Optimistic Transactional Boosting

Ahmed Hassan, Roberto Palmieri, Sebastiano Peluso, and Binoy Ravindran

Abstract—The last two decades witnessed the success of many efficient designs of concurrent data structures. A large set of them has a common base principle: each operation is split into a *read-only traversal* phase, which scans the data structure without locking or monitoring, and a *read-write commit* phase, which atomically validates the output of the *traversal* phase and applies the needed modifications to the data structure. In this paper we introduce *Optimistic Transactional Boosting* (OTB), an optimistic methodology for extending those designs in order to support the composition of multiple operations into one atomic execution by building a single *traversal* phase and a single *commit* phase for the whole atomic execution. As a result, OTB-based data structures are *optimistic* and *composable*. The former because they defer any locking and/or monitoring to the commit phase of the entire atomic execution; the latter because they allow the execution of multiple operations atomically. Additionally, in this paper we provide a theoretical model for analyzing OTB-based data structures and proving their correctness. In particular, we extended a recent approach that models concurrent data structures by including the two notions of *optimism* and *composition* of operations.

Index Terms—Composability, Transactional Memory, Concurrent Data Structures, Linearizability.

1 Introduction

The increasing ubiquity of multi-core processors motivated the development of data structures that can exploit the hardware parallelism of those processors. The current widely used concurrent collections of elements (e.g., Linked-List, Skip-List, Tree) are well optimized for high performance and ensure isolation of atomic operations, but they do not *compose*. For example, it is hard to modify the efficient concurrent linked-list designs [1] to allow an atomic insertion of two elements: if the `add` method internally uses locks, as in [2], issues like managing the dependency between operations executed in the same atomic block of code, and avoiding executions with deadlock due to the chain of lock acquisitions, may arise. Similarly, composing non-blocking operations, as those in [3], is challenging because of the need to atomically modify different places of the data structure using only basic primitives, such as CAS operations. Lack of composability is a limitation of well-known data structure designs, especially for legacy systems, as it makes their integration with third-party software difficult. In this paper we focus on providing high performance composable data structures.

Transactional Memory (TM) [4], [5] is a programming paradigm that leverages the *transaction* abstraction to ensure atomicity of a block of code, and it can be used for providing data structures with composable operations. However, monitoring all of the memory locations accessed by a transaction while executing data structure operations results in a significant overhead, mainly due to the occurrence of *false conflicts*. For example, if two transactions attempts to insert two different elements into a linked-list, these two insertions are usually commutative (i.e., they can be executed concurrently without affecting their correctness). However, TM inherently cannot detect this commutativity property because it is not able to capture the semantics of the data structure itself, and can raise a false conflict, restarting the execution of one of them. As a result, TM-based data structures perform inferior to their optimized, concurrent counterparts. Although this overhead exists for both Software Transactional Memory (STM) [6] and Hardware Transactional Memory (HTM) [4], in this paper we focus on software solutions.

Figure 1 shows an example of a false conflicts occurring in a sorted linked-list. By relying on almost all TM implementations (except some, such as [7], [8]), if a transaction $t1$ attempts to insert 55 in the linked-list, it has to keep track of all the memory locations associated with the traversed nodes (gray nodes in the figure). Assume now that a concurrent transaction $t2$ successfully inserts 4 by modifying the link of node 2 to point to node 4 instead of node 5. In this case, $t1$ cannot successfully complete its execution because the tracked nodes 2 and 5 experienced a change made by $t2$ and therefore $t1$ should be restarted. This is a false conflict because the insertion of 55 should not be affected by the concurrent insertion of 4 (i.e., the two operations are commutative). In some cases, like long linked-lists, these false conflicts dominate any other overheads in the system.

Importantly, most of the efficient concurrent (non-composable) linked-lists, such as fine-grained and lock-free linked-lists [1], do not suffer from this false conflict because they are able to exploit data structure semantics.

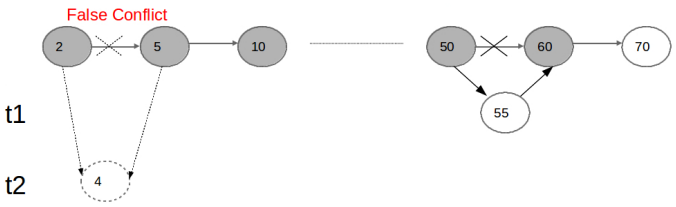


Fig. 1. An example of false conflicts in a sorted linked-list.

In this paper, we provide a methodology for boosting the efficient concurrent data structure designs to be composable while preserving their high performance. To do so, we identify two main challenges to address: *i)* maximizing the number of optimizations in the concurrent version that are not nullified in the composable version due to the generality of the proposed methodology; *ii)*

- *Authors are with the Department of Electrical and Computer Engineering, Virginia Tech, Blacksburg, VA, 24060. Dr. Ahmed Hassan is currently affilieted with University of Alexandria in Egypt.*
- *E-mails: {hassan84,robertop,peluso,binoy}@vt.edu.*

providing a theoretical model for such methodology that helps in analyzing the composable versions of a concurrent data structure and formally proving its correctness. The two challenges listed above form the main objectives of this paper.

Regarding the first objective, we present *Optimistic Transactional Boosting* (OTB), an optimistic methodology for converting concurrent data structures into composable ones. In OTB, data structure operations belonging to the same atomic block of code do not acquire locks and modify the shared data structure at encounter time, as a typical concurrent data structure design would do. Instead, they populate their changes into local logs during their execution, deferring lock acquisitions and modifications until the execution of the whole atomic block is completed. This way, OTB combines the benefits of the optimized design of concurrent data structures; of the custom validation of the composable execution, which aims at verifying the semantics of the data structure itself, as introduced in [9]; and of the optimistic concurrency control of transactional memory.

Following OTB's guidelines, in this paper we show the design and implementation of two different abstract data types: *set*, which is implemented internally using linked-list; and *priority queue*, which is implemented using heap and skip-list. Our experimental results show a significant improvement in the performance of OTB-based data structures over existing (non-optimized) composable implementations (up to 2× better). Solutions attempting to overcome the limitations tackled by OTB are outlined in Section 3. Among them, OTB finds inspiration from Herlihy and Koskinen's *Transactional Boosting* (TB) methodology [9]. This is because TB converts concurrent data structures to composable (also called transactional) ones by providing a *semantic layer* on top of existing concurrent data structures. TB has well-known downsides that limit its applicability (detailed later), while OTB overcomes them by using a more optimistic design.

Our second main contribution is a theoretical framework for proving the correctness of OTB-based data structures. In this paper we did not limit ourselves to provide a formal correctness proof for the presented OTB data structures. Instead, given the essence of OTB, which is proposing guidelines to produce new composable data structures, and the complexity that is always associated with proving their correctness, we decided to formulate a model to help designers reason about the correctness of composable data structures. Specifically, we extend a recent approach that models concurrent data structures [10], [11], and incrementally we extended such a model to be suitable for OTB-based data structures. This second contribution of the paper does not simply enrich the discussion about OTB, but it represents an equally important step forward towards proving the correctness of concurrent and composable data structures.

The rest of the paper is organized as follows. Section 2 details the OTB methodology. Section 3 compares OTB with the other solutions in literature. Section 4 shows how to use OTB to boost different concurrent data structures with different characteristics. Section 5 introduces the theoretical model for OTB-based data structures. We evaluate OTB-Based data structures in Section 6 and conclude in Section 7.

2 Optimistic transactional Boosting

Without loss of generality, we define a concurrent data structure as an abstract data type accessed by a set of *primitive* operations in the form of:

```
ret OP(arg)
```

where `arg` and `ret` abstract the operation's arguments and return values, respectively. Importantly, these primitive operations are assumed to be linearizable on the concurrent data structure.

Optimistic Transactional Boosting (OTB) is a methodology for boosting concurrent data structures to be composable, which means adding the ability to perform more than one of its primitive operations as a single atomic execution. To do that, OTB offers the following new operation to application programmers, which we name *composite*:

```
rets[] Composite-OP(n, ops[], args[])
```

This composite operation includes the invocation of a number of `n` primitive operations. The arrays `ops[]` and `args[]` define those `n` operations with their arguments, and the array `rets[]` stores the return values of those operations. These arrays are assumed to be ordered thus, for instance, the triple ops[0], args[0], and rets[0] depicts the type, arguments, and return value of the first invoked primitive operation; and so forth for all the remaining `n-1` primitive operations. For example, considering the concurrent linked-list-based set described in Section 4.1, the operation `Composite-OP`(2, {"`add`", "`remove`"}, {3, 5}) atomically executes two primitive operations, which are inserting the element 3 into the set and removing the element 5 from it, and returns an array of their return values.

Throughout the paper, we often use the term "transaction" to refer to the `Composite-OP` operation because, similar to the transaction abstraction, `Composite-OP` provides atomic execution of multiple primitive operations. However, under our assumptions any other operation different from the primitive operations of the underlying concurrent data structure is not allowed to be invoked; otherwise, the overall correctness might not be guaranteed. Furthermore, our assumptions do not allow a `Composite-OP` operation to include primitive operations on different data structures. Extending the programming model to allow non-primitive operations to be executed atomically along with primitive operations, and to include operations on multiple data structures in a single `Composite-OP` operation is left as future work. An initial design of such extended programming model has been already presented in [12].

2.1 Methodology

OTB's key observation is that a large set of concurrent data structures (e.g., [1], [2], [13]) has a common feature: data structure operations have an *unmonitored traversal* step, in which the shared data structure is traversed without locking or instrumenting any access of its nodes. To guarantee consistency, this unmonitored traversal is followed by a validation step after acquiring the appropriate locks (to preserve the data structure semantics) and before modifying the shared data structure. We call this set *optimistic data structures* due the optimism in their operation's traversal steps. OTB modifies the design of this class of data structures to support the composition of their operations. Basically, the OTB methodology can be summarized in three main guidelines.

