IJARCCE

ISSN (Online) 2278-1021
ISSN (Print) 2319 5940

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 5, Issue 2, February 2016

Lost Update Problem, Pessimistic and Optimistic Concurrency

Ala Yusef

Department of Computer Science, Sacred Heart University, Bridgeport, CT, USA

Abstract: In this paper, lost update problem occurs when two users read and update the same data in a particular row of the same database at the same time. one of the main problems found in tools that support MDE is the fact that little attention is paid to questions related to the platform features in the software development trajectory. Many machine learning algorithms iteratively transform some global state (e.g., model parameters or variable assignment) giving the illusion of serial dependencies between each operation.

Keywords: Update, Pessimistic, Optimistic, Concurrency, Roles.

I. INTRODUCTION

The lost update problem occurs when two users read and update the same data in a particular row of the same database at the same time. The lost update problem can be happened when the users are modifying some data without knowing about that there are other users are updating the same data that he or she is updating. The result of that, the last user who has updated the data is going to have to correct amount or values [1]. Deadlock is a case that happens when two or more users want to use the same resource at the same time by blocking each another to get to the resource. Clearly the deadlock exists in transaction tables in database, especially when there is a group of blocked transactions each having a data and waiting to get another data that held by another transaction. Obviously no one of them have the ability to continue unless the other transaction unlocked the data. In the main while they are losing and wasting their time and nothing can be done because they are waiting for each other.

The optimistic concurrency term means that there are two or more than two users updating the same row at the same time without locking each other. Whenever that occurs one of those users is going to update the data and the second user will get a message that the data has been updated, so the second user will notify the changes that just happened. pessimistic concurrency means only one user locking the row to disallow another user to update the same data at the same time [3]. One of the advantages of pessimistic concurrency is that the user ensures that the data has been updated to the database safely. Another advantage is it easy to be implemented. It has some disadvantages such as it is not fast as the optimistic concurrency, and it is not able to scalable which many it is not good for online stores because it is tied for limited users.

The aim of this study was to develop a successful PPI with a strong theoretical and empirical foundation that can address he weaknesses and limitations of previous work, as well as increase participant compliance [2]. A wealth of research supports the cultivation of optimism as an individual skill that can improve psychological well-being. Optimism involves a positive outlook on life, both during times of success and struggle (Segerstrom, 2006). Optimistic people believe that good things will happen to them in the future and that their goals are achievable. Optimism is strongly correlated with positive affect and better coping in a wide variety of stressful situations (Carver, Schier, & Segerstrom, 2010). It is also associated with fewer mental and physical health symptoms (Lench, 2011), increased motivation and effort, and increased engagement with one's goals (Segerstrom, 2006). Taken together, research suggests that being optimistic is associated with various indices of positive functioning.

II. TRANSACTION COMMIT

On commit, the STM acquires the locks corresponding to the objects in its log. The system makes use of the fair multiple-reader, single-writer version of the MCS lock in [2] to allow different threads to commit concurrently even if their read-sets overlap. These locks build a queue of requestors to provide FIFO order, while allowing for multiple concurrent readers. The lock header is containing the current reader count and pointers to the queue tail and the first writer in the queue. Each lock requestor allocates a queue node in shared memory, and adds it to the end of the queue determined by the tail pointer. Threads waiting for a lock spin-wait in their own queue node. When a writer receives a lock, it proceeds, and on release, it notifies the next node in the queue. Readers must, additionally, increase and decrease the reader count and notify consecutive readers to allow for concurrent reading. The multiple updates of this reader count field can generate coherence contention.

A lock-based STM adds four main overheads when compared with running the same transactions on a native HTM First, the locking mechanism itself is not necessary in a HTM system. Second, transactions need to maintain the read-set and write-set lists. This introduces a list-search for each object accessed, and an increase in the used memory. In HTM systems the hardware itself tracks the objects accessed in the transaction (with read and write bits, signatures or other mechanisms). Third, on commit,

IJARCCE

ISSN (Online) 2278-1021
ISSN (Print) 2319 5940

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 5, Issue 2, February 2016

the lists have to be traversed to lock and validate the objects. Fourth, the indirection-based object structure makes it necessary to copy entire objects when opening them for update even if only a single field is going to be touched [4].

III. OPTIMISTIC CONCURRENCY

Many machine learning algorithms iteratively transform some global state (e.g., model parameters or variable assignment) giving the illusion of serial dependencies between each operation. However, due to sparsity, exchangeability, and other symmetries, it is often the case that many, but not all, of the state-transforming operations can be computed concurrently while still preserving serializability: the equivalence to some serial execution where individual operations have been reordered. This opportunity for serializable concurrency forms the foundation of distributed database systems. For example, two customers may concurrently make purchases exhausting the inventory of unrelated products, but if they try to purchase the same product then we may need to serialize their purchases to ensure scent inventory. One solution (mutual exclusion) associates locks with each product type and forces each purchase of the same product to be processed serially [7]. This might work for an unpopular, rare product but if we are interested in selling a popular product for which we have a large inventory the serialization overhead could lead to unnecessarily slow response times. To address this problem, the database community has adopted optimistic concurrency control (OCC) [14] in which the system tries to satisfy the customers' requests without locking and corrects transactions that could lead to negative inventory (e.g., by forcing the customer to check out again) figure 1.

