On Using Priority Inheritance-Based Distributed Static Two-Phase Locking Protocol

Sarvesh Pandey and Udai Shanker

Abstract Two-phase locking with high priority (2PL-HP), a well-suited concurrency control protocol for distributed real-time database systems (DRTDBS) because of being free from priority inversion problem, is used for accessing data items to resolve conflicts among the concurrently executing transactions. However, it suffers from the problems of wastage of system resources responsible for degrading the system performance. In DRTDBS, our basic aim is to minimize the number of transactions missing their deadline. In this paper, static two-phase locking with priority inheritance (S2PL-PI) protocol has been proposed specifically to minimize the wasted system resources, i.e., CPU and data items by avoiding unnecessary abort of transactions by optimal use of priority inheritance mechanism. A DRTDBS is simulated for comparison of the performance of S2PL-PI protocol with previous other protocols, and results confirm the significant improvement in system performance.

Keywords Concurrency control · Two-phase locking · 2PL-HP · Priority inheritance · Distributed real-time database

1 Introduction

Database systems (DBS) are a collection of logically interrelated data items shared by multiple users [1, 2]. A user can interact with databases by means of a partially ordered set of read and write actions termed as a transaction. Every transaction follows ACID (Atomicity, Consistency, Isolation, and Durability) property [3, 4]. Isolation is one of the essential transaction properties. The role of isolation comes into play when more than one transactions concurrently execute in the database using

S. Pandey (✉) · U. Shanker
Department of Computer Science and Engineering, MMM University of Technology, Gorakhpur, UP, India
e-mail: pandeysarvesh100@gmail.com

U. Shanker
e-mail: udaigkp@gmail.com

M. L. Kolhe et al. (eds.), *Advances in Data and Information Sciences*, Lecture Notes in Networks and Systems 38, https://doi.org/10.1007/978-981-10-8360-0_17

critical resources [5]. To ensure the isolation among concurrently executing transactions, the system must govern the interaction among them by having a serializable schedule. Concurrency control (CC) schemes ensure the serializability among concurrently executing transactions [6]. Results obtained from concurrently executing transactions can be reflected in the database only when there exists an equivalent serial execution schedule of such concurrently executing transactions. CC schemes are broadly classified as pessimistic and optimistic. In pessimistic CC scheme, conflict is detected before access of data item, while in optimistic CC scheme, conflict is detected after the access of data item. After detection of conflict among a set of concurrently executing transactions, a conflict resolution mechanism comes into play. Conflict resolution mechanism specifically does the following.

1. Select transaction(s) from the set of conflicting transactions to prosecute.
2. Take an appropriate action against selected transaction(s) at a suitable time.

The two most used actions are blocking and abort. If a conflict is detected before data item access, one of both actions can be taken. However, abort action is appropriate in case of conflict detected after data item access. Several CC schemes have been proposed to overcome the problem of inconsistency [6, 7]. Most of the techniques used to facilitate concurrent execution of transactions rely on the notion of locking of data items. The commercial conventional DBS uses two-phase locking (2PL) [8] as a CC scheme. They use locking of data items to ensure isolation among concurrently executing transactions and thereby guaranteeing serializability. Every data item in a DRTDBS is associated with a variable that describes the possible operations that can be performed on it. 2PL protocol says that all locking operations on a transaction precede the first unlock operation. In 2PL, every transaction obtains a lock before accessing any data item. Transaction execution divides into two phases: growing phase and shrinking phase [9]. A transaction can acquire a lock on data items during growing phase, but any of the locks that are acquired during this phase cannot be released. In shrinking phase, a transaction can release all the locks acquired during growing phase, but cannot acquire any new lock.