(G1) *Each primitive operation is divided into three steps:*

- *Traversal*. This step scans the data structure, and computes the operation's results (i.e., its postcondition) and what it depends on (i.e., its precondition). This requires us to define what we call *semantic read-set* and *semantic write-set*, which are private sets shared by all primitive operations

of a composite operation. The semantic read-set maintains information of the nodes that need to be monitored in order to preserve the correct linearization of each primitive operation of a composite operation. On the other hand, the semantic write-set is used to record the modifications that should be applied to the shared data structure to make each primitive operation permanent in memory. Note that, we use the term "semantic" because, as opposed to classical TM processing, our semantic read-/write-sets do not store the memory locations read and written during the operation execution; they rather store information needed to preserve the data structure semantics and its linearization.

- *Validation*. This step checks the validity of the preconditions. Specifically, the entities stored in the semantic read-set are validated to ensure that all invoked primitive operations are correctly linearized.
- *Commit*. This step performs the modifications on the shared data structure. Unlike concurrent data structures, this step is not done at the end of each primitive operation. Instead, it is deferred to a single commit step of the enclosing `Composite-OP` operation. All information needed for performing this step are maintained in the semantic write-sets during the traversal step. To publish the write-sets, a classical two-phase locking is used. This locking mechanism done over the semantic write-set preserves data structure semantics in case of concurrent updates, and also prevents conflicts at the memory level.

(G2) *Data structure design is adapted to support linearizable `Composite-OP` operations.* OTB provides the following guidelines to guarantee that a `Composite-OP` operation, which consists of `n` primitive operations, is linearizable.

(G2.1) Each primitive operation scans the local semantic write-set first, before accessing the shared data structure. This is important to include the effect of the earlier (not yet published) primitive operations in the same `Composite-OP` operation.

(G2.2) The semantic read-set is validated during the commit step of the `Composite-OP` operation (which combines the commit steps of all its primitive operations). This step is needed to guarantee that each `Composite-OP` operation observes a consistent state of the system before commit. Optionally, the semantic read-set is also validated after the traversal step of each primitive operation. This validation has two goals. First, it allows the detection of inconsistencies at encounter time rather than at commit time, which might positively impact performance. Second, it may be needed if stronger guarantees than linearizability are required, such as opacity [14] or LS-linearizability [15].

(G2.3) During the commit step of a `Composite-OP` operation, locks on all entries of the semantic write-set are acquired before performing any physical modification on the shared data structure.

(G2.4) Modifications required by primitive operations are applied during the commit step of the enclosing `Composite-OP` operation in the same order as they appeared in the `Composite-OP` operation itself. In case the outcome of a primitive operation influences the subsequent primitive operations recorded in the semantic write-set, the entries of those operations are updated accordingly.

(G2.5) Results of all primitive operations have to be validated, even if the original (concurrent) operation does not make any validation (like `contains` operation in set, as we show in Section 4.1). The goal of validation in these cases is to ensure that each operation's result is still the same at the commit step of the enclosing `Composite-OP` operation.

(G3) *Data structure design is adapted for more optimizations.* Each data structure can be further optimized according to its own semantics and implementation. For example, in set, if an item is added and then deleted in a `Composite-OP` operation, both operations eliminate each other and can be completed without physically modifying the shared data structure itself.

Unlike the first two guidelines, which are general for any optimistic data structure, the way to deploy the third guideline varies from one data structure to another. It gives a hint to the developers that now data structures are no longer used as a black box, and further optimizations can be applied to improve performance. It is worth to note that the generality of the first two guidelines does not entail that they can be applied "blindly" without being aware of the specific data structure's semantics and how it is linearized. In fact OTB performs better than the naive TM-based data structures mainly because it exploits semantics. We believe OTB's guidelines make a clear separation between the general outline that can be applied to any optimistic data structure (like validation in *G2.2*, and commit in *G2.4*, even if the validation/commit mechanisms themselves vary from one data structure to another) and the optimizations that are related to the specific data structure implementation.

2.2 OTB Interface

We now develop the aforementioned guidelines to produce a set of APIs that can be used to implement a `Composite-OP` operation. Although the implementation of those APIs still depends on the semantics of each data structure, they allow developers to follow a more rigorous procedure while designing new OTB data structures. Moreover, these APIs can be used as an abstraction for different implementations, which allows for a better reusability and maintenance of the code. For example, replacing the linked-list implementation of OTB-set with a tree implementation would require minimal changes in the application's source code thanks to the new APIs. In Section 4, we introduce case studies on how to use those APIs to develop OTB sets and priority queues.

We present the APIs in the same order they should be invoked to implement a `Composite-OP` operation. The first step when such an operation is called is to execute the traversal phases of its primitive operations in sequence. Those traversal phases should implement the following interface:

```
ret PRIMITIVE-OP-TRAVERSAL(OP, arg,
        read-set, write-set)
```

According to *G1* and *G2.1*, `PRIMITIVE-OP-TRAVERSAL` has three steps: scanning the semantic write-set, traversing the data structure (similar to the concurrent version), and then recording the primitive operation's postconditions in the semantic read/write-sets. That is why it is required to pass the operation's semantic read/write-sets as (by-reference) arguments. According to guideline *G2.2*, an optional forth step in `PRIMITIVE-OP-TRAVERSAL` is the *post-validation* of the semantic read-set, which implements the following interface:

```
bool VALIDATE(read-set)
```

After executing the traversal phases of all the primitive operations, the commit procedure of the enclosing `Composite-OP` operation is invoked, using the following interface:

```
ret COMPOSITE-OP-COMMIT(read-set, write-set)
```

As dictated by guidelines *G2.2–G2.5*, this procedure starts by acquiring locks over entities in the semantic write-set. After that, the semantic read-set is validated (using the same aforementioned interface), then the semantic write-set is published, and finally locks are released. To avoid deadlock, any failure during the lock acquisition implies aborting and retrying the whole `Composite-OP` operation (by releasing all acquired locks). Also, any failure during validation (either after traversal or at commit) forces the `Composite-OP` operation to restart its execution from the beginning.

3 Related Work

3.1 Transactional Boosting

Transactional Boosting (TB) [9] is a methodology to convert concurrent data structures to transactional ones by deploying a *semantic layer* that prevents non-commutative operations from operating concurrently. TB has downsides that limit its applicability. Those downsides are raised because in TB: *i)* the abstract lock acquisition and modifications in memory are eager (i.e., they happen encounter time), and *ii)* the technique uses the underlying concurrent data structure as a black box. Although the latter helps in analyzing the transactional version and prove its correctness without the need to know how the concurrent version is designed, optimizations in the original concurrent data structures may be nullified. In addition to that, TB requires the existence of an inverse operation for each primitive operation in order to rollback its execution in case the transaction is aborted.

3.2 Optimistic Semantic Synchronization

A set of recent methodologies leverage the same idea of OTB: dividing the transaction execution into phases and optimistically executing some of them without any instrumentation (also called *unmonitored* phases). We use the term *Optimistic Semantic Synchronization* (OSS) to refer this set of methodologies. The word *optimistic* is because all of these solutions share a fundamental optimism by having the above unmonitored phases. In this section, we overview some of those approaches.

Consistency Oblivious Programming (COP) [16], [17], [18] splits the operations into the same three phases as OTB (but under different names). We observe two main differences between COP and OTB. First, COP is introduced mainly to design concurrent data structures and it does not inherently provide composability unless changes are made at the hardware level [18]. Second, COP does not use locks at commit. Instead, it enforces atomicity and isolation by executing both the *validation* and *commit* phases using STM [16] or HTM [17] transactions.

Partitioned Transactions (ParT) [19] also uses the same trend of splitting the operations into a *traversal* (called *planning*) phase and a *commit* (called *update*) phase, but it gives more general guidelines than OTB. Specifically, ParT does not restrict the planning phase to be a traversal of a data structure and it allows this phase to be any generic block of code. Also, ParT does not obligate the planning phase to be necessarily unmonitored, as in OTB and COP. Instead, it allows both the planning and update phases to be transactions. ParT's generality is also its major obstacle: at the beginning of the completion phase, an application-specific validator has to be defined by the programmer to validate the output of the planning phase.

3.3 Other Related Approaches

In addition to OSS, other works proposed different ways to implement transactional data structures other than the traditional use of TM algorithms. One direction is to adapt STM algorithms to allow the programmer to control the semantics of the data structures. Examples of trials in this direction include elastic transaction [8], open nesting [20], [21], and early release [22]. However, those approaches are focused on adapting TM frameworks more than designing composable data structures. That is why it becomes the obligation of the programmer to (practically) design and (theoretically) model complex data structures implemented using those approaches.

Another direction is to use TM as support to design libraries of data structures. Examples in this direction include transactional predication [23], and speculation friendly red-black tree [24]. The main downside of those approaches is that they address specific data structures, namely maps in the former and trees in the latter. OTB aims at providing a more general methodology.

In [25], Spiegelman et al. presented a library of *transactional* data structure implementations. Although this work shares many OTB's motivating factors and design principles, it lacks providing a general methodology and optimization guidelines to enable the composability of other data structures.

3.4 Contrasting OTB with Existing Solutions

OTB-based composable data structures generally perform better than those implemented using TM. This is because, unlike the classical meaning of read-sets and write-sets in STM (and also HTM, if we consider the L1 cache as an internal read-set and write-set, which is the case in [26]), not all memory reads and writes are saved in the semantic read-sets and write-sets. Instead, only those reads and writes that affect the linearization of the data structure are saved. This avoids suffering from *false conflicts*.

