On Developing Optimistic Transactional Lazy Set

[**Technical Report**]

Ahmed Hassan, Roberto Palmieri, Binoy Ravindran

Virginia Tech
{hassan84;robertop;binoy}@vt.edu

Abstract. Transactional data structures with the same performance of highly concurrent data structures enable performance-competitive transactional applications. Although Software Transactional Memory (STM) is a promising technology for designing and implementing transactional applications, STM-based transactional data structures still perform inferior to their optimized, concurrent (i.e. non-transactional) counterparts. In this paper, we present OTB-Set, an efficient optimistic transactional lazy set based on both linked-list and skip-list implementations. We first provide general guidelines to show how to design a transactional (non-optimized) version of the highly concurrent lazy set with a minimal reengineering effort. Subsequently, we show how to make specific optimizations to the implementations of the OTB-Set for further enhancing its performance. We also prove that our OTB-Set provides linearizable individual operations and opaque transactions. Our experimental study on a 64-core machine reveals that OTB-Set outperforms competitors in most workloads.

Keywords: Software Transactional Memory, Semantic, Set Data Structure, Boosting

1 Introduction

The increasing ubiquity of multi-core processors motivates 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*. This is a significant limitation from a programmability standpoint, especially for legacy systems as they are increasingly migrated onto multicore hardware (for high performance) and must seamlessly integrate with third-party libraries.

Software transactional memory (STM) [20] can be used to implement transactional data structures (e.g., [7,12]), which makes them composable – a significant benefit. However, monitoring all of the memory locations accessed by a

transaction while executing data structure operations is a significant (and often unnecessary) overhead. As a result, STM-based transactional collections perform inferior to their optimized, concurrent (i.e. non-transactional) counterparts.

As an alternative to STM, the transactional boosting methodology was introduced in [14] and further investigated in [11], to convert the highly concurrent data structures into transactional ones. Briefly, in [14], semantic locks are pessimistically acquired at early phases of the transaction to reduce false conflicts. For this reason, and following the trend in [11], we name this approach as *pessimistic transactional boosting* (or PTB). In contrast, the work in [11] lazily acquires the semantic locks, which motivates the name *optimistic transactional boosting* (or OTB). In both approaches, operations are saved in either semantic undo logs (in PTB) or semantic redo logs (in OTB) to correctly commit/abort transactions. As discussed in [11], OTB has benefits over PTB. First, OTB does not require defining inverse operations. Second, it uses the same phases of validation and commit, with the same semantics, as in STM systems, allowing an easy integration of OTB data structures with STM frameworks. Finally, it uses the underlying data structure as a white box, which allows further data structure-specific optimizations. Inspired by the general OTB's principles, in this paper we focus on set-based data structures providing an efficient transactional lazy set (called OTB-Set hereafter), which boosts the highly concurrent lazy set described in [13]. OTB-Set offers the implementation of linked-list and skip-list.

We split the design of OTB-Set into two phases. The first phase consists of: *i)* dividing each operation of the original lazy concurrent data structure into three steps (*traversal*, *validation*, and *commit*); *ii)* deferring the commit step to the end of the transaction; *iii)* modifying the validation step to guarantee opacity [9] rather than linearizability [17]. This phase is general and does not make any data structure-specific optimization as it provides guidelines independent from the actual implementation of the set. These optimizations are taken into account in the second phase, where we modify the previous OTB-Set design (i.e., the result of phase one) with the aim of further enhancing its performance. Here, we apply optimizations related to the implementation of the data structure rather than its semantic. Splitting the design in such a way allows the programmer to follow the same pattern for boosting more lazy data structures by first designing a general "non-optimized" transactional version using well-defined guidelines (phase one), and then adding optimizations to the specific data structure implementation, resulting in an "optimized", more performant, version (phase two).

We acknowledge that using concurrent data structures as "black boxes", as proposed by PTB, saves the effort for re-engineering them as transactional, however through OTB-Set we show that following our general guidelines it is not difficult to develop a transactional version of lazy set-based data structures, still retaining the advantage of enabling data structure specific optimizations.

