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 such 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

\[
[(1, \text{start_txn}); (1, \text{read_item} 0);
(2, \text{start_txn}); (2, \text{read_item} 0);
(1, \text{try_commit_txn}); (1, \text{validate_read_item True});
(1, \text{seq_point}); (1, \text{commit_txn} 0); (2, \text{write_item} 4);
(2, \text{try_commit_txn}); (2, \text{lock_write_item});
(2, \text{validate_read_item True}); (2, \text{complete_write_item} 1);
(2, \text{seq_point}); (2, \text{commit_txn} 1)]
\]

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
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

\[
\forall t : \text{sto} \_\text{trace}, \exists t',\quad \text{is} \_\text{sto} \_\text{trace} t' \rightarrow \text{is} \_\text{serial} \_\text{trace} t' \rightarrow \text{equate} t t'.
\]

### 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.
- **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
STO Verification

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.

<table>
<thead>
<tr>
<th>Date</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>End of April 15th</td>
<td>One memory location implementation</td>
</tr>
<tr>
<td>End of April 22nd</td>
<td>One memory location proof</td>
</tr>
<tr>
<td>End of April 29th</td>
<td>Refinement OR multi-memory location implementation</td>
</tr>
<tr>
<td>End of May 8th</td>
<td>Refinement OR multi-memory locations proof</td>
</tr>
</tbody>
</table>

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 \rightarrow action
| write_item : value \rightarrow action
| try_commit_txn : action
| lock_write_item : action
| validate_read_item : Prop \rightarrow action
| abort_txn : action
| complete_write_item : version \rightarrow action
| commit_txn : version \rightarrow action
| seq_point : action.

B Inductive Type of sto_trace

Inductive sto_trace : trace \rightarrow Prop :=
| empty_step : sto_trace []
| start_txn_step: for all t tid, trace_tid_last tid t = dummy
\rightarrow sto_trace t
\rightarrow sto_trace ((tid, start_txn):: t)
| read_item_step: for all t tid val oldver,
trace_tid_last tid t = start_txn
\slash trace_tid_last tid t =
read_item oldver
\slash trace_tid_last tid t =
write_item val
\rightarrow check_lock_or_unlock t
\rightarrow sto_trace t
\rightarrow sto_trace ((tid, read_item (trace_commit_last t)):: t)
| write_item_step: for all t tid
oldval val ver,
trace_tid_last tid t = start_txn
\slash trace_tid_last tid t =
read_item ver
\slash trace_tid_last tid t =
write_item oldval
\rightarrow sto_trace t
STO Verification

\[
\rightarrow \text{sto_trace} \left( (\text{tid}, \text{write_item_val}) :: t \right)
\]

\[
| \text{try-commit_txn_step}: \forall t \text{ tid ver val},
\text{trace_tid_last tid t = read_item ver}
\backslash/ \text{trace_tid_last tid t = write_item val}
\rightarrow \text{sto_trace t}
\rightarrow \text{sto_trace} \left( (\text{tid}, \text{try_commit_txn}) :: t \right)
\]

\[
| \text{lock_write_item_step}: \forall t \text{ tid},
\text{trace_tid_last tid t = try_commit txn}
\backslash/ \text{~trace_no_writes tid t}
\rightarrow \text{check_lock_or_unlock t}
\rightarrow \text{sto_trace t}
\rightarrow \text{sto_trace} \left( (\text{tid}, \text{lock_write_item}) :: t \right)
\]

\[
| \text{validate_read_item_step}: \forall t \text{ tid},
(\text{trace_tid_last tid t = try_commit txn} \backslash/ \text{trace_no_writes tid t})
\backslash/ \text{trace_tid_last tid t = lock_write_item}
\rightarrow \text{sto_trace t}
\rightarrow \text{sto_trace} \left( (\text{tid}, \text{validate_read_item} \text{ check_version (read_versions tid tid t)} \text{ trace_commit_last t}) :: t \right)
\]

\[
| \text{abort_txn_step}: \forall t \text{ tid},
\text{trace_tid_last tid t = validate_read_item False}
\rightarrow \text{sto_trace t}
\rightarrow \text{sto_trace} \left( (\text{tid}, \text{abort_txn}) :: t \right)
\]

\[
| \text{complete_write_item_step}: \forall t \text{ tid},
\text{trace_tid_last tid t = validate_read_item True (* valid read*)}
\backslash/ \text{~trace_no_writes tid t}
\rightarrow \text{sto_trace t}
\rightarrow \text{sto_trace} \left( (\text{tid}, \text{complete_write_item} \text{ S (trace_commit_last t)}) :: t \right)
\]

\[
| \text{seq_point_step}: \forall t \text{ tid ver},
(\text{trace_tid_last tid t = validate_read_item True}
\backslash/ \text{trace_no_writes tid t})
\backslash/ \text{~trace_no_writes tid t}
\rightarrow \text{trace_no_commits tid t}
\rightarrow \text{sto_trace t}
\rightarrow \text{sto_trace} \left( (\text{tid}, \text{seq_point}) :: t \right)
\]

\[
| \text{commit_txn_step}: \forall t \text{ tid},
\text{trace_tid_last tid t = seq_point}
\rightarrow \text{sto_trace t}
\rightarrow \text{sto_trace} \left( (\text{tid}, \text{commit_txn (trace_commit_complete_last t)}) :: t \right)
\]