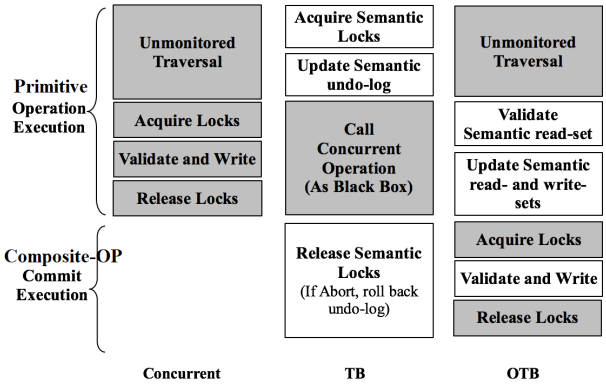


Fig. 2. Execution flow of: concurrent data structures; Transactional Boosting (TB); and OTB. In both TB and OTB, the upper part is called n time (for the n primitive operations included in the `Composite-OP` operation), and then the lower part is called once to commit the `Composite-OP` operation.

To compare OTB with TB, Figure 2 shows the execution flow of concurrent (optimistic) data structures, TB, and OTB. Concurrent (non-composable) data structures yield high performance because they traverse the data structure without instrumentation, and they only acquire locks (or use CAS operations in case of lock-free data structures) at late phases. To add composability, TB acquires semantic locks while executing the data structure operation (i.e., eagerly), and saves the inverse operations in an undo-log to rollback the execution in case of abort. Then, it uses the underlying concurrent data structure as a black box without any modifications. (In both TB and OTB, dark blocks in Figure 2 are the same as the concurrent versions, while white blocks are added/modified.) At commit time, the only task to be accomplished is to release semantic locks because operations have already been applied eagerly.

In contrast to TB, OTB acquires semantic locks only while committing the whole atomic execution of `Composite-OP` (i.e., lazily), and uses the underlying data structure as a white box. Similar to concurrent optimistic data structures, OTB traverses data structures without instrumentation. However, it differs from them in three aspects: *i)* lock acquisition and actual writes are shifted to commit time; *ii)* the validation procedure is modified to satisfy the new composability requirements; and *iii)* the necessary information is saved in local semantic read-sets and write-sets.

Thus, OTB gains the following benefits over TB. First, it does not require well defined commutativity rules or inverse operations. Second, it uses highly concurrent collections as white boxes to design new composable versions of each concurrent (non-composable) data structure. This allows for further optimizations with some re-engineering overhead.

4 OTB Data Structures

In this section we present two types of optimistically boosted data structures: set and priority queue. These were specifically chosen as they represent two different categories:

- *Commutable Objects*. In set, operations are commutative at the level of element keys. In other words, two operations are commutative if they access two different keys in the set.
- *Non-commutable Objects*. In priority queue, operations are commutative at the level of the whole data structure. This means that, even if two operations access two different elements in the queue, they cannot execute in parallel. In fact, any `removeMin` operation is non-commutative with another `removeMin` operation as well as any `add` operation of elements that are smaller than the removed minimum.

4.1 Set

Set is a collection of elements accessed using three primitive operations: `bool add(int)`, `bool remove(int)`, and `bool contains(int)`, with their familiar meanings [1]. No duplicate elements are allowed, therefore `add` (respectively, `remove`) returns false if the element is already (respectively, is not) present in the structure. All primitive operations on different elements of the *set* are commutative – i.e., two operations `add(x)` and `add(y)` are commutative if $x \neq y$. Moreover, two `contains` operations on the same element are commutative as well. Such a high degree of commutativity between primitive operations enables fine-grained semantic synchronization.

4.1.1 Concurrent Set

The first step towards implementing OTB-Set is to select a concurrent (non-composable) version of set that efficiently implements its primitive (`add`, `remove`, and `contains`) operations. Here we overview lazy linked-list [2] as our candidate concurrent version, and in the following sections we show how to design its composable version using OTB. It is worth noting that in addition to the linked-list-based OTB-Set presented here, we also used OTB to boost skip-list-based and tree-based OTB-Set. Details about these versions can be found in [27], [28], [29].

Lazy linked-list [2] is an efficient implementation of concurrent *set*. For write operations, the list is traversed without any locking until the involved nodes are locked. If those nodes are still valid after locking, the write takes place and then the nodes are unlocked. A `marked` flag is added to each node for splitting the deletion phase into two steps: the logical deletion phase, which simply sets the flag to indicate that the node has been deleted, and the physical deletion phase, which changes the references to skip the deleted node. This flag prevents traversing a chain of deleted nodes and returning an incorrect result.

Among the existing concurrent linked-list implementations, we selected the lazy linked-list because, in addition to its effectiveness, it provides the following properties that fit the OTB principles. First, it uses a lock-based technique for synchronizing the operations, which simplifies the applicability of the OTB methodology. Second, its operations start with traversing the data structure without any locking or instrumentation, allowing the separation of an unmonitored traversal phase as required by OTB.

4.1.2 Non-Optimized OTB-Set

One of the main advantages of OTB is that it uses the underlying (optimistic) data structure as a white-box, which allows for more data structure-specific optimizations. In general, decoupling the boosting layer from the underlying concurrent data structure is a trade-off. Although, on the one hand, considering the underlying data structure as a black-box means that there is no need to re-engineer its implementation; on the other hand, it does not allow customizing its implementation and thus exploiting the new composable specification of OTB, especially when the re-engineering is an affordable task. For this reason, as showed in Section 2, we decided to split the re-engineering efforts (required by OTB) into two steps: one general (concluded in OTB guidelines *G1* and *G2*); and one more specific per data structure (concluded in *G3*).

In this section, we follow guidelines *G1* and *G2* to design the composable, yet non-optimized, version of lazy linked-list by implementing the three APIs introduced in Section 2.2: `PRIMITIVE-OP-TRAVERSAL`, `VALIDATE`, and `COMPOSITE-OP-COMMIT`. We discuss them separately in Algorithms 1, 2, and 3, respectively.

In Algorithm 1, as mentioned in Section 2.2, `PRIMITIVE-OP-TRAVERSAL` is split into four parts:

- *Local writes check*. Since writes are buffered, each operation on an element α checks the last operation in the semantic write-set on α, and returns the corresponding result. In case there is no previous operation on α in the semantic write-set, then the operation starts traversing the linked-list.
- *Traversal*. This step is the same as in the concurrent version of lazy linked-list.
- *Logging the reads and writes*. Similar to the concurrent version of lazy linked-list, each primitive operation on OTB-Set involves

Algorithm 1 OTB-Set: `PRIMITIVE-OP-TRAVERSAL`

```
1:  procedure PRIMITIVE-OP-TRAVERSAL(op, x, read-set, write-set)
                                              ▷ Step 1: search local write-sets
2:      if x ∈ write-set then
3:          ret = write-set.get-ret(op,x)
4:          if op is add or remove then
5:              write-set.append(op,x)
6:          return ret
                                                        ▷ Step 2: Traversal
7:      pred = head and
8:         curr = head.next
9:      while curr.item < x do
10:         pred = curr
11:         curr = curr.next
                                          ▷ Step 3: Save reads and writes
12:     read-set.add(new ReadSetEntry(pred,curr,op))
13:     if op is add or remove then
14:         write-set.add(new WriteSetEntry(pred,curr,op,x))
                                                  ▷ Step 4: Post Validation
15:     if ¬ VALIDATE(read-set) then
16:         ABORT
17:     else
18:         if successful operation return true else false
19: end procedure
```

two nodes at commit time: $pred$, which is the node storing the largest element less than the searched one, and $curr$, which is the node storing the searched element itself or the smallest element larger than the searched one. Information about these nodes are logged into semantic read-set and write-set with the purpose of using them at commit time. In particular, each semantic read-set or write-set entry contains the two involved nodes in the operation and the type of the operation. The semantic write-set entry contains also the new element to be inserted in case of a successful `add` operation.

- *Post-Validation*. At the end of the traversal step of a primitive operation, the involved nodes are stored in local variables (i.e., $pred$ and $curr$). At this point, according to *G2.2*, the semantic read-set is validated to ensure that the execution of `Composite-OP` does not observe an inconsistent snapshot.

Algorithm 2 shows the `VALIDATE` procedure (used in both post-validation and commit-time-validation). The validation of each semantic read-set entry is similar to the one in the concurrent version of lazy linked-list: both $pred$ and $curr$ should not be deleted, and $pred$ should still be linked to $curr$ (lines 6-8). According to *G2.5* of OTB guidelines, `contains` operation has to perform the same validation as `add` and `remove`, although it is not needed in the concurrent version. This is because any modification made by other `Composite-OP` operations after invoking the `contains` operation and before committing the enclosing `Composite-OP` operation may invalidate the returned value of the operation, breaking the linearizability of the enclosing `Composite-OP` operation.

To enforce isolation, validation also ensures that its accessed nodes are not locked by another `Composite-OP` operation during validation. This is achieved by implementing locks as *sequence locks* (i.e., locks with version numbers). Before the validation, the versions of the locks are recorded if they are not acquired. If some are already locked, the validation fails (lines 2-5). Finally, the validation procedure ensures that the actual locks' versions match the previously recorded versions (lines 9-12).

Algorithm 3 shows the `Composite-OP-COMMIT` operation on OTB-Set. If all primitive operations in `Composite-OP` are read-only (i.e., `contains`), there is nothing to do during commit (line 2). Otherwise, according to point *G2.3*, the appropriate locks are first acquired using CAS operations. Like the concurrent

Algorithm 2 OTB-Set: `VALIDATE`.

```
1:  procedure VALIDATE(read-set)
2:      for all entries in read-sets do
3:          get snapshot of involved locks
4:          if one involved lock is locked then
5:              return false
6:      for all entries in read-sets do
7:          if pred.deleted or curr.deleted or pred.next ≠ curr then
8:              return false
9:      for all entries in read-sets do
10:         check snapshot of involved locks
11:         if version of one involved lock is changed then
12:             return false
13:     return true
14: end procedure
```

version of lazy linked-list, any `add` operation only needs to lock $pred$, while `remove` operations lock both $pred$ and $curr$.