We prove that OTB-Set (both the non-optimized and the optimized) provides individual linearizable operations and opaque transactions. To evaluate OTB-Set we compared its performance with both PTB [14] and lazy [13] sets. Our results show that OTB-Set's performance is closer to the highly concurrent lazy set

than PTB set in most cases. Beyond the performance improvement, OTB-Set has an added benefit: it is easy to integrate with lazy STM frameworks without violating their correctness or progress guarantees. This way, as we showed in [10], programmer can execute transactions with mixed access types, namely classical memory accesses (managed by the STM framework) and data structure accesses (managed by OTB-Set), without suffering from the disadvantages (i.e., false-conflict) of using an STM on top of a data structure as explained above.

The PTB and OTB approaches take an orthogonal direction to other works in literature for allowing semantic conflict detection. Techniques like open nesting [18], elastic transactions [8], transactional collection classes [4], and transactional predication [3] are different alternatives to design transactional data structures by using STM frameworks more efficiently than the naive STM-based data structures' implementation. The distinguishing point in both PTB and OTB is that they are completely independent and decoupled from STM frameworks[1], and they focus more on digging into the design and the implementation of the highly concurrent data structures and optimize the specific implementation of each one of them. Along the same line of OTB, techniques like COP [1,2] and ParT [21] exploit the same idea of splitting data structures' operations into an unmonitored traversal phase and a speculated validation/update phase. While COP operations are only concurrent (non-transactional), which do not natively compose and cannot be integrated with traditional memory frameworks, ParT discusses how to compose operations by employing a set of validators. In this paper, OTB proposes more reliance on the semantics of the data structure (especially in the "optimized" versions), and provides more details on how to compose dependent operations. Despite their differences, the above trials, along with OTB, confirm the trend of moving towards more optimistic approaches for semantic validation.

Our lazy set is publicly available as JAVA library at `http://www.hyflow.org/software.html`.

2 Optimistic Transactional Boosting

Optimistic transactional boosting (OTB) [11] is a methodology to boost lazy data structures to be transactional. A common feature that can be identified in all lazy data structures is that they have an *unmonitored traversal* step, in which the object's nodes are not kept locked until the operation ends. To guarantee consistency, this unmonitored traversal is followed by a validation step before the last step that physically modifies the shared data structure. As described in [11], OTB modifies the design of these lazy data structures to support transactions. Basically, the OTB methodology can be summarized in three main guidelines.

(G1) *Each data structure operation is divided into three steps.* *Traversal*. This step scans the objects, and computes the operation's results (i.e., its post-condition) and what it depends on (i.e., its precondition). This requires us

[1] In fact, OTB does not use STMs, rather it has been designed to be easily integrated with existing STM frameworks.

to define (in each transaction), what we call *semantic read-set* and *semantic write-set*, which store these information (*semantic write-sets* can also be called *semantic redo-logs*). *Validation*. This step checks the validity of the preconditions. Specifically, the entities stored in the semantic read-set are validated to ensure that operations are consistent. *Commit*. This step performs the modifications to the shared data structure. Unlike concurrent data structures, this step is not done at the end of each operation. Instead, it is deferred to the transaction's commit time. All information needed for performing this step are maintained in the semantic write-sets during the first step (i.e., traversal). To publish the write-sets, a classical (semantic) two-phase locking is used. This semantic (or abstract) locking prevents semantic conflicts at commit.

