]$ is under strict 2-phase locking algorithm. The lock on x in T_j is preceded by the end of T_i. Then, the first check phase of T_i precedes the lock on x in T_j, which precedes the first check phase of T_j.

Lemma 2. Let H be an execution history of committed transactions generated by our algorithm, and let $T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n$ be a path in SG(H), where $n > 1$. Then the first check phase of T_1 precedes the first check phase of T_n.

Proof: The proof is by induction on n. The basis step of induction, for $n = 2$, follows immediately from Lemma 1. For the induction step, suppose the given lemma holds for $n = k$ for some $k > 2$. We will show that it holds for $n = k + 1$. By the induction hypothesis, the path $T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_k$ implies that the first check phase of T_1 precedes the first check phase of T_k. But, by $T_k \rightarrow T_{k+1}$ and Lemma 1, the first check phase of T_k precedes the first check phase of T_{k+1}. Thus, the first check phase of T_1 precedes the first

check phase of T_{k+1}, as desired.

Theorem. Every history H generated by our algorithm is serializable.

Proof: Suppose that SG(H) contains a cycle $T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n \rightarrow T_1$, where $n > 1$. By Lemma 2, the first check phase of T_1 should precedes the first check phase of itself, which is impossible. SG(H) thus has no cycles and so H is serializable.

5. Performance

This section describes the evaluation of the proposed scheme versus the basic algorithms, strict 2-phase locking algorithm and parallel validation technique. We adopted the performance model proposed in [1]. Simulations are performed in varying the number of resources reflecting the environmental variety.

Our simulation parameters reflect that the database contains 1000 data objects, that transactions are issued from 200 terminals, that the transaction inter-arrival time follows the exponential distribution with mean = 5 sec., that transactions may read 4 ~ 20 data objects and update 20 ~ 30% of them, and that disk I/O time for a data object is 16 ms, while CPU time for a data object is 2 ms. The size of database is taken to be small enough to observe frequent conflicts. As in [1], the processing overhead of algorithm itself is neglected.

Our algorithm with threshold equal to three times of the mean transaction execution time is evaluated. Strict 2-phase locking algorithm and parallel validation technique are also evaluated for comparison. Initially, data objects are all L-typed in our scheme.

In order to vary the execution environment, we change the degree of concurrency and the number of resources. Considered degrees are 5, 10, 25, 50, 75, 100, 150, and 200. We assume that there are N CPUs and 2N disks, and vary N from 1 to 8 by doubling it. The throughput is used as the performance metric. Each simulation consists of 20 batches, each of which is a run of 50 seconds succeeding a run of 20 seconds. Thus, the total simulation time is 1000 seconds.

Figure 4 shows the simulation result with N = 1. For

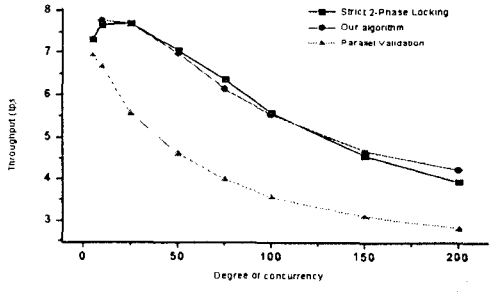

Figure 4. Throughput with 1 CPU and 2 disks

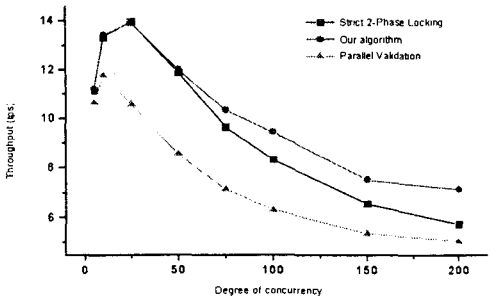

Figure 5. Throughput with 2 CPUs and 4 disks

each algorithm, the throughput curve indicates thrashing; as the degree of concurrency increases, the throughput increases first, then reaches a peak, then finally degrades.

Strict 2-phase locking algorithm outperforms parallel validation technique in this case. The performance of our scheme is quite similar to that of strict 2-phase locking algorithm, as the algorithm change seldom occurs.

When the system has 2 CPUs and 4 disks, the behavior of each algorithm is fairly similar to that in the previous case (Figure 5). The penalty of the resource contention is reduced, however. Our scheme slightly outperforms strict 2-phase locking algorithm as the degree of concurrency increases, verifying that dynamic allocation of algorithm works well.

Figure 6 shows the throughput curves with N = 4 and N = 8. The resource contention is much lowered. Parallel validation technique outperforms strict 2-phase locking as the penalty of wasting resource is lighten. Our scheme works remarkably better than both the basic algorithms. While the parallel validation technique is a better strategy globally, strict 2-phase locking algorithm may outperform for some data objects. The simulation result verifies this expectation sufficiently.