After the lock acquisition, the `VALIDATE` procedure is called in the same way as the above *Post-Validation* to ensure that the semantic read-set is still consistent. If not, the execution of `Composite-OP` is aborted.

The commit procedure ends by publishing writes to the shared linked-list, and then releasing the acquired locks. This step is not straightforward because each node may be involved in more than one primitive operation. In this case, the semantic write-set entries (i.e., $pred$ and $curr$) of these operations should be updated accordingly (lines 17-19 and 23-27)

Algorithm 3 OTB-Set: `Composite-OP-COMMIT`.

```
1:  procedure COMMIT
2:      if write-set.isEmpty then
3:          return
4:      for all entries in write-set do
5:          if CAS Locking pred (or curr if remove) failed then
6:              ABORT
7:      if ¬ VALIDATE(read-set) then
8:          ABORT
9:      for all entries in write-sets do
10:         curr = pred.next
11:         while curr.item < x do
12:             pred = curr
13:             curr = curr.next
14:     if operation = add then
15:         n = new Node(item)
16:         n.locked = true; n.next = curr; pred.next = n
17:         for all entries in write-set do
18:             if entry.pred = pred then
19:                 entry.pred = n
20:     else                                                  ▷ remove
21:         curr.deleted = true
22:         pred.next = curr.next
23:         for all entries in write-set do
24:             if entry.pred = curr then
25:                 entry.pred = pred
26:             else if entry.curr = curr then
27:                 entry.curr = curr.next
28:     for all entries in write-sets do
29:         unlock pred (and curr if remove)
30: end procedure
```

4.1.3 Optimized OTB-Set

In this section we show some optimizations for the linked-list-based version presented in Section 4.1.2. More optimizations on both this version and the other (skip-list-based and tree-based) versions can be found in [27], [28], [29].

Unsuccessful `add` and `remove`. The `add` and `remove` operations are not necessarily considered as writing operations, because duplicated elements are not allowed in the *set*. For example, if an `add` operation returns false, it means that the element to insert already exists in the *set*. To commit such operation,

it is just required to check that the element still exists in the *set*, which allows unsuccessful `add` operations to be treated as successful `contains` operations. This way, the commit operation of `Composite-OP` does not have to acquire any lock for this primitive operation. The same idea can be applied on the unsuccessful `remove` operation which can be treated as an unsuccessful `contains` operation during commit.

Accordingly, in OTB-Set both `contains` and unsuccessful `add`/`remove` operations are treated as *read-only* operations, which add entries only to the semantic read-set and do not acquire any lock during commit. Only successful `add` and `remove` operations are considered *update* operations because they add entries to both the semantic read-set and the write-set, and thus they acquire locks during commit.

Eliminating operations. To handle read-after-write hazards, each primitive operation starts by checking the semantic write-set of the enclosing `Composite-OP` operation before traversing the linked-list. During this step, for improving OTB performance, if some primitive operation in a `Composite-OP` adds an element α and some subsequent operation of the same `Composite-OP` removes the same element α (or vice versa), we allow those operations to locally eliminate each other. This elimination is done by removing both entries from the semantic write-set, which means that the two operations will not make any physical modification on the linked-list. No entry in the semantic read-set is locally eliminated because, this way, the commit time-validation can still be performed on those operations in order to preserve `Composite-OP`'s correctness.

4.2 Priority Queue

Priority queue is a collection of elements whose keys are totally ordered and duplicates are allowed. It provides three primitive operations: `bool add(int)`, `int min()`, and `int removeMin()`, with the familiar meanings [1]. In addition to the well-known heap implementation of priority queue, skip-list has also been proposed for implementing concurrent priority queue [1]. The only difference in semantics between the two implementations is that the former allows duplicate elements, while the latter forbids them. Although both implementations have the same logarithmic complexity, skip-list does not need periodic re-balancing, which is more suited for concurrent execution. Generally, cooperative operations, such as re-balancing, increase the possibility of conflict and decrease concurrency.

4.2.1 TB-based Priority Queue

Herlihy and Koskinen's TB-based priority queue uses a concurrent heap-based priority queue [30]. A global readers/writer lock is used on top of this priority queue to maintain its semantics. An `add` operation acquires a read lock, while `getMin` and `removeMin` operations acquire a write lock. Thus, all `add` operations will be concurrently executed because they are semantically commutative. Global locking is mandatory here, because the `removeMin` operation is not commutative with either another `removeMin` operation or an `add` operation of an element with a smaller key.

Algorithm 4 shows the flow of TB-based priority queue's operations (more details are in [9]). It is important to notice that the inverse of the `add` operation is not defined by most priority queue implementations. This is one of the drawbacks of TB, which cannot be implemented without defining an inverse for each operation. A work-around to this problem is to encapsulate each node in a holder class with a boolean `deleted` flag to mark rolled-back `add` operations (line 4). The `removeMin` operation keeps polling the head until it reaches a non-deleted element (lines 8-10). This adds greater overhead to the boosted priority queue.

Algorithm 4 TB-based priority queue.

```
1:  procedure ADD(x)
2:      readLock.acquire
3:      concurrentPQ.add(x)
4:      undo-log.append(holder(x).deleted = true)
5:  end procedure

6:  procedure REMOVEMIN
7:      writeLock.acquire
8:      x = concurrentPQ.removeMin()
9:      while holder(x).deleted = true do
10:         x = concurrentPQ.removeMin()
11:     undo-log.append(add(x))
12: end procedure
```

TB uses the underlying priority queue as a black box, whose internal design can use any structure other than heap (as long as it carries out the same semantics). This means that, the gains from using skip-list (if used) might be nullified by the eager semantic lock acquisition policy. For example, since TB does not open the black box, if the underlying concurrent priority queue uses fine-grained locking to enhance performance, this optimization will be nullified by the coarse-grained semantic locking when non-commutative operations execute concurrently. OTB, on the other hand, inherits these benefits of using skip-list, and avoids eager coarse-grained locking. Since skip-list does not have a re-balance phase, we can use it to implement an OTB-based priority queue. However, before we show this version, we quickly describe how to extend TB to implement semi-optimistic heap-based priority queue in the following section.

4.2.2 Semi-Optimistic Heap Implementation

A semi-optimistic implementation of a heap-based priority queue is achieved by using the following three optimizations on top of the TB implementation (these optimizations are illustrated in Algorithm 5):

i) The `add` operations are not pessimistically executed until the first `removeMin` or `getMin` operation has occurred. Before that, all `add` operations are saved in a local semantic write-set. Once the first `removeMin` or `getMin` operation occurs, the write lock is acquired and all the `add` operations stored in the semantic write-set are published before executing the new `removeMin` operations. If only `add` operations occur, they are published at commit time after acquiring the read lock. This way, semi-optimistic boosting attempts to acquire the lock only once (either for read or for write).

ii) Since the global lock holder cannot be aborted, there is no need to keep the operations in the semantic write-set anymore. This is because, no operation takes effect on the shared priority queue while the global lock is taken. Moreover, this optimization leads to another advantage, which is relaxing the obligation to define an inverse operation for the `add` operation. This way, the overhead of encapsulating each node in a holder class is avoided.

iii) There is no need for thread-level synchronization after acquiring the write lock, because it guarantees an exclusive access to the underlying priority queue. This means that sequential,

rather than concurrent, `add`, `getMin`, and `removeMin` operations can be used.

Algorithm 5 Semi-optimistic heap-based priority queue.

```
1:  procedure ADD(x)
2:      if lock holder then
3:          PQ.add(x)
4:      else
5:          redo-log.append(x)
6:  end procedure
7:  procedure REMOVEMIN
8:      Lock.acquire
9:      for entries in redo-log do
10:         PQ.add(entry.item)
11:     x = PQ.removeMin()
12: end procedure
```

The same idea of our enhancements has been used before in the TML algorithm [31] for memory-based transactions. In TML, a transaction keeps reading without any locking and defers acquiring the global lock until the first write occurs (which maps to the first `removeMin` operation in our case). Then, it blocks any other transaction from committing until it finishes its execution.

Although these optimizations diminish the effect of global locking, this priority queue implementation still cannot be considered as optimistic because `removeMin` and `getMin` operations acquire the global write lock before committing (this is why we call it a "semi-optimistic" approach). In the next section we show how implement an OTB-based priority queue using a skip-list.

4.2.3 Skip-List OTB Implementation

In this section, we show how OTB guidelines can be used to design a composable version of the concurrent skip-list-based priority queue presented in [1]. As we mentioned earlier, the only difference in semantics between this version and the heap-based version is that this version does not allow duplicate elements.

Unlike previously done for OTB-Set, we now skip the non-optimized version of the OTB priority queue and we discuss the optimized version (OTB-PQ hereafter) directly. OTB-PQ extends OTB-Set to support the priority queue's three primitive operations (i.e., `add`, `min`, and `removeMin`) rather than OTB-Set's primitive operations. To do that, we extend our skip-list-based OTB-Set, presented in [28], in order to preserve the logarithmic time complexity of operations. However, in the rest of this section, to simplify the presentation we assume that OTB-PQ extends the exact linked-list-based OTB-Set presented in Section 4.1.

Slight modifications are made on the extended OTB-Set to ensure priority queue properties. Specifically, each thread saves a local variable, called $lastRemovedMin$, which refers to the last element removed by the `Composite-OP` operation. It is mainly used to identify the next element to be removed if the `Composite-OP` operation calls another `removeMin` operation (note that all these primitive operations do not physically change the underlying list until the commit step of `Composite-OP` is called). Thus, this variable is initialized as the sentinel head of the list. Additionally, each thread saves a local sequential priority queue, in addition to the local semantic read-/write-sets, to simplify the handling of read-after-write cases.

