STO Verification

Hao Bai
Harvard University

Jingmei Hu
Harvard University

Xueyuan Michael Han
Harvard University

1 Introduction

Transactional memory is an appealing alternative to lock-based synchronization mechanisms. It provides in-memory operations with transaction abstraction analogous to database systems. Despite recent works on exploring different implementation strategies of software transactional memory, little attention has been paid to verify their correctness. In this work, We aim to verify in Coq the correctness of a particular software transactional memory system, namely STO. While opacity is deemed as a better candidate of correctness criterion for transactional memory, for the purpose of this project, we only verify that STO's implementation guarantees serializability, a weaker correctness criterion than opacity. We prove that all execution traces generated by STO's commit protocol are serializable. That is, there is always an equivalent serial trace whose execution result is the same as that of the original trace. Currently, our problem is that our Coq inductive types and definitions become increasingly complex as we strictly follow STO's commit protocol. We are concerned that usch complexity might indicate a bad design and thus leads to intractable proofs later on. While we strive to prove the serializability theorem, we might end up turning to simpler theorems (e.g., deadlock-free), if the serializability proof cannot be constructed.

2 Approach

Our first step is to prove serializability of a low level implementation of STO that concerns only *one* memory location. We mandate, for ease of constructing proofs, that only two memory-related operations are allowed on the location: *reading* and *writing* a value. We propose an inductive type `action` (Appendix A) that describes all possible operations within a transaction. Since we assume that many transactions are being executed simultaneously, with each one having its operations (possibly) interleave with those in other transactions, we define a trace to be a list of operations in all undergoing transactions. When one transaction executes an operation, we append this operation, along with the ID of the transaction, to this trace. An example trace with two transactions would look like this

$$
\begin{aligned}
[&(1, start\_txn); (1, read\_item\ 0);\\
&(2, start\_txn); (2, read\_item\ 0);\\
&(1, try\_commit\_txn); (1, validate\_read\_item\ True);\\
&(1, seq\_point); (1, commit\_txn\ 0); (2, write\_item\ 4);\\
&(2, try\_commit\_txn); (2, lock\_write\_item);\\
&(2, validate\_read\_itemTrue); (2, complete\_write\_item\ 1);\\
&(2, seq\_point); (2, commit\_txn\ 1)]
\end{aligned}
$$

We call this kind of traces *STO traces* and use an inductive type (Appendix B) to generate legal STO traces. We will explain what constitutes a legal STO trace by describing the inductive type `sto_trace` in detail in Section 2.1.

We also define a *serial trace* to be a trace that groups all operations of a successfully committed transaction together. For example, a serial trace of

the above example trace would be

$$[(1, start\_txn); (1, read\_item\ 0);$$
$$(1, try\_commit_t xn); (1, validate\_read\_item\ True);$$
$$(1, seq\_point); (1, commit\_txn\ 0); (2, start\_txn);$$
$$(2, read\_item\ 0); (2, write\_item\ 4);$$
$$(2, try\_commit_t xn); (2, lock\_write\_item);$$
$$(2, validate\_read\_item\ True); (2, complete\_write\_item\ 1);$$
$$(2, seq\_point); (2, commit\_txn\ 1)]$$

To prove the correctness of this transactional memory setting (i.e., with only one memory location), we need to prove that a STO trace composed of multiple transactions is serializable. That is, we must prove that

Theorem 1 $\forall$ *traces t, if t is* `sto_trace`*, then* $\exists$ *a reordering of t, t′, such that t′ is a* `sto_trace` *and a* `serial trace`*, and that the output of both traces (i.e., the final state of the machine) is identical.*

In Coq, this capstone theorem can be stated as

```
forall t : sto_trace, exists t',
    is_sto_trace t' ->
    is_serial_trace t' ->
    equate t t'.
```

2.1 Creating STO Traces

We will explain in this section each operation in `sto_trace`. Please refer to Appendix B for the corresponding Coq code.

empty_step is the base case in our inductive type.

start_txn_step signifies the beginning of a transaction. We ensure that the `tid` associated with this transaction is unique.

read_item_step is a `read_item` operation. This operation must follow a `start_txn` action, another `read_item` action, or a `write_item` action, of the same transaction ID. To be able to read from the memory location, we must also ensure that this location is not locked by another transaction ready to commit. `read_item` action must also record the version number of the memory location for later validation purpose.

write_item_step is a `write_item` operation. This operation must follow a `start_txn` action, another `read_item` action, or a `write_item` action, of the same transaction ID.

try_commit_txn_step must follow either a `read_item` operation or a `write_item` operation. We consider only meaningful transactional memory traces. Therefore, we consider it illegal to have a transaction with no reads or writes.

lock_write_item_step is a step before a transaction is ready to commit; therefore, it must follow `try_commit_txn` operation. A transaction is allowed to lock the memory location only if it has writes in it. In addition, it is only allowed to lock the location if there is no lock already placed on that location.

validate_read_item_step must also follow `try_commit_txn` operation. However, when there are writes in a transaction, a transaction must lock the memory location before validating its reads. Therefore, this operation must follow *lock_write_item* action if it exists in the transaction. Validating reads involves checking the version number associated with the memory location.

abort_txn_step occurs when the `validate_read_item` action fails. If the transaction contains writes (i.e., it holds a lock on the memory location), this step will also release the lock so that other transactions can perform writes. An aborted transaction is considered 'dead'. That is, the traditional roll-back can occur, but it will be considered a new transaction by and of itself.