(G2) *Data structure design is adapted to support opacity.* The correctness of transactional data structures does not only depend on the *linearization* of its operations (like concurrent data structures), but it also depends on the sequence of the operations executed in each transaction. Data structure design has to be adapted to guarantee this *serialization* part. OTB provides the following guidelines, which exploit the local read-sets and write-sets to guarantee opacity [9] [2] , the same consistency level of most STM algorithms [5,6,19,15]:

- (G2.1) Each operation scans the local write-set first, before accessing the shared object. This is important to include the effect of the earlier (not yet published) operations in the same transaction.
- (G2.2) The read-set is re-validated after each operation and during commit, to guarantee that each transaction always observes a consistent state of the system (even if it will eventually abort).
- (G2.3) During commit, semantic locks of all operations are acquired before any physical modification on the shared data structure.
- (G2.4) Operations are applied during the commit phase in the same order as they appeared in the transaction and, in case the outcome of an operation influences the subsequent operations recorded in the write-set, they are updated accordingly.
- (G2.5) All operations have to be validated, even if the original (concurrent) operation does not make any validation (like `contains` operation in set). The goal of validation in these cases is to ensure that the same operation's result occurs at commit.

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

Unlike the first two guidelines, which are general for any lazy data structure, the third guideline varies from one data structure to another. It gives a hint to the developers that the data structures now are no longer used as black boxes, and

[2] In section 4, we prove that those guidelines are sufficient to guarantee opacity.

further optimizations can be applied. It is important to note that the generality of the first two guidelines does not mean that they can be applied "blindly" without being aware of the data structure's semantics. OTB, like the former techniques (including PTB) [1,2,21,14], performs better than the naive STM-based data structures only because it exploits semantics. However, we believe that OTB's guidelines make a clear separation between the general outline that can be applied on any lazy 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 specific optimizations that are completely dependent on the data structures implementation.

In Section 3, we show in detail how those guidelines can be used to design OTB-Set, an efficient transactional set based on both linked-list and skip-list. In Section 3.2, we follow the first two guidelines to design a non-optimized transactional version of the lazy set. Then, in Section 3.3, we show how specific optimizations can be applied on our OTB-Set (according to the third guideline).

3 OTB-Set

3.1 Preliminaries

Set is a collection of ordered items, which has three basic operations: `add`, `remove`, and `contains`, with the familiar meanings [16]. No duplicate items are allowed (thus, `add` returns false if the item is already present in the structure). All operations on different items 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 item are commutative as well. Such a high degree of commutativity between operations enables fine-grained semantic synchronization.

Lazy linked-list [13] is an efficient implementation of concurrent (non transactional) *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. It is important to note that the `contains` operation in the lazy linked-list is wait-free and is not blocked by any other operation.

Lazy skip-list is, in general, more efficient than linked-list as it takes logarithmic time to traverse the *set*. In skip-list, each node is linked to multiple lists (i.e., levels), starting from the list at the bottom level (which contains all the items), up to a random level. Therefore, `add` and `remove` operations lock an array of *pred* and *curr* node pairs (in a unified ascending order of levels to avoid deadlock), instead of locking one pair of nodes as in linked-list. For `add` operation, each node is enriched with a $fullyLinked$ flag to logically add it to the *set* after all levels have been successfully linked. Skip-list is also more suited than

linked-list in scenarios where the overhead of rolling back (compared to execution) is dominating. In fact, for a linked-list (and especially a long linked-list), even if aborts are rare, their effect includes re-traversing the whole list again, in a linear time, to retry the operation. In a skip-list, the cost of re-traversal is lower (typically in a logarithmic time), which minimizes the overhead of the aborts.

The implementation of the PTB version of the *set* is straightforward and does not change if the *set* implementation itself changes. In fact, it uses the underlying concurrent lazy linked-list (or skip-list) to execute the *set* operations. If the transaction aborts, a successful `add` operation is rolled back by calling the `remove` operation on the same item, and vice versa (more details are in [14]).

Despite the significant improvement in the traversal cost and abort overhead, the implementation of OTB skip-list and OTB linked-list are very similar. Due to space constraints, and with the purpose of making the presentation clear, we focus on the linked-list implementation, and we highlight the main differences with respect to the skip-list implementation when necessary (the full implementation details of both linked-list-based and skip-list-based OTB-Set can be found in the source code).

