Brief Announcement: On Enhancing Concurrency in Distributed Transactional Memory

Bo Zhang
ECE Dept., Virginia Tech
Blacksburg, VA 24061, USA
alexzbzb@vt.edu

Binoy Ravindran
ECE Dept., Virginia Tech
Blacksburg, VA 24061, USA
binoy@vt.edu

Abstract

Distributed transactional memory (TM) models based on globally-consistent contention management policies may abort many transactions that could potentially commit without violating correctness. To reduce unnecessary aborts and increase concurrency, we propose the distributed dependency-aware (or DDA) model for distributed TM, which manages dependencies between conflicting and uncommitted transactions so that they can commit safely. We present a distributed algorithm to decide whether to abort a transaction based on local precedence graphs that model the established dependency relationships. We analyze the performance of our algorithm and illustrate the inherent tradeoff of the DDA model between communication cost and concurrency.

1. Overview

A distributed TM model supports the TM API in a distributed system consisting of a network of nodes that communicate by message-passing links. Supporting TM in distributed systems is motivated by the similar difficulties of lock-based synchronization methods employed by distributed control-flow programming models such as RPCs. The core design aspects of a distributed TM system include two elements. The first element is the *conflict resolution strategy*. Two transactions *conflict* if they access the same object and one access is a write. Most existing TM implementations adopt a conflict resolution strategy that aborts one transaction whenever a conflict occurs—i.e., by a contention management module. The second element is the *distributed cache-coherence protocol*. When a transaction attempts to access an object in the network, a distributed cache-coherence protocol must locate the latest cached copy of the object, and move a read-only or writable copy to the requesting transaction. The protocol must guarantee that at any time, there exists only one writable copy of each object in the system.

Most of the past works on TM in distributed systems ([1], [2]) focus on the design of cache-coherence protocols, while assuming a contention management-based conflict resolution strategy. While easy to implement, such a contention management approach may lead to significant number of unnecessary aborts, especially for read-dominated workloads [3], thereby reducing concurrency. Thus, we consider the problem of how to increase concurrency in distributed TM conflict management. We present a new distributed TM model—the distributed dependency-aware (or DDA) model, which allows multiple conflicting transactions to proceed as long as the correctness criterion is not violated. In contrast to Herlihy and Sun's globally-consistent, contention management model (or "GCCM") [1], the DDA model relaxes the restriction of allowing only one transaction to proceed in the event of a transactional conflict by setting up precedence relations between the conflicting transactions. A transaction can commit as long as its established precedence relations with other transactions are not violated.

To support TM API in distributed systems, we consider Herlihy and Sun's data-flow model [1], where transactions are immobile (running at nodes which invoke them), and objects move from node to node. Each node has a *TM proxy* that provides interfaces to the TM application and to proxies of other nodes. The TM proxy enables two basic operations: $read$ and $write$. The $read(o, v)$ operation returns the value v of the object o. The $write(o, v)$ operation sets the value of the object to v given as an argument.

2. Distributed Dependency-Aware Model

Object version list. The DDA model allows a distributed TM system to manage multiple versions of shared objects. Each object has to maintain an object version list. We adopt the technique similar to [4], but allow writers to add versions before their commits. Specifically, each object o is associated with a totally ordered sequence of versions. At any given time, the versions of an object are numbered in increasing order. The numbering of the object version may change since the versions are inserted into or removed from the object list. The object version $o.v_n$ includes the data $o.v_n.data$, the writer transaction $o.v_n.writer$, the writer status $o.v_n.writerstatus \in \{committed, pending\}$, and a set of readers $o.v_n.readers$. A read operation of object o returns the value of one of o's version. A write operation of object o adds a new version to o's version list and sets its corresponding writer status to $pending$. Note that the $pending$ status implies that the version $o.v_n$ is written by a live transaction. If this transaction aborts, the corresponding version is removed from the object list. If this transaction commits, the corresponding writer status is set to $committed$. Each transaction keeps a $readList$ and a $writeList$. An entry in a $readList$ points to the version that has been read by the transaction. An entry in a $writeList$ points to the version written by the transaction.

Generally, the following principles are applied for read/write operations in the DDA model:

1 The read operation always returns a value of the object written by the latest committed transaction. Formally, a read operation on object o returns the value $o.v_m$, where $o.v_{m'}.writerstatus = pending$, for all $m' > m$.

2 The write operation always writes a value with the highest version number. Formally, a write operation to object o adds the version $o.v_{n+1}$ to object o if its latest version is $o.v_n$.

