On the Correctness of Optimistic Composable Data Structures

Ahmed Hassan∗
Virginia Tech
hassan84@vt.edu

Sebastiano Peluso
Virginia Tech
peluso@vt.edu

Roberto Palmieri
Virginia Tech
robertop@vt.edu

Binoy Ravindran
Virginia Tech
binoy@vt.edu

1 Introduction

Concurrent data structures are widely used in multithreading applications as they efficiently enable the exploitation of parallelism especially when deployed on multi-core architectures. Intuitively, the (complex) fine-grained design of a data structure [10, 9, 6, 3], in which only the critical part of an operation is synchronized, provides better performance than the (simpler) coarse-grained design where operations act in mutual exclusion. Moreover, there is a growing interest (e.g., [1, 7, 8, 17]) on composing data structure operations into a single transaction, rather than considering them as standalone atomic operations. This is mainly because Transactional Memory [5] (TM), the recent appealing programming abstraction for developing concurrent applications, has been integrated into commodity hardware chips [15, 2] and well-known compilers (e.g., [16]). This integration allows the usage of in-memory transactions to all programmers, including those non-experts. As one of the consequences of that, a programmer can easily wrap multiple data structure operations into a single atomic transaction, which thus enables composability.

The design of a data structure has its own challenges that depend on its semantics and implementation constraints. That is why, for the last decade, proving the correctness of most concurrent (and composable) data structures followed an ad-hoc approach. This lack of generality contributed to make the task of assessing their correctness very challenging. Recently, we observed an initial step towards accomplishing the goal of having a general model for proving the correctness of concurrent data structures, which is the single writer multiple readers model (we name it SWMR hereafter) presented by Lev-Ari et. al. in [13]. The SWMR model focuses on two safety properties (roughly summarized here): validity, which guarantees that no “unexpected” behaviors (e.g., access to an invalid address or a division by zero) can occur in all the steps of a concurrent execution; and regularity, an extension of the classical regularity model on registers [12] that guarantees that each read-only operation is consistent (i.e., linearized) with all the write operations. The appealing advantages of the SWMR model are that: i) it allows the programmer to use general and well defined terms (i.e., base conditions and base points) to prove the validity and regularity of any data structure fitting the SWMR model; and ii) it gives a formal way to prove linearizability [11] by relying on regularity.

Despite the strengths of the SWMR model, the set of data structures that can actually benefit from it does not include most of the recent highly optimized and practical concurrent [9, 6, 3] and composable [1, 7, 8, 17] data structures which, in addition, allow concurrent writes. Fortunately, those recent data structures have some common design principles that we can isolate. Specifically, in all the former examples, each update operation is split into a read-only traversal phase and a read-write commit phase, and the traversal phase is optimistically executed in isolation from the commit phase (and usually without monitoring its steps), counting on the fact that the output of the traversal phase remains “valid” during the commit phase. Given their optimistic nature, we name them as optimistic data structures.

∗Contact author: Ahmed Hassan 467 Durham Hall, Virginia Tech, Blacksburg VA, USA. Email: hassan84@vt.edu. Tel: +1 (540) 231-5336
In this presentation, we focus on the class of optimistic data structures and we provide a set of models for assessing their correctness so that existing and future practical implementations can rely on that. The overall goal of those models is to provide a general approach (which uses the notion of base points and base conditions as SWMR) for proving the correctness of a set of data structures that allow multiple writers multiple readers (MWMR) executions, which are wider and more practical than the set of data structures fitting the SWMR model.

2 The Single Writer Commit (SWC) Model

As mentioned before, in our models each operation is split into read-only traversal phase and read-write commit phase. This representation is general enough to cover also those operations with either an empty traversal (i.e., operations whose first step is a write) or an empty commit phase (i.e., read-only operations).

We start by presenting the Single Writer Commit (SWC) model, a MWMR model in which both read-only and update operations run concurrently with the restriction that only the commit phases are atomically executed with the Single Lock Atomicity (SLA) semantics [14] (i.e., as if they are executed sequentially). For the sake of simplifying the presentation, we first introduce this model by assuming that the commit phases are protected by a single global lock, then we discuss the case of concurrent commits.