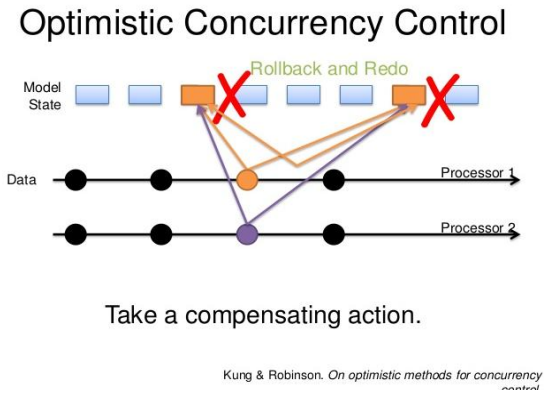

Fig 1: OPTIMISTIC CONCURRENCY

IV. PRECISE SERIALIZATION

The precise serialization (PS) algorithm solves the unnecessary transaction restart problem of the forward validation algorithm by checking every different type of data conflict and explicitly recording serialization order among transactions. It arranges the serialization order of two transactions whenever the two have a data conflict. The PS algorithm uses two different ordering actions to arrange serialization order of transactions: forward and backward ordering. The forward ordering places Tc~ after T v in execution history, i.e., their serialization order becomes T v --~ Tc~. This ordering is used to resolve conflicts incurred by write operations of T~R, i.e., read-write and write-write conflicts. The serialization order reflects the fact that Tc~'S updates do not affect the operations of T v. To resolve the other conflict type, i.e., write-read conflicts between T v and T~R, the PS algorithm, unlike FV, does not unconditionally restart Tc~. Instead, PS places To, ahead of T v in execution history, i.e., To, ---~ T v, which implies that TcR did not read from T v. This placement of To, ahead of T v in execution history is referred to as the backward ordering.

In the PS algorithm, a running transaction, Tc~, restarts only when it is involved in one or more conflicts caused by its write operations (read-write and write-write conflicts) as well as conflicts caused by its read operations (write-read conflicts) with a validating transaction T v. In such a situation, PS will attempt to place Tc~ both behind and in front of T v in execution history, which means a violation of serializability figure 2. Such Tc~ is referred to have an antagonistic conflict with T v and needs to be restarted. Note that such a transaction restart is inevitable and absolutely necessary to ensure data consistency [8].

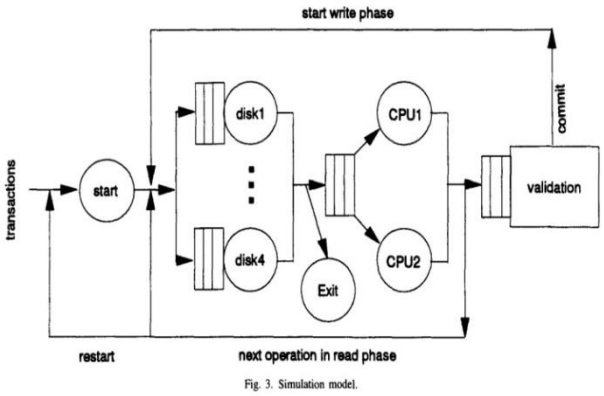

Fig 2: PRECISE MODEL

V. SYMBOLIC MODULAR DEADLOCK ANALYSIS

Concurrent programs are prone to a variety of thread-safety violations arising from the presence of data races and deadlocks. In practice, as data races are abundant and difficult to debug, they have garnered considerable attention from the program analysis community. A knee-jerk response to avoiding race conditions is evident in the prolific use of locking constructs in concurrent programs. Languages such as Java have promoted this by providing a convenient synchronized construct to specify mutual exclusion with monitors [6]. Locking is sometimes naively used as a "safe" practice, rather than as a requirement. Overzealous locking not only causes unnecessary overhead, but can also lead to unforeseen deadlocks. Deadlocks can severely impair real-time applications such as web-servers, database systems, mail-servers, device drivers, and mission-critical systems with embedded devices, and typically culminate in loss of data, unresponsiveness, or other safety and liveness violations. we focus on deadlocks arising from circular dependencies in synchronization constructs such as locks and signaling

