

# Brief Announcement: Single-Version Permissive STM

Hagit Attiya  
Technion and EPFL  
hagit@cs.technion.ac.il

Eshcar Hillel  
Technion  
eshcar@cs.technion.ac.il

## ABSTRACT

We present a *single-version* STM that satisfies a practical notion of *permissiveness*: it never aborts read-only transactions, and it only aborts an update transaction due to another conflicting update transaction, thereby avoiding many spurious aborts. It avoids unnecessary contention on the memory, being *strictly disjoint-access parallel*.

## Categories and Subject Descriptors

D.1.3 [Software]: Programming Techniques—*Concurrent programming*; F.1.2 [Theory of Computation]: Computation by Abstract Devices—*Modes of Computation* [Parallelism and concurrency]

## General Terms

Algorithms

## 1. INTRODUCTION

In *Software transactional memory* (STM), a *transaction* encapsulates a sequence of operations on a set of *data items*: it is guaranteed that if a transaction commits, all its operations appear to be executed atomically. The STM can abort a transaction if it may violate the correctness of the memory. The items written by the transaction are its *write set*, the items read by the transaction are its *read set*, and together they are the transaction’s *data set*.

When many transactions need to simultaneously access the same data, frequent aborts may significantly impact the performance. Therefore, avoiding spurious aborts is an important goal. A *permissive* STM [4] never aborts a transaction unless necessary for correctness. There are scenarios where existing STMs abort a transaction even if it can be committed without violating correctness [4].

*Multi-version* STMs maintain multiple versions per data item. The main benefit of multi-versioning is supposed to be the avoidance of spurious aborts. However, multi-version STMs usually involve a complex implementation of a precedence graph and an intricate garbage collection mechanism, to remove old version, making it a less practical approach.

*Multi-version (MV) permissiveness* [3] (originally aimed for multi-version STMs) ensures read-only transactions (with an empty read set) never abort, and permits update transactions to abort only when in conflict with other update trans-

actions. This notion is practically achievable, especially useful in *read-dominated* workloads.

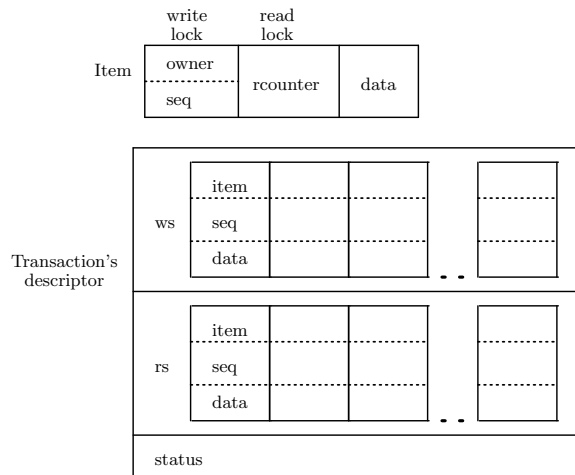
We present a simple *single-version* STM using read-write locks. This means that processes acquire locks for reading as well as for writing, making reads visible. Our STM maintains only a single version per item, yet it provides the benefits of MV-permissiveness: read-only transactions never abort (without having to a priori know they are read-only), and update transactions abort only if some item in their read set is overwritten by another transaction. Moreover, an update transaction blocks only due to a conflicting transactions; although it is blocking, the algorithm is deadlock-free.

Read locks are used instead of a central mechanism such as a global version clock, ensuring that the algorithm is *strictly disjoint-access parallel* [5], namely, processes executing transactions with disjoint data sets do not access the same memory locations. This property is considered critical for the scalability of an STM. An additional benefit of read locks is avoiding the overhead of incremental validation, which has great affect on performance, especially in read-dominated workloads.

## 2. THE ALGORITHM

The design principles of our algorithm are very simple: During the execution of the transaction a read operation acquires a read lock, which in case of update transactions is upgraded to an exclusive (write) lock at commit time; this allows to have “light” read-only transactions, which guarantee consistency without requiring validation and *without specifying them as such in advance*. An update transaction acquires exclusive locks only at commit time on *all* items in its data set by their order; this avoids deadlocks due to blocking cycles, and livelocks due to repeated aborts. An update transaction commits only in a “quiescent” configuration, where no other transaction is reading an item in its write set; this approach guarantees the consistency of reads by other transactions. Some of these principles are inspired by TLRW [2], an STM using read-write locks. TLRW, however, is not permissive as read only transaction may abort due to time out while attempting to acquire a lock.