3.2 Non-Optimized OTB-Set

Following the first two guidelines (*G1* and *G2*) mentioned in Section 2, in this section we show how to boost the lazy *set* to design a transactional *set* without any specific optimization related to the details of its implementation. According to *G1*, we divide OTB-Set operations into three steps. The *Traversal* step is used to reach the involved nodes, without any addition to the semantic read-set. The *Validation* step is used to guarantee the consistency of the transaction and the linearization of the list. We define two different validation procedures: one is named *post-validation*, which is called after each operation, and the other is named *commit-time-validation*, which is called at commit time and after acquiring the semantic locks. The *Commit* step, which modifies the shared list, is deferred to transaction's commit. Following *G2*, we show how the usage of lazy updates, semantic locking, and post-validation guarantees opacity.

Similar to the lazy linked-list, each operation in OTB-Set involves two nodes at commit time: *pred*, which is the largest item less than the searched item, and *curr*, which is the searched item itself or the smallest item larger than the searched item[3]. To log the information about these nodes, with the purpose of using them at commit time, we adopt the same concept of read-set and write-set as used in lazy STM algorithm (e.g., [5,6]), but at the semantic level. In particular, each read-set or write-set entry contains the two involved nodes in the operation and the type of the operation. In addition, the write-set entry contains also the new value to be added in case of a successful `add` operation.

[3] Sentinel nodes are added as the head and tail of the list to handle special cases.

The only difference in skip-list is that the read-set and write-set entries contain an array of *pred* and *curr* pairs, instead of a single pair. This is because the searched object can be in more than one level of the skip-list.

Algorithm 1 OTB Linked-list: `add`, `remove`, and `contains` operations.

```
 1: procedure OPERATION(x)
                          ▷ 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 curr = head.next
 8:     while curr.item < x do
 9:         pred = curr
10:         curr = curr.next
                          ▷ Step 3: Save reads and writes
11:     rse = new ReadSetEntry(pred,curr,op)
12:     read-set.add(rse)
13:     if op is add or remove then
14:         wse = new WriteSetEntry(pred,curr,op,x)
15:         write-set.add(wse)
                                ▷ Step 4: Post Validation
16:     if ¬ post-validate(read-set) then
17:         ABORT
18:     else if Successful operation then
19:         return true
20:     else
21:         return false
22: end procedure
```

Algorithm 1 shows the pseudo code of the linked-list operations. We can isolate the following four parts of each operation.

Local writes check (lines 2-6). Since writes are buffered and deferred to the commit phase, this step guarantees consistency of further reads and writes. Each operation on an item x checks the last operation in the write-set on the same item x and returns the corresponding result. For example, if a transaction previously executed a successful `add` operation of item x, then further additions of x performed by the same transaction must be unsuccessful and return false. In addition, if the new operation is a writing (i.e., `add/remove`) operation, it should be appended to the corresponding write-set entry (line 5). If there is no previous (local) operation on x in the write-set, then the operation starts traversing the shared linked-list as shown in the next step.

Traversal (lines 7-10). This step is the same as in the lazy linked-list. It saves the overhead of all unnecessary monitoring during traversal that, otherwise, would be incurred with a native STM algorithm for managing concurrency.

Logging the reads and writes (lines 12-15). At this point, the transaction records the accessed nodes, that are semantically relevant to the *set*, into its local read-set and write-set. All operations must add the appropriate read-set entry, while `add/remove` operations modify also the write-set (line 15). It is worth to note that having no entries in the write-set for `contains` operation means that it does not need to acquire locks during the commit phase. This way, although the `contains` operation is no longer wait-free, like its concurrent lazy version (because it may fail during the commit-time-validation), it still performs efficiently due to the absence of the semantic locks acquisition. We recall that, rather than OTB, PTB has to acquire semantic locks even for the `contains` operation to maintain consistency and opacity.