Two widely known 2PL variants available in the literature are static 2PL and dynamic 2PL. In static 2PL, the data item access list, i.e., locks required by a transaction are presumed to be granted prior to the start of its execution [10]. The transaction locks all needed data items before it begins its execution. Here, a data item access list is predeclared before execution of any transaction. Note that a data item access list consists of a set of data items required to be granted access for completion of a transaction. If any of the predeclared data items of the data item access list cannot be granted an access, then the transaction does not lock any of the data items; however, it waits until all data items are available for locking. In dynamic 2PL, transactions obtain locks to access data items on request and release locks upon expiry or commit. The working principle of static 2PL is like dynamic 2PL except the technique of setting locks on data items. In general, static 2PL requires smaller number of messages for setting locks as compared to dynamic 2PL. Static 2PL does not suffer from deadlock since blocked transactions cannot hold a lock on any data item.

In non-real-time static 2PL, a transaction gets blocked if any of its required lock(s) from the predeclared data item access list is/are locked by some other transactions. At the time of transaction blocks, some of the data items required by it may be free. Such data items that were free at the time when conflict occurred can be seized by other transactions later. As a result, even after the original conflicting data items are released, the transaction may get blocked by other transactions arrived after it. Consequently, the requesting transaction blocking time can be randomly long because of extended blocking which is a result of waiting for more than one locks. Real-time S2PL (RT-S2PL) protocol [11] overcomes this problem. Here, each data item in the database is assigned a priority equal to the priority of highest priority transaction from the set of transactions that have requested access to this data item. Like S2PL, all the data items required to be accessed by a transaction need to be locked before starting execution of a transaction. If a conflict occurs for any of the data items, none of the required locks will be assigned to a requesting transaction. Note that in case a data item requested by a transaction has lower priority than that of the transaction itself, its priority will be updated to that of the requesting transaction. All such features of RT-S2PL lead to its suitability for DRTDBS.

The 2PL wait promote (2PL-WP) protocol [12, 13] is identical to 2PL in its resolution of conflicts. As in 2PL, 2PL-WP resolves the conflict (if any) by means of blocking a requesting transaction. It is the first concurrency control protocol based on priority inheritance mechanism to reduce the negative impact of priority inversion problem [14], which is inherent in a real-time environment. In case of priority inversion, low-priority lock-holding transaction inherits the priority of the highest priority transaction from the set of transaction requested access to the data item. Further, lock-holding transaction retains this inherited priority until it either commits or restarted. Priority inheritance mechanism specifically reduces the priority inversion duration so that requesting high-priority transactions can get the conflicting resource earlier [15, 16].

The 2PL high-priority (2PL-HP) protocol [12, 13] ensures that low-priority transactions do not delay high-priority transaction by eliminating priority inversion problem. It does so by resolving data conflicts instantly in favor of the transaction with higher priority. 2PL-HP concurrency control protocol suffers from the problem of the cyclic restart. Just like 2PL-HP, the S2PL high-priority (S2PL-HP) protocol also [1] does not suffer from priority inversion problem. It does so by resolving data conflicts instantly in favor of the transaction with the higher priority. Lengthy transactions suffer from the starvation problem due to use of S2PL-HP. One more severe problem with S2PL-HP protocol is that it may lead to undesirable wastage of resources due to ABORT of low-priority lock-holding transactions in case of conflict with high-priority transaction. This may lead to the increase in a number of transactions missing their deadline. S2PL-HP concurrency control protocol suffers from the problem of the cyclic restart. Let us consider an example to explain this problem.

Suppose, at time t_1, a distributed real-time transaction T_1 starts its execution at site S. Coordinator of transaction T_1 divides it into a set of subtransactions as $T_1 = \{T_{11}, T_{12}, T_{13}, \ldots, T_{1N}\}$, where N is the number of cohorts participating to complete T_1. All these subtransactions execute at different sites in the database system. At time