Figure 1 presents the data structures of items and transactions’ descriptors used in our algorithm. We associate read and write locks with every item. The *write lock* includes an *owner* field, and a sequence number, *seq* field. The *owner* field is set to the id of the update transaction owning the lock and is set to 0 if no transaction holds the lock. The *seq* field holds the sequence number of the *data*, it is incremented whenever a new data is committed to the item, and



**Figure 1: Data structures used in the algorithm.**

it is used to assert the consistency of reads. The *read lock* is a simple counter, *rcounter*, of the number of transactions reading the current version of the item. The descriptors of each transaction, as in many other STMs, consists of the read set, *rs*, the write set *ws*, and the *status* of the transaction that is initially NULL, and can be set to COMMITTED or ABORTED. An entry in the read set includes a reference to an *item*, the *data* read from the item, and the sequence number of this data, *seq*. An entry in the write set includes a reference to an *item*, the *data* to be written in the item, and the sequence number of the new data, *seq*, i.e., the sequence number of the current data plus 1.

The *abstract data* and *abstract sequence number* of an item, are not always defined by the values the items holds (physically) in its attributes. If the write lock of the item is owned by a committed transaction that writes to this item, then the abstract data and sequence number of the item appear in the write set of the owner transaction. Otherwise (*owner* is 0, or the owner is not committed, or the item is not in the owner's write set), the abstract data and sequence number appear in the item.

We now outline how operations are handled.

**Read operation:** If the item is already in the transaction's read set, return the value from the read set. Otherwise, acquire a read lock on the item by increasing the read counter of the item. Then, the reading transaction adds the item to its read set with the abstract data and sequence number of the item.

**Write operation:** If the item is not already in the transaction's write set, then add the item to the write set. Set *data* of the item in the transaction's write set to the new data to be written. *No lock is acquired yet.*

**TryCommit:** If the transaction is *read-only*, i.e., the write set of the transaction is empty, then **commit** by setting *status* to COMMITTED, release all read locks by decreasing the read counters of the items in the transaction's read set. If this is an *update* transaction, it is committed as follows:

- Release the read locks on the items in the read set by decreasing the read counter of each item.
- Acquire write locks on all items in the data set of the transaction by their order.

If the item is in the read set, check that the sequence number in the read set is the same as the abstract sequence number of the item. If the sequence number has changed then

the data read is overwritten by another committed transaction and the transaction aborts: release all write locks acquired by the transaction by setting the *owner* in these items to 0. Then, **abort** by setting *status* to ABORTED.

Use **cas** to set the write lock: set *owner* from 0 to the id of the transaction, while asserting that *seq* is unchanged. If *owner* is non-zero there is another owner, so spin, re-reading the write lock until *owner* is 0 and *seq* is unchanged.

If the item is not in the read set, use **cas** to set *owner* from 0 to the id of the transaction. If *owner* is non-zero there is another owner, so spin, re-reading the *owner* until it is 0.

If the item is in the write set, set the sequence number of the item in the transaction's write set, *seq*, to the sequence number of the current data plus 1.

- Commit the transaction: use *k-compare-single-swap* [6] to set *status* to COMMITTED, while ensuring that all read counters of items in the transaction's write set are 0. If *rcounter* of one of these items is not 0, some transaction is reading from this item, then spin, until all *rcounters* are 0.
- Commit the new data: for each item in the write set, set *data* to the new value in the transaction's write set.
- Release the write locks on the write set: for each item in the write set, set *owner* to 0 and increase the sequence number by 1.
- Release the write locks on the read set: for each item in the read set, set *owner* to 0.

### 3. DISCUSSION

Our update transactions are not *obstruction free* since they may block due to other conflicting transactions. Indeed, a single-version, obstruction-free STM cannot be strictly disjoint-access parallel [5].

A read-only transaction modifies a number of memory locations that is equal to the size of its read set, as it increases and decreases the read counter of all these items. This matches the lower bound for disjoint-access parallel STMs [1], for read-only transactions that never abort.

Our STM points out the progress condition assumed for proving that a (weakly) disjoint-access parallel STM cannot be MV-permissive [3, Theorem 2]: a transaction *delays only due to a pending operation* (by another transaction). Note that in our algorithm, an update transaction may delay due to other transactions reading from its write set, even if none of their operations is pending. Our algorithm improves on their UP-MV STM, which maintains many versions of each data item, and is not disjoint-access parallel.

### 4. REFERENCES

- [1] H. Attiya, E. Hillel, and A. Milani. Inherent limitations on disjoint-access parallel implementations of transactional memory. In *SPAA '09*, pages 69–78.
- [2] D. Dice and N. Shavit. TLRW: Return of the read-write lock. In *SPAA '10*.
- [3] D. Perelman, R. Fan and I. Keidar. On maintaining multiple versions in STM. In *PODC '10*.
- [4] R. Guerraoui, T. A. Henzinger, and V. Singh. Permissiveness in transactional memories. In *DISC '08*, pages 305–319.
- [5] R. Guerraoui and M. Kapalka. On obstruction-free transactions. In *SPAA '08*, pages 304–313.
- [6] V. Luchangco, M. Moir, and N. Shavit. Nonblocking *k-compare-single-swap*. In *SPAA '03*, pages 314–323.