Post-Validation (lines 16-21). At the end of the traversal step, the involved nodes

are stored in local variables (i.e., *pred* and *curr*). At this point, according to point *G2.2* and to preserve opacity [9], the read-set is post-validated to ensure that the transaction does not observe an inconsistent snapshot. The same post-validation mechanism is used at memory-level by STM algorithms such as NOrec [5]. More details about post-validation are discussed later in Algorithm 2.

As mentioned before, there is a difference between linked-list and skip-list regarding the `add` operation. In fact, in the skip-list the new node has to be linked to multiple levels, thus there could be a time window where the new node is only linked to some (and not all) levels. To handle this case in our OTB-Set, any concurrent operation waits until the *fullyLinked* flag becomes true, and then it proceeds.

Algorithm 2 shows the post-validation step. The validation of each read-set entry is similar to the one in lazy linked-list: both *pred* and *curr* should not be deleted, and *pred* should still link 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 transactions after invoking the `contains` operation and before committing the transaction may invalidate the returned value of the operation, making the transaction's execution semantically incorrect.

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

Algorithm 2 OTB Linked-list: validation.

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 $\neq$ *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**

Algorithm 3 shows the commit step of OTB-Set. Read-only transactions have nothing to do during commit (line 2), because of the incremental validation during the execution of the transaction. For write transactions, according to point *G2.3*, the appropriate locks are first acquired using CAS operations (lines 4-6). Like the original lazy linked-list, any `add` operation only needs to lock *pred*, while `remove` operations lock both *pred* and *curr*. As described in [13], this is enough for preserving the correctness of the write operations. To avoid

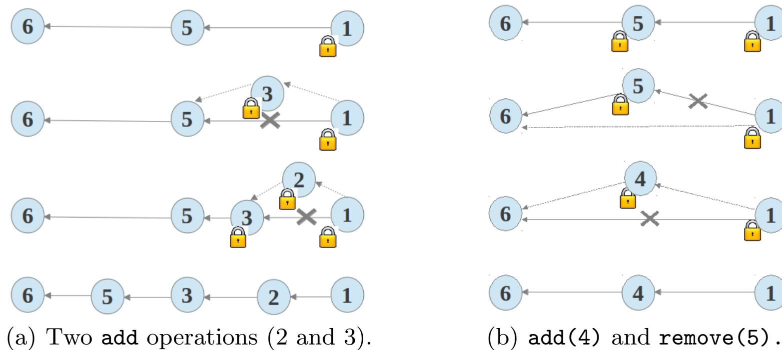
deadlock, any failure during the lock acquisition implies aborting and retrying the transaction (releasing all previously acquired locks).

After the semantic lock acquisition, the validation is called, in the same way as in Algorithm 2, to ensure that the read-set is still consistent (line 7). If the commit-time-validation fails, then the transaction aborts.

Algorithm 3 OTB Linked-list: commit.

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

The next step of the commit procedure is to publish writes on the shared linked-list, and then release the acquired locks. This step is not straightforward because each node may be involved in more than one operation of the same transaction. In this case, the saved *pred* and *curr* of these operations may change according to which operation commits first.



(a) Two `add` operations (2 and 3). (b) `add(4)` and `remove(5)`.

Fig. 1. Executing more operations that involve the same node in the same transaction.

For example, in Figure 1(a), both 2 and 3 are inserted between the nodes 1 and 5 in the same transaction. During commit, if node 2 is inserted before node 3, it should be the new predecessor of node 3, but the write-set still records node 1 as the predecessor of node 3. In OTB guidelines, *G2.4* solves this issue. When node 2 is inserted, the operation scans the write-set again to find any other operation that has node 1 as its *pred* and replaces it with node 2. The same technique is used in the case of removal (Figure 1(b)). When node 5 is removed, any write-set entry that has node 5 as its *curr* replaces it with node 6, and any write-set entry that has node 5 as its *pred* replaces it with node 1. Lines 20-22 and 26-30 illustrate these cases.

It is clear that the inserted nodes have to be locked until the whole commit procedure is finished. Then they are unlocked along with the other *pred* and *curr* nodes (line 17). For example, in Figure 1(a), all nodes (1, 2, 3, 5) are locked and no transaction can access them until the commit terminates.

