A Delayed-Initiation Risk-Free Multiversion Temporally Correct Algorithm[1]

Azzedine Boukerche and Terry Tuck

Department of Computer Sciences, University of North Texas
{boukerche, tuck}@cs.unt.edu

Abstract. In this paper, we devise a "temporally reordering" mechanism of supporting update transactions that are impacted by delays (e.g., network delays) to the extent that they cannot be executed because of the irreversible progress of other conflicting transactions (i.e., data dependent and temporally dependant) extend this scheme with a delayed-initiation mechanism. This mechanism allows (a) the impacted update transaction to be repositioned to the earliest supportable point in the temporal ordering of transactions, and (b) the associated transaction manager (and in turn, the application or entity that submitted the transaction) to be notified of the new position, thereby providing the opportunity for adjustments or transaction termination. We describe the risk-free MVTC (RF-MVTC) algorithm and its delay-initiation variation (RF-MVTCD). We also present the set of experiments we have carried out to study the performance of MVTC and its variations: RF-MVTC, and RF-MVTCD.

1 Introduction

In our earlier work, we have proposed table-level writeset predeclarations as a method for identifying a priori inter-transactional conflict. This novel method allows transaction concurrency (i.e., speed) to be increased without a corresponding increase in the risk of encountering unidentified conflict. Consequently, a risk-free MVTC concurrency control algorithm as a risk free alternative to Conservative MVTC was proposed in [1,2]. In this paper, we devise a temporally reordering" mechanism of supporting update transactions that are impacted by delays (e.g., network delays) to the extent that they cannot be executed because of the irreversible progress of other conflicting transactions (i.e., data dependent and temporally dependant) extend this scheme with a delayed-initiation mechanism. This mechanism allows (a) the impacted update transaction to be repositioned to the earliest supportable point in the ordering of transactions, and (b) the associated transaction manager (and in turn, the application or entity that submitted the transaction) to be notified of the new position, thereby providing the opportunity for adjustments or transaction termination.

In our database model, the requirement of immediate execution of write operations requires support for multiple versions of each data item. Reducing semantic incorrectness associated with the return of an incorrect data item version suggests a non-

[1] This work was supported by the Texas Advanced Research Program ARP/ATP and UNT Faculty Research Grants.

aggressive, if not conservative, synchronization technique. The requirement for an execution schedule that is equivalent to a timestamp-ordered serial schedule necessitates the use of timestamp ordering. Finally, the requirement for improved concurrency combined with a non-aggressive synchronization technique requires predeclaration of the data items to be accessed. Combining this with the requirement for ease of use for the application developer requires that predeclaration be done at other than the data-item level.

2 Risk-Free MVTC and Its Delayed-Initiation Variation

The correctness constraint for all schedules produced by the MVTC concurrency control algorithm is that they are conflict serializable and computationally equivalent to a temporally ordered serial schedule for the same set of transactions. As its name implies, the operation of the Risk-free MVTC algorithm is further constrained to be risk-free with respect to the semantic correctness of all responses to *read* messages. In other words, the risk associated with the temporary return of incorrect values followed by an abort is avoided. In our earlier work, we have shown that both conservative and risk-free MVTC exhibit a better performance than previous concurrency control schemes. However, due to unexpected events such as network failures or site failures in a distributed environment, these schemes may fail. Hence, win this paper, we introduce a delay-initiation variant of the Risk-free MVTC algorithm. The key idea of this scheme is that if we delay the initiation of every transaction's first *read* message, we provide a delay for transactions that offsets anomalous[2] network delays incurred by *begin* messages of older transactions. This delaying strategy effectively normalizes the network delays incurred by transactions during the first part of their execution. This scheme is relatively simple in comparison with the original MVTC algorithm, yet is potentially more robust with respect to late-arriving *begin* messages. The tradeoff for this increased robustness is an expected delay in the response to a transaction's first *read* operation.

While Write and Read rules are the same as those for MVTC. The Read rule, when combined with the design constraint of being risk-free (i.e., avoiding semantic incorrectness associated with the return of an incorrect version for the targeted data item), requires that a database delay its response to a *read* request until the correct version is both available and committed. The Delay rule ensures that every *read* request is returned to the correct and committed data-item value. When combined with the Read rule, the Delay rule allows databases to respond to *read* requests with only data values that have been written by committed transactions that immediately precede the reader in the temporal order in terms of data-item-level conflict. When combined with the Reorder rule (below), the second part of Delay rule offers the advantage that all responses are risk-free, and no transaction will ever need to be restarted for updates of a late-arriving writing transaction. The Delay rule is an effective *read-write* synchronization mechanism as long as the writing transaction's *begin* message is received at the database prior to its processing of a conflicting *read* from a younger transaction. However, it is possible for the writer's *begin* message to be affected by network de-

[2] If network delays are identical for all messages there can be no "late" *begins*, as all younger *read* msgs will be equally late.

lays (or similar) to the extent that it is received at the local database after the servicing of a conflicting *read*. In such cases, it is necessary to reposition the writer in the temporal order of transactions to the earliest position that maintains that the writer is younger than all serviced readers of the declared tables.