Figure 1 shows an example of this case with five update operations, \( u_{o1}, ..., u_{o5} \), and one read-only operation \( ro \). In this example, the commit phases of all the update operations do not interleave, even if the operations themselves interleave. The read-only operation \( ro \) is concurrent with \( u_{o3}, u_{o4}, \) and \( u_{o5} \). In particular, it inter-leaves with the commit phases of \( u_{o3} \) and \( u_{o4} \), while its commit phase only interleaves with \( u_{o4} \).

![Figure 1: An example of a MWMR concurrent execution (a) that can be executed using our model by converting it to a Single Writer Commit scenario (b).](image)

Figure 2(a) shows how we model a typical optimistic data structure operation. Any operation \( O \) is split into two sequences of steps: \( O^T = s^1 \cdot ... \cdot s^m \); and \( O^C = s^{m+1} \cdot ... \cdot s^n \). The sequence \( O^T \) represents the traversal phase, which does not contain any write step. The sequence \( O^C \) represents the commit phase, which always ends with \( return_O(v_{ret}) \) and can contain both read and write steps. Given that a data structure under the SWC model allows concurrent traversal phases and a single commit phase at a time, the transitions from the shared traversal phase to the exclusive commit phase and vice versa are represented by two auxiliary steps \( s' \) and \( s'' \) (e.g., they can be an acquisition/release of a global lock). We do not assume the presence of such a transition in read-only operations, thus, in those cases, \( s' \) and \( s'' \) are just dummy steps that do nothing. Excluding the auxiliary steps, the commit phase of a read-only operation \( O \) is \( O^C = return_O(v_{ret}) \).

In practice, optimistic data structures usually start the commit phase by a validation mechanism to ensure that the output of the traversal phase remains valid until the transition to the exclusive commit mode; otherwise the traversal phase is re-executed. That is why it is important to include this re-execution mechanism in our model. To do so, we define for each operation \( O \) on a data structure \( ds \) a variable \( u \) that represents the number of unsuccessful trials \( (u \in \{0, 1, ..., \infty\}) \). The value of \( u \) is determined according to the design of \( ds \) and

\[1\] If for an operation \( O \) it is possible to have an execution with \( u = \infty \), this (informally) entails that the operation is not wait-free.
Figure 2: a) Splitting the operation to support concurrent MWMR execution with Single Writer Commit (SWC). $O^T$ is the traversal phase; $O^C$ is the commit phase. $I$: invoke, $r$: read, $w$: write, $R$: return, $L$: lock, $U$: unlock. b) Unsuccessful trials are part of the overall traversal phase in our model.

the concurrent execution $\mu$ that includes $O$. Every unsuccessful trial resets the local state (i.e., the values of the operation’s local variables) of the operation to the initial $\bot$ state before starting the next trial. The commit phases of all the unsuccessful trials are clearly not allowed to write on the shared memory because of their inconsistent local state. As shown in Figure 2(b), the traversal phase of the operation $O$ includes all those unsuccessful trials and the commit phase of $O$ is only the successful commit phase of the last trial ($t_{u+1}^C$).

Under such a model we define two states for each update operation $w$: pre-commit-state$_{uw}$, which represents the local state of $u$ after the auxiliary step $s'$ and before the first real step in the commit phase, $s^{n+1}$; and post-commit$_{uw}$, which is the shared state of the data structure (i.e., the values of its variables) after the last step of $u$. Then we enforce that for every update operation $u_i$ in a concurrent execution $\mu$, pre-commit-state$_{uw}$, observes post-commit$_{uw_{i-1}}$, where $u_{i-1}$ is the update operation whose commit phase precedes the commit phase of $u_i$ in $\mu$.

The main idea of modeling data structures and concurrent executions in this way is that we can redefine the notion of base points and base conditions by following the main idea presented in [13]. Doing so we provide the programmer with a general methodology to prove the correctness of a generic MWMR data structure by identifying the base conditions associated with the steps of its operations.

By correctness here we mean validity, namely the execution of every step on a data structure $ds$ is never subject to “bad behaviors” (e.g., division by zero or null-pointer accesses), and regularity, namely for each history $H$ on $ds$, the sub-history composed of all write operations in $H$ enriched with one read-only operation (if any) in $H$ is linearizable.