Algorithm 6 shows the `PRIMITIVE-OP-TRAVERSAL` procedure of OTB-PQ. The other two procedures, `VALIDATE` and `COMPOSITE-OP-COMMIT`, just call their corresponding procedures in the underlying OTB-Set. As for `PRIMITIVE-OP-TRAVERSAL`, if the primitive operation is an `add` operation, it calls `PRIMITIVE-OP-TRAVERSAL` of the underlying OTB-Set's `add` operation (i.e., shown in Algorithm 1). If it is a successful `add`, it saves the added element in the local sequential priority queue (line 4). If the operation is a `removeMin`, it compares the minimum elements in both the local and shared priority queues and removes the lowest (line 11). Whether the minimum is selected from the local or the shared priority queue, `Composite-OP` has to validate that the shared minimum is not changed later by any concurrent `Composite-OP` operation. To achieve that, lines 12 and 18 call `PRIMITIVE-OP-TRAVERSAL` of the underlying list's `contains` and `remove` operations, respectively, in order to implicitly add this shared minimum to the local semantic read-set (to be validated later).

Before returning the minimum, a further validation is invoked to ensure that the shared minimum is still linked by its $pred$ (lines 14 and 20). Then, $lastRemovedMin$ is updated (line 22). A similar procedure (not shown in Algorithm 6) is used for the `getMin` operation.

Algorithm 6 OTB-PQ: `PRIMITIVE-OP-TRAVERSAL`

```
1:  procedure PRIMITIVE-OP-TRAVERSAL(op,x,read-set,write-set)
2:      if op = ADD then
3:          if list.PRIMITIVE-OP-TRAVERSAL(ADD,x,read-set,write-set) then
4:              localPQ.add(x)
5:              return true
6:          else
7:              return false
8:      else if op = REMOVEMIN then
9:          localMin = localPQ.getMin()
10:         sharedMin = lastRemovedMin.next
11:         if localMin < sharedMin then
12:             if ¬ list.PRIMITIVE-OP-TRAVERSAL(CONTAINS, sharedMin,
    read-set, write-set) then
13:                 Abort
14:             if lastRemovedMin.next ≠ sharedMin then
15:                 Abort
16:             return localPQ.removeMin()
17:         else
18:             if ¬ list.PRIMITIVE-OP-TRAVERSAL(REMOVE, sharedMin, read-
    set, write-set) then
19:                 Abort
20:             if lastRemovedMin.next ≠ sharedMin then
21:                 Abort
22:             lastRemovedMin = sharedMin
23:             return sharedMin
24: end procedure
```

Using this approach, a `Composite-OP` operation accessing (list-based) OTB-PQ does not acquire any lock until its commit phase. Unfortunately, the same approach cannot be easily used in a heap-based implementation because of its complex and cooperative re-balancing mechanism.

One of the main advantages of this optimistic implementation is that the `getMin` operation does not acquire any locks. It is worth noting that, even with our enhancements on the heap-based implementation, TB enforces `getMin` to acquire a write lock, thereby becoming a blocking operation, even for commutative `add` or `getMin` operations.

5 Modeling OTB-Based Data Structures

As mentioned in the introduction of this paper, proving the correctness of concurrent and composable data structures, as OTB, is a hard task, always done manually by designers without a formal support. In this section we present a theoretical framework (or model) to help designers in proving correctness of OTB-based data structures[1].

1. For completeness, we included in the supplemental material a manual correctness proof (i.e., a proof that has not be derived using the framework proposed in this section) for the OTB data structures presented in Section 4.

In Section 5.1 we introduce some definitions and formalisms that will be used subsequently while developing the model. Note that, these definitions do not contradict the terminology used during the explanation of OTB in the previous sections; however they are necessary to be introduced at this stage to provide a sound theoretical ground to prove the correctness of OTB-based data structures. The new presented framework has been designed upon a baseline framework to prove correctness of concurrent (non-composable) data structure called *Single Writer and Multiple Readers* (*SWMR*), which is overviewed in Section 5.2. In Section 5.3 we detail our new framework, which can be used to prove correctness of a class of OTB-based data structures that prevents concurrent execution of writer's commit phases. In Section 5.4 we provide an initial proposal towards having a general model suited for OTB-based data structures by relaxing the above assumptions.

5.1 Definitions

We provide a formal definition of a data structure for the purpose of producing a model to prove its correctness. We define a data structure as a set of shared variables $X = \{x_1, x_2, \dots x_i, \dots\}$ where operations can be invoked on. When applying our model to OTB data structures, the same definition of an operation holds for both primitive operations and `Composite-OP` operations.

An operation execution (or just *operation*) O, is a sequence of $Steps_O = s_O^1 \cdot s_O^2 \cdot ... \cdot s_O^n$, where: s_O^1 is the invocation of the operation, i.e., $invoke_O$, and s_O^n is the operation's $return$, i.e., $return_O(v_{ret})$. A dummy $return(void)$ step is added for any operation that does not have an explicit return value. Any other step is either $read_O(x_i)$ or $write_O(x_i, v_w)$ (those steps comply with their common meaning)[2]. Steps are assumed to be executed atomically. An operation is called *read-only* if it does not execute any *write* step; otherwise it is called *update*. A sequential execution of a data structure ds is a sequence of non-interleaving operations on ds. A *concurrent execution* μ is a sequence of interleaved steps of different operations.

The history H of a concurrent execution μ on a data structure ds, $H|\mu$, is the subsequence of μ with only the *invoke* and the *return* steps. A *pending* operation in H is an operation with no $return$ step. With *complete(H)* we indicate the sub-history of H that has no pending operations. We say that two operations are concurrent if the return step of one does not precede the invoke step of the other one. A history H is sequential if no two operations in H are concurrent.

Any data structure ds has a *sequential specification*, which corresponds to the set of all the allowed sequential histories on ds. A history H is *linearizable* [32] if it can be extended (by appending zero or more *return* steps) to some history H' that is equivalent (i.e., has the same operations and *return* steps) to a legal (i.e., satisfies the sequential specification) sequential history S such that non-interleaving operations in H' appear in the same order in S.

The *shared state* s of a data structure is defined at any time by the values of its shared variables, and it is selected from a set of shared states $\mathcal{S}$. Each operation has a *local state* l, selected from a set of local states $\mathcal{L}_{op}$, which is defined by the values of its local variables. The sets $\mathcal{S}$ and $\mathcal{L}_{op}$ contain initial states $\mathcal{S}_0$ and $\perp_{op}$, respectively. A *step* in the execution of each operation represents a transition function on $\mathcal{S}$ and $\mathcal{L}_{op}$ that changes the *shared state* of the data structure and the *local state* of the operation from $\langle l, s\rangle$ to $\langle l', s'\rangle$. At any point of a concurrent execution μ, if we have n pending operations, we will have n local states (one for each operation) and one shared state s.

Given a data structure ds and an operation O in a concurrent execution μ on ds, we define $pre\text{-}state_O$ as the shared state of ds right before $invoke_O$, and $post\text{-}state_O$ as the shared state of ds right after $return_O(v_{ret})$. We also say that a *shared state* is *sequentially reachable* if it can be reached by some sequential execution of ds.

5.2 Background: the SWMR model

Lev-Ari et al. in [10] proposed an initial step towards having a general model for proving the correctness of concurrent (non-composable) data structures. This model considers data structures that allow concurrency among read operations only, thus we name it *single writer and multiple readers* (*SWMR*) hereafter. 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 [33] that guarantees that each read-only operation is consistent (i.e., linearized) with all the write operations. A recent extension of *SWMR* allows concurrent writers [11] (thus, we name it *MWMR* hereafter). The appealing advantage of those two models is that they allow the programmer to use general and well-defined terms to prove *validity*, *regularity*, and *linearizability* [32] of any concurrent data structure. In the subsequent sections, we adapt those models to better fit the notion of OTB-based data structure.

The *single writer multiple reader (SWMR)* model assumes *concurrent executions* on a data structure ds where the steps of two *update* operations in any concurrent execution μ on ds do not interleave. Conversely, a *multiple writer multiple reader (MWMR)* model is the one that allows such an interleaving. In the following we report the definition of *base condition* and *base point* as defined in the *SWMR* model of [10]. Note that, unlike in [10], in our models those definitions apply to all operation and not only to read-only operations.

- *Base condition*. Given a local state l of an operation O on a data structure ds, a base condition ϕ for l is a predicate over the shared state of ds where every sequential execution of O starting from a shared state s such that $\phi(s) = true$ reaches l. A base condition for a step s^i (named also ϕ^i) is the base condition for the local state right before the execution of s^i.
- *Base point*. An execution of a step s^i, with a local state l, in a concurrent execution μ has a base point if there is a sequentially reachable *post-state* s such that the base condition $\phi^i(s)$ holds.

The combination of the two definitions makes an interesting conclusion: if an execution of a step s^i in an operation O has a base point in a concurrent execution μ, this means that there is a sequentially reachable *post-state* from which O can start and reach s^i with the same local state l. That also means that the execution of s^i in μ would have been the same as performed in a sequential execution. Accordingly, if every step in every concurrent execution of ds has a base point, then we say that ds is *valid*. Informally, that means that ds is never subject to "bad behaviors" (e.g., division by zero or null-pointer accesses) because every step *observes* a sequentially reachable local state.

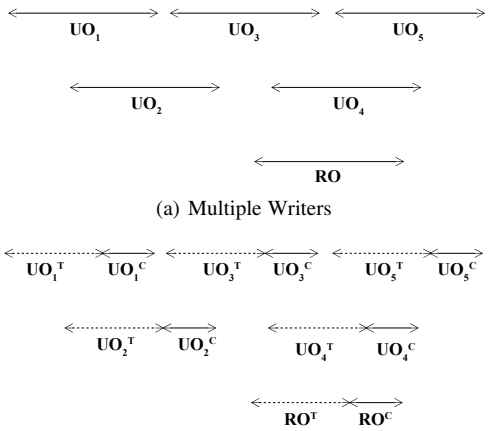
2. We define later in Section 5.3 two more steps called S' and S'' that represent lock acquisition/release. Unlike $read$ and $write$ steps, those steps appear only at the commit phase of an update operation and cannot appear anywhere else, as we detailed later.