t_2, subtransaction T_{12} is in the processing phase and completed locking phase. The data item O is locked by T_{12}, during its locking phase. Note that execution period consists of locking phase and processing phase, respectively. This same data item O is further required by some other transaction T_2 that enters in the system. Note that priority of T_2 is greater than the priority of T_1. So, as per S2PL-HP, the transaction T_1 is aborted (or restarted), and the data item O will be given to T_2. Note that all the subtransactions participating to complete the execution of T_1 will be restarted, no matter conflict arrives only at the site where subtransaction T_{12} is running. There is a possibility that after a restart of transaction T_1, its priority may become higher than the priority of T_2 as a result of decrease in slack time, which directly affects the deadline of the transaction. In such case, transaction T_2 is aborted (or restarted) and transaction T_1 will again lock the data item O. This leads to the cyclic restart problem among transactions (transactions T_1 and T_2). Transactions involved in cyclic restart miss their deadline at the end, which in turn leads to the wastage of resources and degradation in performance of the system in terms of increase in a number of transactions missing their deadline.

Based on S2PL, S2PL-HP, and 2PL-WP, static two-phase locking with priority inheritance (S2PL-PI) protocol has been proposed which optimistically uses the priority inheritance mechanism to minimum resource wastage. It also improves a chance of the low-priority transaction to get completed even after the conflict with some high-priority transaction provided that high-priority transaction can manage to wait for completion of low-priority transaction. S2PL-PI also overcomes the starvation problem with lengthy transactions up to some extent.

Section 2 discusses proposed S2PL-PI concurrency control protocol in detail. The performance study presented in Sect. 3 shows significant performance improvement in S2PL-PI protocol over other protocols. Finally, in Sect. 4, conclusions are drawn with future directions of works.

2 S2PL-PI: A Real-Time Concurrency Control Protocol

The lifetime of a cohort is divided into two phases, i.e., execution phase and commit phase. In SWIFT protocol, execution phase is further divided into two phases, i.e., locking phase and processing phase [17]. During locking phase, the cohorts lock all the required data items, and then, during processing phase, the cohort does some necessary computation. WORKSTARTED message to the coordinator is sent before the start of processing phase. If there are dependencies, then the sending of WORKDONE message is deferred till the removal of dependencies. S2PL-HP is used as a concurrency control protocol in SWIFT protocol which is a combination of S2PL and 2PL-HP. Further, it is suggested that a focused and coordinated research work is required to develop a new concurrency control protocol which directly affects the performance of SWIFT and all the existing commit protocols.

In case a high-priority distributed transaction T_{H} requests access to the data item that is already locked by some low-priority lock-holding transaction T_{L}, then con-

flict resolution strategy is used to resolve such conflict that affects the system's performance. Although it is clear that minimum the wastage of resources (CPU and data items) because of ABORT, maximum the chance of successful completion of transaction, but at the same time, there is need to ensure that the concurrency control algorithm is more focused toward respecting priority of transaction rather than minimizing wastage of resources or increasing throughput of the system. Hence, a new S2PL-PI protocol has been proposed which optimistically minimizes wastage of resources, reduces the starvation problem, and in turn minimizes the number of transactions missing their deadline. In S2PL-PI protocol, the ABORT of lock-holding cohort is done based on the intermediate priority of all the cohorts of a lock-holding transaction T_{L} and the priority of lock-requesting transaction T_{H}. It is a temporary priority assignment policy without affecting the initial priorities [18] and is based on the remaining execution time (T_{Remain}) needed by the lock-holding low-priority cohorts ($T_{\mathrm{L1}}, T_{\mathrm{L2}},\ldots, T_{\mathrm{L}i}$) and the slack time available with the newly arrived higher-priority cohort (T_{H}). It also solves the problem of starvation of long cohorts that arises due to the high probability of access conflicts. The slack time is the amount of time the distributed transaction can afford to wait in order to complete before its deadline. The remaining execution time of the lock-holding cohorts ($T_{\mathrm{L1}}, T_{\mathrm{L2}},\ldots, \mathrm{T}_{\mathrm{L}i}$) is given as:

$$T_{\mathrm{Remain}} = R_i - T_{\mathrm{Elapse}}$$

where R_i is the minimum transaction response time; T_{Remain} is remaining execution time needed by T_{L}; T_{Elapse} is elapsed execution time of T_{L}.

There are three ways to minimize the number of transactions missing their deadline.