The concept of delayed initiation is best explained by first detailing the events that motivate its use. With the original Risk-free MVTC algorithm, whenever a transaction's *begin* message arrives at a local database at which some younger reader has already accessed, there is a chance that allowing the older transaction to proceed will lead to a non-serializable schedule. This chance exists when the reader has read from one of the tables included in the table list of the delinquent *begin*, and is a tested condition in the algorithm's processing of *begin* messages. In order to avoid the risk of a non-serializable schedule, the corrective action with the Risk-free MVTC algorithm is to reject the *begin*, and return with the rejection a new timestamp that will reorder the associated transaction to a younger temporal position. Recall that the key to the concept of delayed initiation is the realization that a delay between the receipt and the processing of the younger transaction's *read* message decreases the likelihood of a reorder.

With respect to transaction turnaround times, the delayed initiation approach may appear to be a costly approach for the sake of reducing transaction reorders. Since the issue is one of trading longer turnaround times for fewer transaction reorders, the actual cost is application specific, and calculable only after identifying within the application the impact of reordered transactions. However, the general cost of the approach is limited by several factors that are identified in the following points: (i) Update-only transactions need not be delayed. If a transaction includes no *read* operations, cannot cause another older transaction to be reordered; (ii) At most one delay is needed per transaction. By delaying the processing of a transaction's first *read*, all *reads* within the transaction are effectively delayed; and (iii) Transaction timestamps facilitate delay calculations. Since timestamps reflect the global system time of a transaction's initiation at its associated transaction manager, the duration of the delay between receipt and processing of a particular *read* can be limited to the needed amount.

To be more in-line with the intent of delayed-initiation, it is more desirable to induce a fixed-duration delay between the start of transactions and the processing of their first *read* messages. For example, it might be desirable to ensure that (a) no first *read* message is processed without a delay of, say, twice the expected network delay, and (b) no first *read* message is further delayed if it has already incurred a delay of more than twice the expected network delay. Transaction timestamps provide a simple way to calculate the duration of delays. Since timestamps reflect the global system time of a transaction's initiation at its associated transaction manager, the actual delay incurred by a transaction's first *read* can be calculated: $delay_{actual} = t_{receipt} - timestamp$. Given a specific value for the desired delay before the processing of transactions' first *read* messages, $delay_{total}$, the delayed-initiation delay is the difference of the two: $delay_{di} = delay_{total} - delay_{actual}$, such that $delay_{actual} < delay_{total}$.

3 Simulation Experiments

In our experiments to evaluate the performance of MVTC and its Risk-Free delayed initiated variation scheme, we have used two types of platforms interconnected with a 10 Mbs LAN. In order to reduce the likelihood of conflict between update transactions, the writeset size was fixed at two data items. The data items selected for update were chosen at random, thereby distributing the probability of update uniformly across all data items within the database. In order to increase the likelihood of conflict with the read-only transactions, the ratio of update to read-only transactions was set at 4-to-1. This is accomplished in the experiment runs by restricting each TM to one of the two types of transactions, and allowing the TMs to execute as many transactions as possible within a run. In all runs, 10 TMs were executed simultaneously against a single database. The readsets for the read-only transactions were selected using a sequential pattern in order to produce readsets consisting of adjacent data items. This pattern was chosen in an effort to focus the accesses of each read-only transaction to the fewest tables without reading any data item more than once. Using this pattern, experiment runs were executed with mean readset sizes for the read-only transactions of 5- 100 data items. As a percentage of the total database size, these readset sizes correspond to 1-10, and 20%, respectively.

Let us now turn to our results.

(a) Our results indicate that each of the algorithms executed the most update transactions for the runs with smaller readset sizes. For the proposed algorithms, with only two TMs executing read-only transactions, resource contention at the database is relatively low with the smaller transactions, and the update transactions are consequently able to execute more quickly. With C-MVTO, however, the higher throughput for the runs with smaller read-only transactions is an indirect result of the blocks by the update transactions on read-only transactions being relatively short. Unlike the proposed algorithms, as the size of the read-only transactions is increased, the throughput with C-MVTO decreases substantially, as the duration of blocks is increased by larger readset sizes.

Regarding the relative performance of the different algorithms, our results clearly shows the poor performance of C-MVTO; it is able to execute update transactions at an average throughput of only 37% of that with RFMVTC-D, the poorest performing of the proposed algorithms. This result was expected, since the lack of table predeclarations with C-MVTO means that progress on each TM's transaction must be blocked until no other younger transactions are active. RFMVTC, on the other hand, provided the highest throughput at an average of over 112 offsets the negative impact on turnaround time. The most significant positive effect caused by the delayed initiation is a smaller fraction of update transactions that experience blocks on conflicting transactions. Indeed, we have observed that the fraction transactions per second. MVTC was the middle-performing proposed algorithm. It averaged 84% of the throughput of RFMVTC, and 9% better than RFMVTC-D.