The *SWMR* model names a data structure ds as *regular* if, for each history H and every read-only operation ro in H (if any), the sub-history composed of all write operations in H and ro is linearizable. Leveraging the definition of *base point*, a concurrent execution μ is regular if, for every *read-only* operation ro in μ, the *base points* of ro's *return* step is the *post-state* of either an *update* operation executed concurrently with ro in μ or the last *update* operation that ended before ro's *invoke* step in μ. Those candidate *base points* are called *regularity base points*.

5.3 The Single Writer Commit (SWC) Model

In this section we present the Single Writer Commit (SWC) model, an adapted version of the *SWMR* model in which both *read-only* and *update* operations run concurrently with the restriction that only the writing phases of *update* operations (i.e., *commit* phases) are executed sequentially. Note that, this restriction needs to be relaxed in order to meet OTB-based data structures' design; more details are in Section 5.4.

Figure 3 shows an example of an execution under the *SWC* model with five *update* operations, $uo_1, \ldots, uo_5$, 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 uo_3, uo_4, and uo_5. In particular, it interleaves with the commit phases of uo_3 and uo_4, while its *commit* phase only interleaves with uo_4.



(a) Multiple Writers

(b) Single Writer Commit

Fig. 3. An example of a concurrent execution (a) that can be executed using our model by converting it to a *single writer commit* scenario (b).

Algorithm 7 shows a practical (and simple) data structure implementation under the *SWC* model: a linked-list with three operations $readLast$, $insertLast$, and $removeLast$ (which gives the semantics of stacks). The head of the list is assumed to be constant (i.e., the widely used sentinel node). Although all the executions of $readLast$ (respectively $insertLast$) are *read-only* (respectively *update*), some execution of $removeLast$ (those that return at line 27) are *read-only* and some others (those that return at line 38) are *update*. Unlike the *SWMR* model, *SWC* does not categorize operations in a concurrent execution μ as *read-only* or *update* a priori, but rather it assigns the operation's type considering its actual execution in μ, which therefore increases the level of concurrency. Accordingly, in Algorithm 7, *removeLast* is not treated as an *update* operation when the linked-list is empty.

Figure 4(a) shows how we model an operation in a OTB-based data structure. Any operation O is split into two sequences of steps: $O^T = s^1 \cdot \ldots \cdot s^m$; and $O^C = s^{m+1} \cdot \ldots \cdot s^n$. 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 transition from the *shared traversal* phase to the *exclusive commit* phase is represented by an auxiliary steps S', and the end of the commit phase is represented by another auxiliary step S''. For example, S' and S'' can represent an acquisition and a release of a global lock as in Algorithm 7. 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 $O^C = return_O(v_{ret})$.

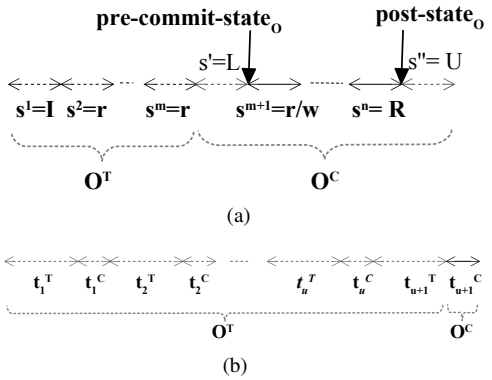


Fig. 4. *a)* Splitting the operation to support concurrent MWMR execution with single writer commit (SWC). O^T is the *traversal* phase; O^C is the *commit* phase. I:*invoke*, r:*read*, w:*write*, R:*return*, L:*lock*, U:*unlock*. *b)* Unsuccessful trials are part of the overall *traversal* phase in our model.

Based on OTB's guideline (G2.2), the *commit* phase starts 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. To include this re-execution mechanism in our model, we define for each operation O on a data structure ds a variable u that represents the number of *unsuccessful trials* ($u \in \{0, 1, ..., \infty\}$)[3]. The value of u is determined according to the concurrent execution μ that includes O. Every unsuccessful trial resets the local state of the operation to the initial $\perp$ state before starting the next trial. The commit phases of all the unsuccessful trials are clearly not allowed to write on the shared data structure because of their inconsistent local state. As shown in Figure 4(b), the *traversal* phase of the operation O includes all those unsuccessful trials $t_1 \cdot t_2 \cdot \ldots \cdot t_u$, and the *commit* phase of O is only the successful *commit* phase of the last trial ($t_{u+1}{}^C$).

In Algorithm 7, the *commit* phase of $readLast$ is always successful (in fact, the operation itself is wait-free [34]). Thus, it is easy to identify the *traversal* phase (the whole execution before line 7), and the *commit* phase (the *return* step at line 7). Identifying *insertLast*'s phases is more difficult because it may have unsuccessful *commit* phases. According to our definitions, the *traversal* phase of an operation O with u unsuccessful trials is a concatenation of u executions of lines 10–18 in which the condition of line 16 is false, followed by one execution until right before line 15. The *commit* phase is formed by lines 15–20 in

3. For an operation O, if it is possible to have an execution with $u = \infty$, in practice it entails that the operation is not wait-free.

Algorithm 7 A linked list with three operations implemented under the *SWC* model.

```
1:  procedure READLAST
2:      last ←⊥
3:      next ← read(head.next)                          ▷ φ1 : true
4:      while next ≠⊥ do
5:          last ← next
6:          next ← read(last.next)                      ▷ φ2 : head ⇒* last
7:      return(last)                                    ▷ φ3 : head ⇒* last
8:  end procedure

9:  procedure INSERTLAST(n)
10:     last ←⊥
11:     next ← read(head.next)                          ▷ φ4 : true
12:     while next ≠⊥ do
13:         last ← next
14:         next ← read(last.next)                      ▷ φ5 : head ⇒* last
15:     lockAcquire(gl)
16:     if read(last.next) ≠⊥ or last.deleted then      ▷ φ6 : head ⇒* last
17:         lockRelease(gl)
18:         go to 10
19:     write(last.next, n)
20:     lockRelease(gl)
21: end procedure

22: procedure REMOVELAST
23:     last ←⊥
24:     secondlast = read(head)                         ▷ φ7 : true
25:     next ← read(head.next)                          ▷ φ8 : true
26:     if next =⊥ then
27:         return                                      ▷ φ9 : head.next =⊥
28:     while next ≠⊥ do
29:         secondLast = last
30:         last ← next
31:         next ← read(last.next)                      ▷ φ10 : head ⇒* last
32:     lockAcquire(gl)
33:     if read(last.next) ≠⊥ or last.deleted then      ▷ φ11 : head ⇒* last
34:         lockRelease(gl)
35:         go to 23
36:     write(last.deleted, true)
37:     write(secondlast.next, ⊥)
38:     lockRelease(gl)
39: end procedure
```

which the condition of line 16 is true. The phases of the read-only (respectively update) executions of *removeLast* are determined similar to *readLast* (respectively *insertLast*).

The definitions of *base conditions*, *base points*, and *validity* are similar to the *SWMR* model, but in the *SWC* model they are defined for both *read-only* and *update* operations. However, the definitions of *regularity base point* and *regularity* need to be refined. Specifically, in the *SWMR* model, *update* operations are linearizable because they are executed sequentially. However, in the *SWC* model, this is not trivially guaranteed because an *update* operation may be invalidated before performing its exclusive *commit* phase. To avoid that, in our model we first define a new state for each update operation uo, called $pre\text{-}commit\text{-}state_{uo}$, which represents the local state of uo after the auxiliary step s' and before the first real step in the *commit* phase, s^{m+1}. Then, we guarantee the linearization of the *update* operations as follows.

Definition 1. ***(Executions under SWC)*** *In a concurrent execution μ with k update operations whose commit phases are sequential, those k operations are totally ordered according to the order of their commit phases $\{u_1 \prec_c u_2 \prec_c ... \prec_c u_k\}$. A dummy u_0 operation is added such that $post\text{-}state_{u_0}$ is $\mathcal{S}_0$[4]. A concurrent execution μ is under the SWC model if update operations have sequential commit phases, and for every update operation u_i in μ, $post\text{-}state_{u_{i-1}}$ is a base point for $pre\text{-}commit\text{-}state_{u_i}$.*

Hereafter, we focus on those executions under the *SWC* model according to Definition 1. Theorem 1 shows that in those executions, *update* operations are linearizable (the complete proofs of all the presented theorems are in the supplemental material).

Theorem 1. *Given a concurrent execution μ with k completed update operations, and the corresponding history of those k operations $H_k|\mu$, if μ is under SWC model, all the post-states of the k operations are sequentially reachable, and $H_k|\mu$ is linearizable.*

The intuition of the proof is that the operation that commits first trivially produces a *sequentially reachable* state. Based on Definition 1, at commit time each operation observes the $post\text{-}state$ of the operation right before it. Then, by induction, all operations produce *sequentially reachable* states. Therefore, *update* operations are linearized according to their *commit* phases order.