1. If T_{L} is in the execution phase and $\mathrm{Max}T_{\mathrm{Remain}}(T_{\mathrm{L1}}, T_{\mathrm{L2}},\ldots, T_{\mathrm{L}i})$ is less than the slack time (T_{H}), then the priority of T_{L} gets inherited to T_{H}, and T_{H} is inserted into the wait queue.
2. If T_{L} is in the execution phase, $\mathrm{Max}T_{\mathrm{Remain}}(T_{\mathrm{L1}}, T_{\mathrm{L2}},\ldots, T_{\mathrm{L}i})$ is greater than or equal to the slack time (T_{H}), and T_{L}, then the low-priority lock-holding transaction gets aborted.
3. If $T_{\mathrm{L}i}$ is in commit phase (have sent a PREPARED message to its coordinator), then the priority of T_{L} gets inherited to T_{H}, and T_{H} is inserted into wait queue, no matter the requesting transaction T_{H} is a high-priority transaction. Although, the priority inheritance applied here, gives the conflicting transactions a chance of successful completion by reducing the priority inversion duration.

In brief, S2PL-PI protocol optimistically minimizes the wastage of resources by using priority inheritance mechanism and overcomes the starvation problem with lengthy transactions up to some extent. The following algorithm shows how the locks are granted in S2PL-PI.

2.1 S2PL-PI Algorithm to Resolve Data Conflict

Input: T_H is a high priority transaction requesting access to the data item O.
T_L is a low priority transaction that has locked the data item O.
T_{L1}, T_{L2}, ..., T_{Li} are the subtransactions of a transaction T_L

```
BEGIN
S2PL-PI_lock_acquire ()
{
for each Data item, di
if (! lockConflict)
        assign a  data item to T_H;
    else
   {
   check priority of conflicting cohorts  of  T_L holding the data item;
  if (priority (T_H)  > MaxPriority (T_L1, T_L2, ...., T_Li))
            {
            if (conflicting cohort T_Li haven't sent PREPARED message to its
            coordinator, and is in processing phase of execution period)
                    {
                    if (slack time(T_H)  ≥ MaxT_Remain (T_L1, T_L2, ...., T_Li))
                    //*slack time(T_H)  ≥ 0 *//
                            {
                                    insert T_H in wait queue;
                                    T_L inherits priority of T_H;
                            }
                            else
                            {
                                    T_H aborts T_Li;
                                    allocate data item to T_H;
                             }
                    }
            else
                    {
                                    insert T_H in wait queue;
                                    T_L inherits priority of T_H;
                    }
                }
   else T_H waits for completion of T_L;
   }
}
END
```

2.2 Major Contributions

The major contributions of S2PL-PI protocol are as follows.

1. Minimization of number of Aborts optimistically by use of priority inheritance scheme, and intermediate priority assignment policy (based slack time of lock-requesting high-priority transaction and remaining execution time of lock-holding low-priority transaction), which in turn minimizes the number of transactions missing their deadline.
2. Overcomes the starvation problem with lengthy transactions up to some extent using the intermediate priority assignment policy. Such problem arises due to the high probability of access conflicts in case of lengthy transactions.

3 Performance Evaluation

In the DRTDBS research community, there is no hands-on benchmark available to assess the performance of the proposed protocol. Therefore, a DRTDBS including N sites was simulated in accordance with the environment assumed in earlier studies [17, 19–21]. We ensured a significant level of resource and data contention during performance study. Table 1 presents different parameters used in a simulation study with their default values.