3. Decision Making Algorithm

In the DDA model, the TM system allows multiple conflicting transactions to proceed simultaneously, while ensuring transactional correctness. A transaction is aborted only if it violates the correctness criterion, e.g. opacity [5]. The basic idea to guarantee correctness is to maintain a *precedence graph* of transactions and keep it acyclic, which has been adopted by some recent TM efforts in multiprocessor systems (e.g., [4], [5]). The vertices of the global precedence graph PG are transactions. The directed edges of PG are formed by the real-time order and dependency relationships (write after read, write after write, read after write). Applying this method for distributed TM systems introduces some unique challenges. The key challenge is that, in distributed systems, each node has to make decisions based on its local knowledge. A centralized algorithm—e.g., assign a coordinator node to maintain the precedence graph and make decisions whenever a conflict occurs, involves frequent interactions between each individual node and the coordinator node.

A centralized algorithm is impractical due to the underlying high communication cost. In practice, each node can only maintain a *local precedence graph*, which contains partial knowledge of the global precedence relations. Transaction T_i forms a directed labeled precedence graph, PG_i, based on the dependencies created during the transaction history.

Construction. Basically, PG_i only records the direct precedence relations between T_i and other nodes. Note that the cache-coherence protocol always carries a read/write request to the latest object writer. We do not add edges to/from a committed transaction in this algorithm. A CYCLEDETECTION algorithm is invoked to decide whether an edge can be added without generating cycles in PG_i; if not, the transaction is aborted. For a read operation, a SEARCHVERSION(T_j, o) may be invoked to search the latest committed version of object o. The idea to search that version is to first find a "possibly" latest committed writer $o.v_m.writer$ of o based on the object version list. Due to the possible delay of the update, $o.v_m.writer$ may not be the latest committed writer of o. However, we can find the correct writer by following the sequence of $W \rightarrow W$ edges starting from $o.v_m.writer$. An $R \rightarrow W$ edge is added by T_j, and the transaction writes the version after it. A write operation is more complex. Generally, if T_j writes to the object o held by T_i, one $W \rightarrow W$ edge and a set of $R \rightarrow W$ edges are added to PG. Hence, T_i collects the response from itself and all its readers to determine whether a cycle has been generated after the edges are added. If the write operation is allowed, a set of corresponding edges are established. Note that our algorithm is different from past distributed deadlock detection algorithms in substantial technique ways to construct and update local precedence graphs.

Garbage Collection. In the DDA model, a terminated transaction can be removed from the global precedence graph without violating correctness. We prove the following theorem in [6]:

Theorem 1: If a transaction commits or aborts, it will not participate in any cycle even if it is not removed from PG_i.

Run-time Complexity. The write complexity is $O(\#C + \#L^2)$, where $\#C$ is the number of nodes visited by one invocation of the cache-coherence protocol and $\#L$ is the number of live transactions. The read complexity is $O(\#C + \#L)$. A commit or abort cost is $O(\#L)$ since a transaction only has edges with live transactions. The complexity analysis of the operations of the DDA model illustrates the inherent tradeoff between the communication overhead and the concurrency of transactions. Higher concurrency introduces larger sizes of both global and local precedence graphs, which leads to higher costs in detecting cycles (due to the larger size of possible cycles formed).

References

[1] Maurice Herlihy and Ye Sun, "Distributed transactional memory for metric-space networks," *Distributed Computing*, vol. 20, no. 3, pp. 195–208, 2007.

[2] Bo Zhang and Binoy Ravindran, "Dynamic analysis of the relay cache-coherence protocol for distributed transactional memory," in *IPDPS '10: Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium*, Washington, DC, USA, 2010, IEEE Computer Society.

[3] Hagit Attiya and Alessia Milani, "Transactional scheduling for read-dominated workloads," in *OPODIS '09: Proceedings of the 13th International Conference on Principles of Distributed Systems*, Berlin, Heidelberg, 2009, pp. 3–17, Springer-Verlag.

[4] Idit Keidar and Dmitri Perelman, "On avoiding spare aborts in transactional memory," in *SPAA '09: Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, New York, NY, USA, 2009, pp. 59–68, ACM.

[5] Rachid Guerraoui and Michal Kapalka, "On the correctness of transactional memory," in *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, New York, NY, USA, 2008, pp. 175–184, ACM.

[6] Bo Zhang and Binoy Ravindran., "On enhancing concurrency in distributed transactional memory," Tech. Rep., Virginia Tech, 2010, http://www.real-time.ece.vt.edu/dependence_dtm_TR.pdf.