Based on Theorem 1, in any concurrent execution μ with k completed *update* operations, we can identify $k+1$ *sequentially reachable* shared states that would be the candidate *base points* for each step in μ. Those points are the $post\text{-}states$ of the k completed *update* operations, in addition to the initial state $\mathcal{S}$. Next, we refine the definition of *regularity base points* as follows:

Definition 2. ***(Regularity base points under SWC)*** *A base point bp of a step s^i in a read-only operation ro of a concurrent execution μ under the SWC model is a regularity base point if bp is the post-state of either an update operation whose commit phase is executed concurrently with ro in μ or of the update operation whose commit phase is the last one completed before ro's invoke step in μ (the initial state is the default).*

This definition simply restricts the candidate *regularity base points* of any *read-only* operation to be the *post-state* of the operations with interleaving *commit* phases rather than those of the interleaving *update* operations. For example, in Figure 3 the *post-state* of uo_3 and uo_4 are candidate *regularity base points* for ro's steps, while uo_5 is excluded because its *commit* phase starts after ro's return point (uo_5 is not excluded in the original *regularity* in [10]). Also, the definition uniquely identifies one *update* operation among those committed before ro (uo_2 in our example). uo_1 is excluded because its *commit* phase is not the last one before ro's invocation. Because the commit phases of update executions do not interleave, this candidate *regularity base point* for ro's steps is always deterministic.

Finally, we define the meaning of *regular execution* under *SWC* in Theorem 2. Intuitively, the theorem states that a concurrent execution is regular if *i)* every step in every *read-only* operation and in the *traversal* phase of every *update* operation is *valid* (i.e., all operations execute without any "unexpected" behavior); and *ii)* the history of the *update* operations plus one *read-only* operation is *linearizable* (recall that Theorem 1 already proves that the set of the *update* operations is *linearizable*).

Theorem 2. *A concurrent execution μ under the SWC model is regular if:*

1) In the traversal phase of every operation in μ, every step has

4. This dummy operation is added only to cover the case of the initial state.

a base point with some base condition.

2) *The $pre\text{-}commit\text{-}state$ of every read-only operation in μ has a regularity base point with some base condition.*

Definition 3. ***(Regular data structures under SWC)*** *A data structure ds is regular if every concurrent execution on ds is regular under the SWC model.*

We applied SWC on the data structure in Algorithm 7 by: defining the appropriate base conditions, which are listed to the right of each operation, and using Definition 3 and Theorem 2 to prove that this data structure is *regular* through Theorem 3.

Theorem 3. *The linked-list in Algorithm 7 is regular.*

5.4 Applying SWC to OTB-Based data structures

The SWC model is an initial step towards our main goal, which is modeling OTB-based data structure. In this section, we discuss the missing steps towards reaching that goal and how we address them. Those steps can be summarized as follows:

- In order to model a concurrent data structure using SWC, its sequential specification has to be defined. As we discussed in Section 2, an OTB-based data structure is a composable version of an existing concurrent data structure. While the sequential specification of the latter (i.e., concurrent) is assumed to be given, the sequential specification of the former (i.e., composable) is not defined.
- Under SWC model, the commit phases of update operations are assumed to be sequential, while operations in OTB-based data structures allow concurrent commit phases protected by the two phase locking, as stated by guideline *G2.3*. To model OTB-based data structures, the assumption of having sequential commit phases has to be relaxed.
- SWC can be used to prove that a concurrent data structure is regular, while an extension to SWC is needed to prove *linearizability* in addition to regularity.

In Section 5.4.1, we address the first point by introducing a formal way to define the sequential specification of composable OTB-based data structures. In Sections 5.4.2 and 5.4.3 we informally discuss how to exploit the recent MWMR model [11] to address the last two points. The formal adaptation of SWC is left as a future work.

5.4.1 Sequential Specification of Composable OTB-based data structures

As introduced in Section 2, a composable OTB-based data structure can be seen as a concurrent data structure with a single `Composite-OP` operation instead of a set of primitive operations. This way, no modification is needed in SWC to model the composable version. The challenge is therefore shifted to define the sequential specification of this composable data structure. In Lemma 1, we introduce a formal way to inherently define this sequential specification based on the sequential specification of the concurrent data structure boosted by OTB.

We first introduce the following terms that are used in the lemma: $ds\_concurrent$ represents the concurrent data structure boosted using OTB; $ds\_composable$ represents the corresponding composable version; P represents a primitive operation on $ds\_concurrent$; C represents the `Composable-OP` operation on $ds\_composable$; $C.P$ is an array that represents (in order) the primitive operations of C; and $C.size$ is the size of $C.P$.

Lemma 1. *For every sequential history $H = < P_1, ..., P_n >$ in the sequential specification of $ds\_concurrent$, all the histories of the form $H' = < C_1, ..., C_k > (k = 1, ..., n)$ where:*
$< C_1.P[1], ..., C_1.P[C_1.size], ..., C_k.P[1], ..., C_k.P[C_k.size] >$ $= < P_1, ..., P_n >$ are in the sequential specification of $ds\_composable$

Informally, the lemma states that a sequential history of C operations is in the sequential specification of $composable\_ds$ if the corresponding history of P operations, where P are the primitive operations that compose (in order) the C `Composite-OP` operations, is in the sequential specification of $concurrent\_ds$.

5.4.2 Relaxing the Single Writer Commit Assumption.

The implementations of OTB-based data structures typically 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 leverages a combination of two-phase locking (*G2.3*) and commit-time-validation (*G2.2*). Interestingly, the *MWMR* model presented in [11] already extended the original *SWMR* model to cover the aforementioned point. In this section we summarize those extensions and show how the same intuitions can be used to extend our *SWC* model. We believe that formalizing this intuition is a straightforward extension of the definitions and theorems in Section 5.3, following the same roadmap in [11].

Authors in [11] showed that using two-phase locking mechanism at commit time[5] provides guarantees that they proved to be sufficient for having linearizable *update* operations. Those guarantees are summarized as follows:

- All the commit phase steps of an *update* operation *commute* with all the commit phase steps of any interleaving *update* operation.
- Every step in the commit phase of any *update* operation is *base point preserving* with respect to any interleaving *update* operation. Briefly, a step s is *base point preserving* with respect to an operation O if it satisfies the following condition: the shared state before executing s is a base point for any step s' in O if and only if the shared state after executing s is also a base point for s'.
- The shared state observed before the first write in the commit phase of an operation is a base point for all the write steps as well as the return step of that commit phase.

5.4.3 Replacing Regularity with Linearizability

In order to prove *linearizability* instead of *regularity*, we need to additionally prove that the return steps of all *read-only* operations observe the same serialization of *update* operations. In [11], the authors did so by adding one more condition: *"Every update operation has at most one step that is not base point preserving with respect to all read-only operations."*. Informally, this condition means that every *update* operation has a single step that changes the abstract state of the data structure, which nominates that step to be the linearization point of the operation. Interestingly, the authors of [11] already proved that for the concurrent version of lazy linked-list [2]. As a future work, we plan to extend that for our composable version presented in Section 4

6 Evaluation

In this section, we evaluate the performance of OTB-Set, OTB-PQ, and the semi-optimistic priority queue. The experiments were

5. The authors used the term *critical sequence* instead of *commit phase*. However both terms have the same meaning in OTB-based data structures.

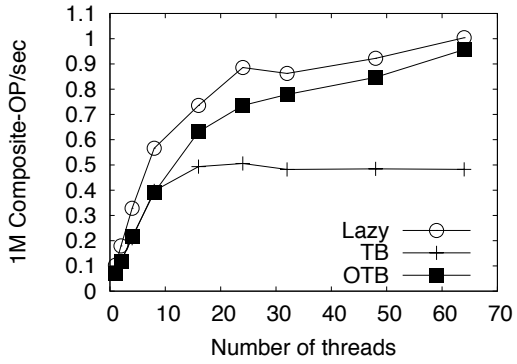


Fig. 5. Throughput of skip-list-based set with 64K elements, and five primitive operations, 80% writes and 20% reads, per `Composite-OP`.

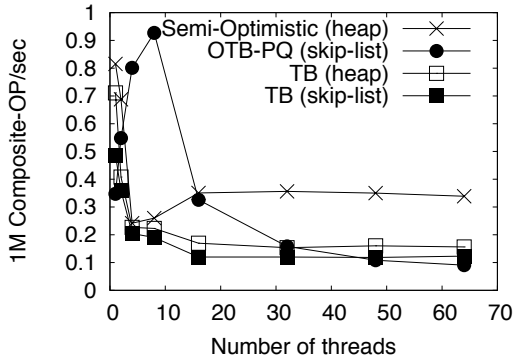


Fig. 6. Throughput of *priority queue* with 10K nodes and five primitive operations, 50% `add` and 50% `removeMin`, per `Composite-OP`.

conducted on a 64-core machine, which has four AMD Opteron processors, each with 16 cores running at 1400 MHz and 16KB of L1 data cache. The total memory installed is 32 GB. Threads start execution with a warm up phase of 2 seconds, followed by an execution of 5 seconds, during which the throughput is measured. Each plotted data-point is the average of five runs.

Set. A comprehensive experimental study on all OTB-Set implementations (linked-list, skip-list, and tree) has been already shown in [28], [29]. In this section, for the sake of completeness, we report one of those experiments that showed the typical trend of OTB-based data structures. We compared the skip-list-based OTB-Set against lazy *set* [2] and TB *set* [9]. Recall that the lazy *set* is not capable of running multiple primitive operations atomically (i.e., it is a concurrent data structure, not composable). We only show it as a rough upper bound for both OTB-Set and TB.

For a fair comparison, both lazy and TB *sets* are also skip-list-based. We used a skip-list of size 64K and a workload of 80% write operations and 5 primitive operations per each `Composite-OP` operation. Also, to conduct an evenhanded experiment the percentage of the writes is made to be the percentage of the successful ones, because, as we mentioned in Section 4.1.3, an unsuccessful `add/remove` operation is considered as a read operation. Roughly speaking, in order to achieve that, the range of elements is made large enough to ensure that most `add` operations are successful. Also, each `remove` operation takes an element added by the previous `Composite-OP` operations as a parameter, such that it will likely succeed. Also, the number of `add` and `remove` operations are kept equal to avoid significant fluctuations of the data structure size during the experiment. We measured throughput as the number of successful Composite-OP operations per second.