Parameter	Meaning	Default setting
DB_{Size}	Size of database (no. of pages in databases)	200 data objects/site
N_{db}	No. of database sites	4
AR	Transaction arrival rate per site	0–4 transactions/sec (uniformly distributed)
T_{com}	Communication delay among transactions	Either 1 ms or 100 ms
N_{op}	No. of operations in transaction	4–20 (uniformly distributed)
SF	Transaction slack factor	1–4 (uniformly distributed)
$P(w)$	Probability of write operation	0.60
CPU_{page}	Processing time required for accessing CPU page	5 ms
$Disk_{page}$	Processing time required for accessing disk page	20 ms

Earliest deadline first (EDF) is used as the cohort's priority assignment policy for the performance study of S2PL-PI protocol. As per EDF, a transaction with the closest deadline is assigned the highest priority in the system. In case of a tie, we

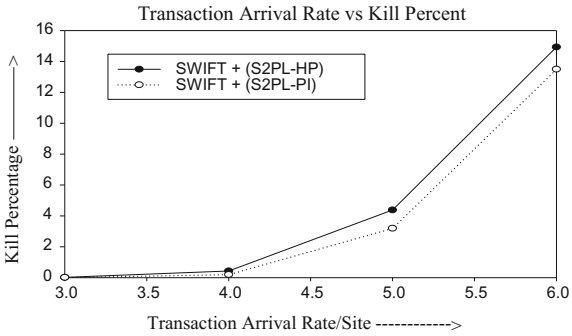

Fig. 1 Transaction kill percentage with resource and data contention at 1 ms communication delay under normal load

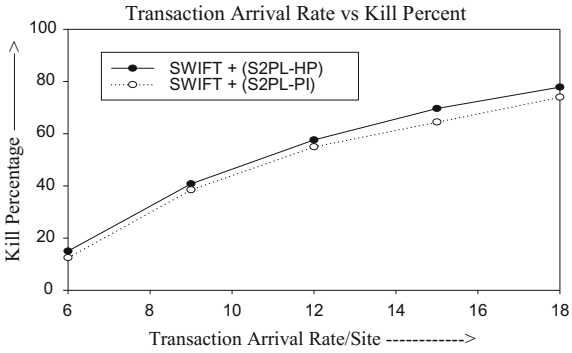

Fig. 2 Transaction kill percentage with resource and data contention at 1 ms communication delay under heavy load

assign priority to the transaction using FCFS scheme. The performance of S2PL-PI protocol is measured based on the number of transactions missing their deadline. Mathematical calculation of kill percent is done as the following,

$$\text{Kill Percent} = \frac{\text{Number of transactions aborted}}{\text{Total number of transactions in the system}}$$

3.1 Simulation Study and Performance Results

To investigate the performance of the S2PL-PI when applied to a DRTDBS, a wide range of operations ($N_{op} = 4$–20) in global as well as local transactions are introduced. The simulation study is performed with disk-resident databases. We compared the S2PL-PI concurrency control protocol with S2PL-HP protocol. Figures 1, 2, and 3 show the transaction kill percent at communication delay of either 1 ms or 100 ms in disk-resident databases with different transaction arrival rates.

The proposed protocol performs better than SWIFT protocol+S2PL-HP protocol under all load conditions. This is because of avoidance of ABORT of lock-holding transaction wherever possible by using priority inheritance scheme.

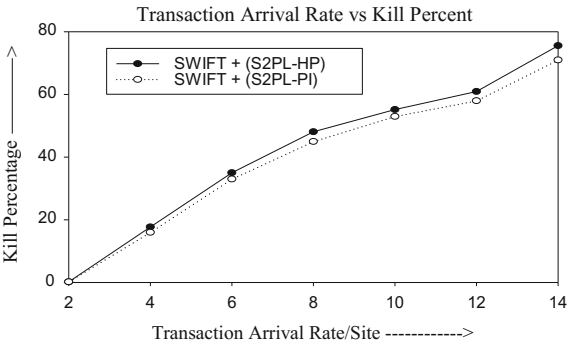

Fig. 3 Transaction kill percentage with resource and data contention at 100 ms communication delay under normal and heavy load

4 Conclusions

In this paper, the S2PL-PI protocol has been proposed to minimize the unnecessary abort of the transaction via the optimal use of priority inheritance mechanism. Here, an intermediate priority assignment policy has been introduced to assign an intermediate priority to the conflicting transactions at the time of data contention between them. This policy avoids the wastage of resources such as CPU and data item by not aborting a near to completion lock-holding transaction provided that a high-priority lock-requesting transaction can wait for the conflicting data item without missing its deadline or at least one of the lock-holding conflicting cohort is in PREPARED state. In this way, cooperative execution of conflicting transactions may lead to successful completion of all the competing transactions. It provided performance benefits over SWIFT protocol+S2PL-HP by optimum use of priority inheritance scheme and combination of initial and intermediate priority assignment strategies. A DRTDBS is simulated for the comparison of the performance of S2PL-PI protocol with previous other protocols, and results confirm the significant improvement in system performance.

