On Ordering Transaction Commit

Mohamed M. Saad Roberto Palmieri Binoy Ravindran

Virginia Tech

{msaad, robertop, binoy}@vt.edu

Abstract

In this poster paper, we briefly introduce an effective solution to address the problem of committing transactions enforcing a predefined order. To do that, we overview the design of two algorithms that deploy a cooperative transaction execution that circumvents the transaction isolation constraint in favor of propagating written values among conflicting transactions. A preliminary implementation shows that even in the presence of data conflicts, the proposed algorithms outperform other competitors, significantly.

Categories and Subject Descriptors D.1.3 [*Software*]: Concurrent Programming; H.2.4 [*Systems*]: Transaction processing

Keywords Transactional Memory, Commitment Ordering

1. Introduction

Transactional Memory (TM) [5] is an easy abstraction to program concurrent applications. Its integration into main-stream compilers and programming languages such as GCC and C++ gives TM, respectively, accessibility and concreteness.

In this poster paper we provide the design of two TM implementations that commit transactions enforcing an order defined prior to their execution. Roughly, by transaction ordering we mean considering not just the set of transactions as input of the problem, but also the specific commit order that must be enforced for them. Such a reformulation inherently brings up a fundamental trade off between the level of parallelism achievable given the need of committing in-order, and the performance of the single threaded execution without any software instrumentation (which is rather needed to prevent conflicts when running in parallel).

Ordering transactions before the execution is a known problem, mostly relevant to deployments where an external service is in charge of providing the commit order to satisfy certain properties (e.g., equivalent semantics or system dependability). Examples of those deployments include (but are not limited to): loop parallelization [4], and fault-tolerance using the state machine replication (SMR) approach [9]. In the former, loops designed to run sequentially are parallelized by executing their iterations concurrently, and guarded by TM transactions (as in [4] and [7]) to handle conflicts (i.e., data dependency) correctly. In that case, providing an order matching the sequential one is fundamental to enforce a

[Copyright notice will appear here once 'preprint' option is removed.]

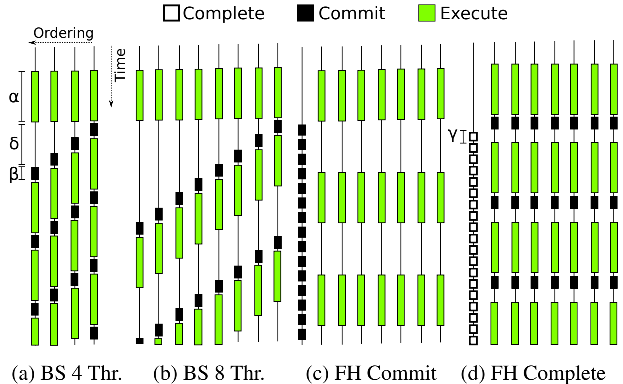


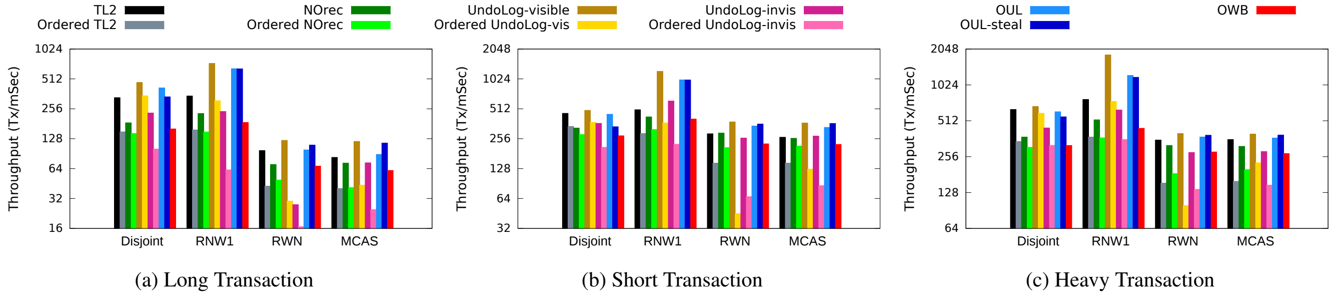
Figure 1: ACO using Blocking/Stall (BS) and Freeze/Hold (FH)

semantics (of the parallel code) that is equivalent to the original (sequential) code. Regarding the latter, SMR-based transactional systems order transactions (totally or partially) before their execution to guarantee that a state always evolves on several computing nodes, consistently. To do that, usually a consensus protocol is employed (e.g., Paxos [6]), which establishes a common order among transactions. After that, this order must be enforced while processing and committing those transactions.

In this poster paper we focus on Software Transactional Memory and we briefly introduce the design of two solutions to commit transactions in order while exploiting their parallel execution: *Ordered Write Back* (OWB) and *Ordered Undo Logging* (OUL). They represent two protocols that assume two different widely used techniques to merge transactions' modifications into the shared state, namely write back (in OWB) and write through (in OUL). Both of them share a common design using a dependency-aware cooperative model. Transactions employ a weaker isolation level, and exchange both data and locks to increase concurrency while preserving the commit order. More specifically, OWB uses data forwarding for uncommitted transactions that complete their execution successfully, while OUL leverages encounter time locking with the ability to pass the lock ownership to other transactions.