DOI 10.17148/IJARCCE.2016.5265

primitives. Languages such as Java combine the mutual exclusion provided by locks with the cooperative synchronization provided by signaling primitives into a single monitor construct. In this paper, we use the abstract term lock to mean both specialized lock variables in languages such as C, C++ /pthread, and monitors used for enforcing mutual exclusion in Java. Deadlock detection is a well-studied problem, and both static and dynamic approaches have been proposed (Havelund 2000; Agarwal and Stoller 2006; Corbett 1996; von Praun 2004; Artho and Biere 2001;Naiketal.2009; Williams et al. 2005). Typically, such techniques construct lock-order graphs that track dependent- cites between locks for each thread. Lock-order graphs for concurrent threads are then merged, and a cycle in the resulting graph indicates a possibility of a deadlock. Such techniques typically assume a closed system, and are thus useful for detecting existing deadlocks in a given application. analyzing concurrent libraries for deadlocks has two main aspects: First of all, we wish to identify if, for any client, there are library methods that can be concurrently called in a manner that causes a deadlock. This is termed the deadlock ability problem. Secondly, we wish to use the results of this analysis to search for the existence.

We have shown how optimistic concurrency control can be usefully employed in the design of distributed machine learning algorithms. As opposed to previous approaches, this preserves correctness, in most cases at a small cost. We established the equivalence of our distributed OCC DP-means, OFL and BP-means algorithms to their serial counterparts, thus preserving their theoretical properties. In particular, the strong approximation guarantees of serial OFL translate immediately to the distributed algorithm. Our theoretical analysis ensures OCC DP-means achieves high parallel is without ascribing correctness. We implemented and evaluated all three OCC algorithms on a distributed computing platform and demonstrate strong scalability in practice [10]. We believe that there is much more to do in this vein. Indeed, machine learning algorithms have many properties that distinguish them from classical database operations and may allow going beyond the classic formulation of optimistic concurrency control. In particular, we may be able to partially or probabilistically accept non-serializable operations in a way that preserves underlying algorithm invariants. Laws of large numbers and concentration theorems may provide tools for designing such operations. Moreover, the convict detection mechanism can be treated as a control knob, allowing us to softly switch between stable, theoretically sound algorithms and potentially faster coordination-free algorithms [11].

VI. CONCLUSION

one of the main problems found in tools that support MDE is the fact that little attention is paid to questions related to the platform features in the software development trajectory. As a result, MDE tools are limited to certain platforms and PIM-into-PSM model transformation processes. In order to achieve efficient, easily adaptable model transformation processes, the specification of independent platform features is necessary. Concerning RTOS-based embedded software development, the benefits in using this approach become even more evident due to both the inherent complexity of this kind of software and the existence of a wide variety of applicable platforms.

We are concave able but less costly than persistent save points. In a system where ceded here with volatile save points or mark points, which are not the level of data contention is low and the check pointing overhead is high, check pointing will result in degraded performance (increased transaction response time) even when adequate processing resources are available. This is especially so when multiple checkpoints are taken. An adaptive method need be introduced to suppress check pointing at low data contention levels and activate it only when the data contention level is high.

REFERENCES

[1] Gray, J., & Lamport, L. (2006). Consensus on transaction commit. ACM Transactions on Database Systems (TODS), 31(1), 133-160.
[2] Lamport, L., & Fischer, M. (1982). Byzantine generals and transaction commit protocols (Vol. 66). Technical Report 62, SRI International.
[3] Deshmukh, J. V., Emerson, E. A., & Sankaranarayanan, S. (2011). Symbolic modular deadlock analysis. Automated Software Engineering, 18(3-4), 325-362.
[4] Deshmukh, J., Emerson, E. A., & Sankaranarayanan, S. (2009, November). Symbolic deadlock analysis in concurrent libraries and their clients. In Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering (pp. 480-491). IEEE Computer Society.
[5] Farzan, A., Chen, F., Meseguer, J., & Roşu, G. (2004, July). Formal analysis of Java programs in JavaFAN. In Computer aided verification (pp. 501-505). Springer Berlin Heidelberg.
[6] Lee, J. (1999). Precise serialization for optimistic concurrency control. Data & knowledge engineering, 29(2), 163-178.
[7] Lehr, M. R., Kim, Y. K., & Son, S. H. (1995, June). StarBase: a firm real-time database manager for time-critical applications. In Real-Time Systems, 1995. Proceedings., Seventh Euromicro Workshop on (pp. 317-322). IEEE.
[8] Gray, J., & Lamport, L. (2006). Consensus on transaction commit. ACM Transactions on Database Systems (TODS), 31(1), 133-160.
[9] Mohan, C., Strong, R., & Finkelstein, S. (1983, August). Method for distributed transaction commit and recovery using Byzantine agreement within clusters of processors. In Proceedings of the second annual ACM symposium on Principles of distributed computing (pp. 89-103). ACM.
[10] Gray, J. (1990). A comparison of the Byzantine agreement problem and the transaction commit problem. In Fault-tolerant distributed computing (pp. 10-17). Springer New York.
[11] McCarthy, D., & Dayal, U. (1989, June). The architecture of an active database management system. In ACM Sigmod Record (Vol. 18, No. 2, pp. 215-224). ACM.

DOI 10.17148/IJARCCE.2016.5265