Our results also indicate that the higher performance for throughput in comparison to turnaround time, a positive effect caused by the delayed initiation that of blocks for RFMVTC-D was less than 50% of that for the other two algorithms. By delaying the initiation of a given transaction, it becomes more likely that conflicting transactions will commit prior to the transaction's initiation. Consequently, it becomes less likely that blocks are required for any given transaction.

(b) During the course of our experiments, we have observed that as the size of the read-only transactions is increased, the necessary decrease in throughput occurs with all algorithms. For experiment runs with the smallest readset sizes (i.e., those with readsets fewer than 25 data items), the proposed MVTC algorithm and its delayed initiated risk free variation yield significantly greater throughput than C-MVTO. Without the benefit of table-level writeset predeclarations, C-MVTO must block the execution of every read-only transaction while younger transactions are active, thereby missing the advantage of concurrency experienced with the proposed algorithms. As the size of the readsets is increased, the duration of the blocks decreases relative to the time required for reading more data items. For runs with readset sizes of 50 or more, throughput with C-MVTO surpasses that with MVTC and RFMVTC. This is due to contention at the database, and is covered in the discussion section.

We have also observed that RFMVTC outperforms both MVTC and RFMVTC-D in terms of throughput for runs with the smallest readset sizes. The impact of additional overhead with MVTC and delayed initiation with RFMVTC-D is too great in comparison to the potential benefits, constrained by only two TMs executing relatively small read-only transactions. For runs with readsets of five data items, throughput with MVTC and RFMVTC-D lag that with RFMVTC by 10% and 14%, respectively. For runs with readset sizes of 25 or more data items, RFMVTC-D replaces RFMVTC as the best-performing algorithm. This appears to be an indirect benefit of the delayed initiation mechanism. With eight TMs concurrently executing the smaller update transactions, the fraction of their time spent idle during the delayed initiation is significant. The execution of the read-only transactions benefits from the reduced-contention database resulting from the delays with the update transactions.

(c) Our results indicate that the turnaround time increases as the readsets become larger. Our results indicate that RFMVTC provides the highest performance for smaller readsets, and RFMVTC-D the highest for larger readsets. The explanations for the relative performance differences provided in the throughput discussion hold here.

Read-only transaction performances with the MVTC and RFMVTC algorithms are contrasted. As the readset size increases from 5 data items, the relative performance of MVTC drops. At 25 data items, the advantage for RFMVTC reaches its maximum: the overall read-only transaction performance for MVTC is only 70% of that with RFMVTC. However, at this point the trend reverses, and the relative performance of MVTC increases with larger readset sizes. For runs with the largest readset size, performance with MVTC is within 5% of that with RFMVTC.

4 Conclusion

In this paper, we have proposed to enhance the MVTC scheme by introducing the delayed-initiation variant to the Risk-free MVTC concurrency control algorithm. With the inclusion of the delayed-initiation mechanism, this algorithm maintains the benefits of the original Risk-free MVTC algorithm while addressing its shortcoming: increased temporal reordering. We have also presented a set of experiments to study the performance of MVTC and its risk-free and delay initiation variations. Our results indicate that for less-tolerant applications, the delayed-initiation variant of Risk-free MVTC is the algorithm of choice; our experimental results show that it provides good performance even when long-duration delays are used for the delayed-initiation mechanism.

References

[1] Boukerche, A., S. K. Das, A. Datta and T. LeMaster. “Implementation of a Virtual Time Synchronizer for Distributed Databases.” *Proceedings of EuroPar ’98, LNCS*, 534-538.

[2] Boukerche A, Tuck T., "Improving Conservative Concurrency Control in Distributed Databases", *Proc. EuroPar 2001, LNCS 2150*, Springer Verlag, pp. 301-309, 2001.

[3] Georgakopoulos, D., M. Rusinkiewicz and W. Litwin. “Chronological Scheduling of Transactions with Temporal Dependencies.” *VLDB Journal* 3:1 (January 1994): 1-28.

[4] Jefferson, D., and A. Motro. “The Time Warp Mechanism for database concurrency control.” *Proc. of the 2nd Int’l Conference on Data Engineering*, (1986): 474-481.

[5] Nicol, D.M. and X. Liu. “The Dark Side of Risk (What your mother never told you about Time Warp).” *Proc. of the 11th Workshop on Parallel and Distributed Simulation*, (1997): 188-195.

[6] Özsu, M.T. and P.Valduriez. *Principles of Distributed Database Systems.* Prentice Hall, 1999.