2. Transaction Ordering

A concurrency control that enforces an order of commits [8] ensures that when two operations o_i and o_j, issued by transactions T_i and T_j respectively, are conflicting, then o_i must be processed before o_j if and only if $T_i \prec T_j$ (i.e., i precedes j). Age-based Commit Order (ACO) deploys this idea, which mandates an adaptation of the classical TM model. As an example, the transaction conflict detection should guarantee that when $T_i \prec T_j$, T_i must not read a value written by T_j. We define a transaction T_j as *reachable* if all lower age transactions T_i, with $i < j$, are committed. This term depicts the fact that T_j has been reached by a serial execution where all T_i transactions committed in the order $\{1, \ldots, i, j\}$.



(a) Long Transaction

(b) Short Transaction

(c) Heavy Transaction

Figure 2: Peak performance of all competitors (including unorderd) using all micro benchmarks (Y-axis is log scale).

Assume an ACO where transaction T_i precedes transaction T_j ($T_i \prec T_j$). Although T_j may finish executing its operations before T_i, it must wait for T_i to commit first; then T_j can start committing (and validating if needed) its changes. This strategy forces the thread executing transactions with higher age to either: *block* [4] (see Figures 1a & 1b), or *freeze* [10] (see Figure 1c). Either way, an additional delay for those transactions is introduced.

To construct our cooperative model, we start by decoupling two events of a transaction execution: the time it *ends*, and the time it becomes *reachable*. Whenever a transaction performs all its operations successfully (i.e., without encountering any conflict with any lower-age concurrent transaction), it *commits* its modifications. We name this time as *end*. However, exposing written objects and releasing all transaction's meta-data may violate the ACO if all lower-age transactions are not entirely finalized or when the transaction conflicts with a lower-age transaction that did not run at the time of exposing the changes. For this reason, we define a new operation, called *complete*, invoked whenever a transaction becomes *reachable*. The main difference between an ended transaction and a reachable transaction is that the former, although it already exposed its modifications, can still be aborted (and trigger the abort of other transactions), whereas the latter cannot be aborted anymore. Figure 1d shows a possible execution using our transactional model.

3. Algorithms

OWB employs a write-buffer approach, therefore a transaction writes its own modifications into a local buffer. At commit time, the transaction acquires a versioned-lock over its write-set and writes its changes to the shared memory. To avoid concurrent writers, the locks are not released until the transaction becomes completed or is aborted. However, to allow an early propagation of the modifications, greater-age transactions can still access those locked objects and, in case an abort is triggered, the committed transaction, which produced the invalidation, is responsible to abort any dependent transaction that has read the exposed values.

OUL is an undo-log algorithm that preserves the ACO. Here, transactional updates affect the shared memory at encounter time, while the old value is kept in a local undo-log. Unlike OWB, such a scheme implies that the transactions' order is guaranteed while operations are invoked. In order to deploy the above idea, each object is associated with a read-write lock. Each lock stores the reference to the (single) writer transaction, which can be either the current transaction holding the lock or the one that committed that version, and a list of concurrent readers, namely those active or committed transactions that accessed the version for reading it.

4. Evaluation

We provided a preliminary implementation of OWB, OUL, the ordered version of four existing well-known TM designs (i.e., TL2 [3], NOrec [2], and UndoLog with and without visible readers, and compared their performance using RSTM micro-benchmarks [1]. To evaluate the effect of different workload characteristics, such as the amount of operations per transaction, the transaction length, and the read/write ratio, on the performance. Each experiment included running half million transactions. For all micro benchmarks, we configured three types of transactions: short, long, and heavy. Both *short* and *heavy* have the same number of accesses, but the latter adds more local computation in between them. Such a workload is representative of workloads produced by parallelization frameworks. *Long* transactions simply produce more transactional accesses.

Figure 2 summarizes the peak performance of all competitors. Results show interesting trends: our OUL outperforms other ordered competitors consistently. In particular, the maximum speedup achieved is 4× over Ordered TL2, 4.3× over Ordered NORec, 8× over Ordered UndoLog visible, and 10× over Ordered UndoLog invisible.

Acknowledgments

This work is partially supported by Air Force Office of Scientific Research (AFOSR) under grant FA9550-14-1-0187.

References

[1] RSTM: The University of Rochester STM. http://www.cs.rochester.edu/research/synchronization/rstm/.

[2] L. Dalessandro, M. F. Spear, and M. L. Scott. Norec: Streamlining stm by abolishing ownership records. In *PPoPP*, pages 67–78, 2010.

[3] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *DISC*, pages 194–208, 2006.

[4] M. Gonzalez-Mesa, E. Gutierrez, E. L. Zapata, and O. Plata. Effective transactional memory execution management for improved concurrency. *ACM TACO*, 11(3):24, 2014.

[5] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, pages 289–300, 1993.

[6] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16 (2):133–169, 1998.

[7] M. Mehrara, J. Hao, P.-C. Hsu, and S. Mahlke. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. In *PLDI*, pages 166–176, 2009.

[8] Y. Raz. The principle of commitment ordering, or guaranteeing serializability in a heterogeneous environment of multiple autonomous resource managers using atomic commitment. In *VLDB*, volume 92, pages 292–312, 1992.

[9] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[10] L. Zhang, V. K. Grover, M. M. Magruder, D. Detlefs, J. J. Duffy, and G. Graefe. Software transaction commit order and conflict management, May 4 2010. US Patent 7,711,678.