If a lock is successfully obtained (if necessary), and if read validation also succeeds, then a *complete_write_item_step* operation will proceed to increment the version number of the memory location by one. Other transactions in the trace that read the memory location with a different version number (i.e., a smaller version number) will therefore fail their validation.

Once writes are completed and reads are successfully validated, *commit_txn_step* will proceed to complete the transaction. However, before each transaction commits, (in this case, there is only one memory location affected by the commit of a transaction) we first record the sequence of the transactions using `seq_point_step` operation so that we can easily generate a correct serial trace corresponding to the STO trace. we will prove the correctness as in our capstone theorem. We make

sure that `seq_point_step` is always followed by the first `commit_txn` action if there are multiple `commit_txn` actions (to commit multiple memory locations) in a transaction.

3 Project Schedule and Division of Labor

Since it is crucial to make sure that our implementation is correct before proceeding to proofs, all team members meet frequently to collaborate on writing implementation code. However, once implementation is complete and all lemmas and theorems are defined, each team member will take charge of proving a portion of the lemmas. Currently, we have basically finished a low-level implementation of STO in Coq. Our next step is to prove that this low-level implementation is correct (serializable). After that we plan to work on either writing a high-level implementation of STO and constructing a refinement proof, or extending our current implementation to allow transactions to access the entire memory. Table 1 lists our proposed schedule for this work.

Date	Task
End of April 15^{th}	One memory location implementation
End of April 22^{th}	One memory location proof
End of April 29^{th}	Refinement OR multi-memory location implementation
End of May 8^{th}	Refinement OR multi-memory locationn proof

Table 1: Project Schedule

4 Future Work

We have not taken data structures into consideration when deciding whether it is necessary to abort a transaction when possible conflicts arise. However, this is the essence of STO. STO improves the efficiency of transactional memory by taking into account the data structure located in the memory. Hence, future work involves, for example, proving the correctness of transactional memory that hosts a specific data structure (e.g., red-black tree and linked list). We believe one can reuse many of our definitions, lemmas, and theorems to prove such correctness. For instance, one only needs to slightly modify the inductive type `action` to include operations such as checking whether aborting is needed even when there is an invalid read.

5 Conclusion

A Inductive Type of *action*

```
Inductive action:=
|dummy: action
|start_txn: action
|read_item: version -> action
|write_item: value -> action
|try_commit_txn: action
|lock_write_item: action
|validate_read_item: Prop ->
   action
|abort_txn: action
|complete_write_item: version ->
   action
|commit_txn: version -> action
|seq_point: action.
```

B Inductive Type of *sto_trace*

```
Inductive sto_trace : trace ->
   Prop :=
| empty_step : sto_trace []
| start_txn_step: forall t tid,
  trace_tid_last tid t = dummy
  -> sto_trace t
  -> sto_trace ((tid, start_txn)::
     t)
| read_item_step: forall t tid val
    oldver,
  trace_tid_last tid t = start_txn
  \/ trace_tid_last tid t =
     read_item oldver
  \/ trace_tid_last tid t =
     write_item val
  -> check_lock_or_unlock t
  -> sto_trace t
  -> sto_trace ((tid, read_item (
     trace_commit_last t)) :: t)
| write_item_step: forall t tid
   oldval val ver,
  trace_tid_last tid t = start_txn
  \/ trace_tid_last tid t =
     read_item ver
  \/ trace_tid_last tid t =
     write_item oldval
  -> sto_trace t
```

```
  -> sto_trace ((tid, write_item
     val) :: t)
| try_commit_txn_step: forall t
   tid ver val,
  trace_tid_last tid t = read_item
      ver
  \/ trace_tid_last tid t =
     write_item val
  -> sto_trace t
  -> sto_trace ((tid,
     try_commit_txn)::t)
| lock_write_item_step: forall t
   tid,
  trace_tid_last tid t =
     try_commit_txn
  /\ ~ trace_no_writes tid t
  -> check_lock_or_unlock t
  -> sto_trace t
  -> sto_trace ((tid,
     lock_write_item) :: t)
| validate_read_item_step: forall
   t tid,
  (trace_tid_last tid t =
     try_commit_txn /\
     trace_no_writes tid t )
  \/ trace_tid_last tid t =
     lock_write_item
  -> sto_trace t
  -> sto_trace ((tid,
     validate_read_item (
     check_version (
     read_versions_tid tid t) (
     trace_commit_last t) ))::t)
| abort_txn_step: forall t tid,
      trace_tid_last tid t =
         validate_read_item False
      -> sto_trace t
      -> sto_trace ((tid,
         abort_txn) :: t)
| complete_write_item_step: forall
    t tid,
  trace_tid_last tid t =
     validate_read_item True (*
     valid read*)
  /\ ~ trace_no_writes tid t
  -> sto_trace t
  -> sto_trace ((tid,
     complete_write_item (S (
     trace_commit_last t))) :: t)
| seq_point_step: forall t tid ver
   ,
  (trace_tid_last tid t =
     validate_read_item True
        /\ trace_no_writes tid t)
  \/ (trace_tid_last tid t =
     complete_write_item ver
        /\ ~ trace_no_writes tid t
           )
  -> trace_no_commits tid t
  -> sto_trace t
  -> sto_trace ((tid, seq_point)
     :: t)
| commit_txn_step: forall t tid,
  trace_tid_last tid t = seq_point
  -> sto_trace t
  -> sto_trace ((tid, commit_txn (
     trace_commit_complete_last t)
     ) :: t).
```