As a part of future work, an exhaustive real-life implementation work is required to establish this approach as a value-based commercial product.

Acknowledgements We acknowledge the financial support provided by the Council of Scientific and Industrial Research (CSIR), New Delhi, India under grant no 1061461137 during this research work.

References

1. Shanker U, Misra M, Sarje AK (2008) Distributed real time database systems: background and literature review. Int J Distrib Parallel Databases 23(02):127–149
2. Shanker U, Misra M, Sarje AK (2001) Hard real-time distributed database systems: future directions. IIT Roorkee, India, pp 172–177

3. Pandey S, Shanker U (2016) Transaction execution in distributed real-time database systems. In: Proceedings of the international conference on innovations in information embedded and communication systems, pp 96–100
4. Ramamritham K (1993) Real-time databases. Distrib Parallel Databases 01(02):199–226
5. Faleiro JM, Abadi DJ (2015) FIT: a distributed database performance tradeoff. Data Eng 38(01):10–17
6. Yu PS, Wu K-L, Lin K-J, Son SH (1994) On real-time databases: concurrency control and scheduling. Proc IEEE 82(01):140–157
7. Kao B, Garcia-Molina H (1993) An overview of real-time database systems. Real Time Comput 127:261–282
8. Faleiro JM, Abadi DJ (2014) Rethinking serializable multiversion concurrency control. VLDB 08(11):1190–1201
9. Harding R, Aken DV, Pavlo A, Stonebraker M (2016) An evaluation of distributed concurrency control. VLDB 10(05):553–564
10. Lam KY (1994) Concurrency control in distributed real time database systems. Ph.D. thesis
11. Lam K-Y, Hung S-L, Son SH (1997) On using real-time static locking protocols for distributed real-time databases. Real-Time Syst 13(02):141–166
12. Abbott RK, Molina HG (1992) Scheduling real-time transactions: a performance evaluation. ACM Trans. Database Syst 17(03):513–560
13. Haritsa JR, Carey MJ, Livny M (1992) Data access scheduling in firm real-time database systems. Real-Time Syst 04(03):203–241
14. Pandey S, Shanker U (2018) A one phase priority inheritance commit protocol. In: Proceedings of the 14th international conference on distributed computing and information technology (ICDCIT), Bhubaneshwar, India, 11–13 Jan 2018 (Accepted)
15. Huang J, Stankovic JA, Towsley D (1991) On using priority inheritance in real-time databases. In: Real-time systems symposium, pp 210–221
16. Huang J, Stankovic JA, Ramamritham K, Towsley D, Purimetla B (1992) Priority inheritance in soft real-time databases. Real-Time Systems, vol 04, no 03, pp 243–278
17. Shanker U, Misra M, Sarje AK (2006) SWIFT—a new real time commit protocol. Distrib Parallel Databases 20(01):29–56
18. Shanker U, Misra M, Sarje AK (2005) Priority assignment heuristic to cohorts executing in parallel. In: 9th international conference on world scientific and engineering academy and society (WSEAS)
19. Lee VCS, Lam KW, Hung SL (2002) Concurrency control for mixed transactions in real-time databases. IEEE Trans Comput 51(07):821–834
20. Ulusoy O (1995) A study of two transaction-processing architectures for distributed real-time data base systems. J Syst Softw 31(02):97–108
21. Qin B, Liu Y (2003) High performance distributed real-time commit protocol. J Syst Softw 68(02):145–152