2.1 Allowing Concurrent Commits

The implementation of optimistic data structures usually do not rely on a global lock-based mechanism to finalize the writes, but rather, in order to increase the level of concurrency, the commit phase either executes inside TM transactions (hardware or software) [17] [1], or leverages the locking mechanism with fine-grained locks that protect (at least) the written locations [9] [10]. Fortunately, some of those techniques provide the same atomicity guarantees as global locks. For example, some TM implementation provides single lock atomicity (SLA) guarantees [14] (e.g., the HTM transactions provided by Intel’s TSX extensions [15] and the SLA version of NORec [4]). By definition, SLA guarantees that all the non-transactional reads observe the same serialization of all the concurrent transactions. Thus, if those TM are used to execute the commit phases instead of serializing them with a global lock, then we can easily prove that the same guarantees are fulfilled. In fact, in [14] the authors formally prove that executing atomic blocks with SLA semantics is equivalent to executing them using synchronized blocks protected by a single lock, which implies that our model is safe under this new assumption.
3 The Composable Single Writer Commit (C-SWC) Model

We now extend our model by allowing the composition of multiple operations into atomic transactions. For the sake of simplicity, we assume that all the operations belong to the same data structure, and then we remove this assumption by showing that operations on different data structures can be executed in the same transaction under the C-SWC model as long as the data structures are independent.

In the C-SWC model, as shown in Figure 3, each operation $O_i$ is split into traversal ($O^T_i$) and commit ($O^C_i$) phases, and the transaction itself is split into a traversal phase that combines all the traversal phases of the operations (i.e., $T^T = \text{start} \cdot O^T_1 \cdot O^T_2 \cdot \ldots \cdot O^T_k$), and a commit phase that combines all the commit phases, surrounded by two auxiliary steps to move the execution to/from the exclusive mode (i.e., $T^C = S' \cdot O^C_1 \cdot O^C_2 \cdot \ldots \cdot O^C_k \cdot \text{commit} \cdot S''$). Like SWC, we assume for simplicity that commit phases are protected by a single global lock. However, the same arguments adopted in SWC can be applied here to consider concurrent executions under the SLA semantics. We also assume that the commit phases of transactions are the successful ones, and any unsuccessful trial is included in the transaction traversal phase.

Figure 3 shows how operations are split in the C-SWC model. First, the return step of each operation is shifted to be the last step of its traversal phase. This is important because the return value of the operation may be used later in the transaction body. Second, the auxiliary steps $S'$ and $S''$ are removed from the commit phases of operations and they appear only once in the commit phase of the enclosing transaction. Finally, a dummy step $s^{ro-commit}$ is added to the commit phase of any read-only operation $ro$. This dummy step becomes the only one in the commit phase of $ro$ because the real return step is shifted to the traversal phase (as said before).

As shown in Figure 3, we define for each operation $O_i$ one more state called post-traversal-state$_{O_i}$, which is the local state before $O_i$’s return step. We also define for the whole transaction $T$ a state called pre-commit-state$_T$ which is the local state after $s'$.

Analogously to the SWC case, under C-SWC we enforce that for every update transaction $T_i$ in a transactional execution $\mu$, pre-commit-state$_T_i$ observes the post-state$_{T_{i-1}}$, where $T_{i-1}$ is the transaction whose commit phase precedes the commit phase of $T_i$ in $\mu$. By doing so we redefine the notion of base points and base conditions by starting from the definition provided in the SWC model. Therefore we provide the programmer with a general methodology to prove the correctness of a transactional MWMR data structure by identifying the base conditions associated with the steps of its operations.

By correctness here we mean i) validity, namely the execution of every step on a data structure $ds$ is never subject to “bad behaviors” (e.g., division by zero or null-pointer accesses); ii) internal consistency which means that the return steps of the operations in the same transaction observes the same shared state, and can be informally defined as follows: the post-traversal-states of every operation in a transaction $T$ have the same base point as pre-commit-state$_T$; iii) and a variant of regularity applied to transactional histories, and that we can informally define as follows: for each transactional history $H$ on $ds$, the sub-history composed by all the committed update transactions in $H$ plus another transaction in $H$ (i.e., either read-only or update and either live/aborted or committed) is strict serializable.
References