6. Related Works

The need for a highly adaptable concurrency control algorithm has driven several studies on the integration. Several r-w and w-w synchronization techniques were integrated in centralized database systems [4]. Two-phase locking algorithm and serial validation technique were merged in distributed database environment [11]; every transaction is typed either locking or optimistic. These

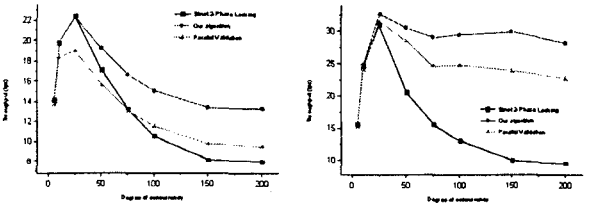

Figure 6. Throughput with N = 4 and N = 8

works concentrated on integration and correctness.

The concept of per-object heterogeneity was introduced in [10]. Data objects are typed, as well as transactions, either locking or optimistic. While the concepts are quite similar, this work utilizes time interval, which is useful in distributed database systems, but not useful in centralized database environment.

A modification of two-phase locking algorithm with optimistic philosophy was proposed in [12]. Under this scheme, transactions can access some predefined amount of data objects without locks, but it needs to acquire locks for accessed objects, in order to access more objects.

7. Concluding Remarks

Our goal for this work was to propose a concurrency control scheme based on strict 2-phase locking algorithm and parallel validation technique. In order to satisfy our goal, we proposed per-object based algorithm integration strategy.

The control algorithm for each object can change when transactions are running, but the change is hidden to running transactions for the synchronization purpose. This enables the type of each object to be transaction-consistent; every transaction considers that data objects are static-typed.

The key idea of integrating both algorithms is to assign the transaction number when all of the necessary locks are acquired. The transaction number problem with parallel validation technique [8] was also solved in our scheme by separating validation job into two phases.

The rate of conflict is not a good measure to estimate the system state. As we have seen, both algorithms show the similar behavior when the rate of conflict is low. We have developed a new measure based on the wasted time: each algorithm waste some transactions' execution time in synchronizing the running transactions. The control algorithm for a data object changes when the wasted transaction time in accessing that object exceeds some threshold value. The evaluation we have performed with this measure shows that our measure works fine. Proposed algorithm outperforms two basic algorithms we have used.

Algorithm change procedure and the storage overhead are major defects of our scheme. Using lock table instead of read-set and write-set may reduce both the overhead of algorithm changing and storage overhead.

Additional information essential to maintain the type of data objects dynamically is also a large space overhead. The number of data objects in real-life databases reaches to near several millions or more, and it would be difficult to maintain only types in main memory. Maintaining only the types of data objects which are currently being used is a feasible solution to this problem. Newly accessed data object can be treated by using a default type.

Only the recent conflicts are considered in our scheme, in order to keep up quickly with the change of the conflict tendency. It is not clear how to define the recentness, but it seems fair to let the recentness interval be the multiple of mean transaction execution time, since transactions are sensitive only to the events during their life time.

References

[1] R. Agrawal, M. J. Carey, and M. Livny, "Concurrency Control Performance Modeling: Alternatives and Implications," ACM Trans. on Database Systems, vol. 12, no. 4, 1987, pp. 609-654.

[2] P. A. Bernstein, and N. Goodman, "Concurrency Control in Distributed Database Systems," ACM Computing Surveys, vol. 13, no. 2, 1981, pp. 185-221.

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

[4] H. Boral, and I. Gold, "Towards A Self-Adapting Centralized Concurrency Control Algorithm," ACM SIGMOD Record, vol. 14, no. 2, 1984, pp. 18-32.

[5] M. J. Carey, "An Abstract Model of Database Concurrency Control Algorithms," ACM SIGMOD Record, vol. 13, no. 4, 1983, pp. 97-107.

[6] M. J. Carey, and M.R. Stonebraker, "The Performance of Concurrency Control Algorithms for Database Management Systems," in Proc. 10th Conf. on Very Large Data Bases, 1984, pp. 107-118.

[7] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger, "The Notions of Consistency and Predicate Locks in a Database System," Commun. of ACM, vol. 19, no. 11, 1976, pp. 624-633.

[8] H. T. Kung, and J. T. Robinson, "On Optimistic Methods for Concurrency Control," ACM Trans. on Database Systems, vol. 6, no. 2, 1981, pp. 213-226.

[9] P. Peinl, and A. Reuter, "Empirical Comparison of Database Concurrency Control Schemes," in Proc. 9th Conf. on Very Large Data Bases, 1983, pp. 97-108.

[10] J. F. Pons, and J. F. Vilarem, "Mixed concurrency control: Dealing with heterogeneity in distributed database systems, " in Proc. 14th Conf. on Very Large Data Bases, 1988, pp. 445-456.

[11] A. P. Sheth, and M. T. Liu, "Integrating Locking and Optimistic Concurrency Control in Distributed Database Systems", in Proc. 6th Int. Conf. on Distributed Computing Systems, 1986, pp. 89-99.

[12] P. S. Yu, and D. M. Dias, "Concurrency Control Using Locking with Deferred Blocking," in Proc. 16th Conf. on Very Large Data Bases, 1990, pp. 30-36.