Figure 5 plots the results of the experiment. Overall, OTB-Set performs close to the (upper bound) performance of the lazy *set* and up to 2× better than TB. This is mainly because, considering the contention level of the workload, TB's eager locking mechanism is ineffective and a more optimistic algorithm, such as OTB-Set, is preferable.

Priority Queue. Figure 6 shows priority queue results. Regarding heap-based priority queues, we used Java atomic package's priority queue as underlying concurrent data structure in the TB implementation, and we adapted it for our semi-optimistic implementation. For skip-list priority queues, both TB and OTB-PQ boosted the skip-list implementation described in [1]. The results show that the performance of TB is almost the same regardless of the underlying implementation, which confirms our claims about the effect of using the underlying data structures as a black box. The results also illustrate how our three optimizations (described in Section 4.2.2) enhance the performance of the heap-based priority queue and allow our semi-optimistic priority queue to saturate on a higher throughput than TB. Finally, OTB-PQ is better than its corresponding (skip-list-based) TB in almost all the cases, except for the high contention case (more than 48 threads). In fact, OTB-PQ achieves the best performance with respect to all other algorithms (both heap-based and skip-list-based) for small number of threads. This improvement is achieved at the cost of a slightly lower performance when the number of concurrent threads increases. This is expected, and reasonable for optimistic approaches in general, given that the gap in performance for high contention cases is limited also considering that priority queue is a non-commutative data structure.

7 Conclusion

This paper has two main contributions. First, we presented Optimistic Transactional Boosting (OTB), a novel methodology for boosting concurrent data structures to be composable. We deployed OTB on a number of concurrent data structures with different characteristics, and we showed that their performance is better than competitors providing composable executions. Second, we provided a theoretical model for OTB-based data structures. This model is an extension of a recent model for concurrent data structures that includes the two notions of *optimism* and *composability*.

Acknowledgments

Authors would like to thank the anonymous IEEE TPDS reviewers for the valuable comments. This work is partially supported by Air Force Office of Scientific Research (AFOSR) under grant FA9550-14-1-0187.

References

[1] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming, Revised Reprint*. Elsevier, 2012.

[2] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. Scherer, and N. Shavit, "A lazy concurrent list-based set algorithm," *Principles of Distributed Systems*, pp. 3–16, 2006.

[3] T. L. Harris, "A pragmatic implementation of non-blocking linked-lists," in *Distributed Computing, 15th International Conference, DISC 2001, Lisbon, Portugal, October 3-5, 2001, Proceedings*, 2001, pp. 300–314.

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

[5] D. Dice, O. Shalev, and N. Shavit, "Transactional locking II," in *Distributed Computing*. Springer, 2006, pp. 194–208.

[6] N. Shavit and D. Touitou, "Software transactional memory," *Distributed Computing*, vol. 10, no. 2, pp. 99–116, 1997.

[7] M. M. Saad, R. Palmieri, A. Hassan, and B. Ravindran, "Extending TM primitives using low level semantics," in *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2016, Asilomar State Beach/Pacific Grove, CA, USA, July 11-13, 2016*, 2016, pp. 109–120.

[8] P. Felber, V. Gramoli, and R. Guerraoui, "Elastic transactions," in *Proceedings of the 23rd International Symposium on Distributed Computing (DISC)*. Springer, 2009, pp. 93–107.

[9] M. Herlihy and E. Koskinen, "Transactional boosting: a methodology for highly-concurrent transactional objects," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. ACM, 2008, pp. 207–216.

[10] K. Lev-Ari, G. Chockler, and I. Keidar, "On correctness of data structures under reads-write concurrency," in *Distributed Computing - 28th International Symposium, DISC 2014, Austin, TX, USA, October 12-15, 2014. Proceedings*, 2014, pp. 273–287.

[11] ——, "A constructive approach for proving data structures' linearizability," in *Distributed Computing - 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9, 2015, Proceedings*, 2015, pp. 356–370.

[12] A. Hassan, R. Palmieri, and B. Ravindran, "Integrating transactionally boosted data structures with stm frameworks: A case study on set," in *9th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT)*, 2014.

[13] T. Crain, V. Gramoli, and M. Raynal, "A contention-friendly binary search tree," in *Euro-Par 2013 Parallel Processing - 19th International Conference, Aachen, Germany, August 26-30, 2013. Proceedings*, 2013, pp. 229–240.

[14] R. Guerraoui and M. Kapalka, "On the correctness of transactional memory," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. ACM, 2008, pp. 175–184.

[15] V. Gramoli, P. Kuznetsov, and S. Ravi, "In the search for optimal concurrency," in *SIROCCO 16: 23rd International Colloquium on Structural Information and Communication Complexity, Helsinki, Finland, July 19-21, 2016, Revised Selected Papers*, 2016, pp. 143–158.

[16] Y. Afek, H. Avni, and N. Shavit, "Towards consistency oblivious programming," in *Proceedings of the 15th International Conference on Principles of Distributed Systems*, ser. OPODIS'11, 2011.

[17] H. Avni and B. Kuszmaul, "Improve htm scaling with consistency-oblivious programming," *TRANSACT 14: 9th Workshop on Transactional Computing.*, March, 2014.

[18] H. Avni and A. Suissa-Peleg, "Brief announcement: Cop composition using transaction suspension in the compiler," in *Proceedings of the 28th International Conference on Distributed Computing*, ser. DISC '14, 2014, p. 550552.

[19] L. Xiang and M. L. Scott, "Software partitioning of hardware transactions," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, San Francisco, CA, USA, February 7-11, 2015*, 2015, pp. 76–86.

[20] Y. Ni, V. S. Menon, A.-R. Adl-Tabatabai, A. L. Hosking, R. L. Hudson, J. E. B. Moss, B. Saha, and T. Shpeisman, "Open nesting in software transactional memory," in *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 2007, pp. 68–78.

[21] B. D. Carlstrom, A. McDonald, M. Carbin, C. Kozyrakis, and K. Olukotun, "Transactional collection classes," in *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 2007, pp. 56–67.

[22] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III, "Software transactional memory for dynamic-sized data structures," in *Proceedings of the twenty-second annual symposium on Principles of distributed computing*. ACM, 2003, pp. 92–101.

[23] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun, "Transactional predication: high-performance concurrent sets and maps for stm," in *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*. ACM, 2010, pp. 6–15.

[24] T. Crain, V. Gramoli, and M. Raynal, "A speculation-friendly binary search tree," in *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*. ACM, 2012, pp. 161–170.

[25] A. Spiegelman, G. Golan-Gueta, and I. Keidar, "Transactional data structure libraries," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, 2016, pp. 682–696.

[26] J. Reinders, "Transactional synchronization in Haswell," http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell/, 2013.

[27] A. Hassan, R. Palmieri, and B. Ravindran, "Optimistic transactional boosting," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, Orlando, FL, USA, February 15-19, 2014*, 2014, pp. 387–388.

[28] ——, "On developing optimistic transactional lazy set," in *Principles of Distributed Systems - 18th International Conference, OPODIS 2014, Cortina d'Ampezzo, Italy, December 16-19, 2014. Proceedings*, 2014, pp. 437–452.

[29] ——, "Transactional interference-less balanced tree," in *Distributed Computing - 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9, 2015, Proceedings*, 2015, pp. 325–340.

[30] G. C. Hunt, M. M. Michael, S. Parthasarathy, and M. L. Scott, "An efficient algorithm for concurrent priority queue heaps," *Inf. Process. Lett.*, vol. 60, no. 3, pp. 151–157, 1996.

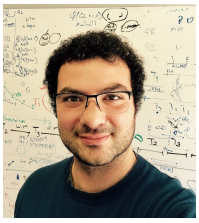
[31] L. Dalessandro, D. Dice, M. L. Scott, N. Shavit, and M. F. Spear, "Transactional mutex locks," in *Euro-Par 2010 - Parallel Processing, 16th International Euro-Par Conference, Ischia, Italy, August 31 - September 3, 2010, Proceedings, Part II*, 2010, pp. 2–13.

[32] M. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, 1990.

[33] L. Lamport, "On interprocess communication. part II: algorithms," *Distributed Computing*, vol. 1, no. 2, pp. 86–101, 1986.

[34] M. Herlihy, "Wait-free synchronization," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 1, pp. 124–149, 1991.

Ahmed Hassan is a Postdoctoral Research Associate in the ECE Department at Virginia Tech. He received his BSc degree in computer science and his MSc degree in computer engineering at Alexandria University, Egypt. He received his PhD degree in Computer Engineering at Virginia Tech. His research interests include transactional memory, concurrent data structures, and distributed computing.



Roberto Palmieri is Research Assistant Professor in the ECE Department at Virginia Tech. He received his BSc in computer engineering, MS and PhD degree in computer engineering at Sapienza, University of Rome, Italy. His research interests include exploring concurrency control protocols for multicore architectures, cluster and geographically distributed systems, with high programmability, scalability, and dependability.

Sebastiano Peluso received the MS degree in computer engineering from Sapienza University of Rome and the PhD degree in computer engineering from Sapienza University of Rome and Instituto Superior Tecnico, Universidade de Lisboa. He is a Research Assistant Professor in the ECE Department at Virginia Tech. His research interests lie in the area of distributed systems and parallel programming, with focus on scalability and fault tolerance of transactional systems.

Binoy Ravindran is a Professor of Electrical and Computer Engineering at Virginia Tech, where he leads the Systems Software Research Group, which conducts research on operating systems, run-times, middleware, compilers, distributed systems, fault-tolerance, concurrency, and real-time systems. Ravindran and his students have published more than 220 papers in these spaces, and some of his group's results have been transitioned to the DOD. His group's papers have won the best paper award at 2013 ACM ASP-DAC, 2012 USENIX SYSTOR (student paper), and selected as one of The Most Influential Papers of 10 Years of ACM DATE (2008) conferences. Dr. Ravindran is an Office of Naval Research Faculty Fellow and an ACM Distinguished Scientist.