3.3 Optimized OTB-Set

One of the main advantages of OTB over the original PTB is that it uses the underlying (lazy) data structure as a white-box, which allows more data structures-specific optimizations.

In general, decoupling the boosting layer from the underlying concurrent data structure is a trade-off. Although, on the one side, considering the underlying data structure as a black-box means that there is no need to re-engineer its implementation, on the other side, it does not allow to customize its implementation and thus to exploit the new transactional specification, especially when the re-engineering effort can be easily achieved. For this reason, as showed in the previous section, 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 *G3*). We believe this division makes the re-engineering task easier and, at the same time, it allows specific optimizations for further enhancing the performance.

In this section, we show optimizations for our OTB-Set, leveraging the fact that it treats the underlying lazy linked-list as a white-box and, therefore, it can be adapted as needed. Due to space constraints, we defer the details on how to modify the aforementioned "non-optimized" algorithms to Appendix B.

Unsuccessful `add` and `remove`. The `add` and `remove` operations are not necessarily considered as writing operations, because duplicated items are not allowed in the *set*. For example, if an `add` operation returns false, it means that the item to insert already exists in the *set*. The commit of such operation can be done by only checking that the item still exists in the *set*, which allows to treat unsuccessful `add` operations as successful `contains` operations. This way, the transaction does not acquire any lock for this operation at commit. The same idea can be applied on the unsuccessful `remove` operation which can be treated as an unsuccessful `contains` operation during commit.

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

In the lazy linked-list, the `add` and `remove` operations acquire locks on the *pred* and *curr* nodes even if the operations are unsuccessful. PTB inherits this unnecessary lock acquisition because it uses the lazy linked-list as a black-box.

Eliminating Operations. As shown in Algorithm 1, each operation starts with checking the local writes before traversing the shared list. During this step, for improving OTB performance, if a transaction adds an item x and then removes the same item x, or vice versa, we allow those operations to locally eliminate each other. This elimination is done by removing both entries from the write-set, which means that the two operations will not make any physical modification on the shared list. No entry in the read-set is locally eliminated because, this way, the commit time-validation can still be performed on those operations in order to preserve transaction's correctness.

In PTB, due to the usage of the underlying lazy linked-list as a black-box, this scenario is handled by physically adding x to the shared *set*, and then physically removing it, introducing an unnecessary overhead.

Simpler Validation. In the case of successful `contains` and unsuccessful `add` operations, we use a simpler validation than the original validation of the lazy linked-list. In these particular cases, the transaction only needs to check that *curr* is still not deleted, since that is sufficient to guarantee that the returned value is still valid (recall that if the node is deleted, it must first be logically marked as deleted, which will be detected during validation). This optimization prevents false invalidations, where conflicts on *pred* are not real semantic conflicts.

The validation in the skip-list is similarly optimized because we leverage the rule that all items have to appear in the lowest level of the skip-list. For successful *contains* and unsuccessful *add* operations, it is sufficient to validate that *curr* is not deleted, which ensures that the item is still in the *set*. We can also optimize unsuccessful *remove* and *contains* by only validating the *pred* and *curr* in the lowest level to make sure that the item is still not in the *set*, because if the item is inserted by another transaction, it must affect this level. For successful *add* and *remove* operations, all levels need to be validated to prevent conflicts.

Optimized Commit. To ensure that the operations in Figure 1 are executed correctly, the write-set has to be re-scanned for each write operation (according to the OTB guideline *G2.4*), as we showed in Section 3.2. This overhead becomes significant if the write-set is relatively large. We optimize this routine and avoid the need of re-scanning the write-set by the following points. *(1)* The items

are added/removed in descending order of their values, regardless of their order in the transaction execution. This guarantees that the *pred* of each write-set entry is always valid, non-deleted, and not touched by any previous operation in the transaction. *(2)* Operations resume traversal from the saved *pred* to the new *pred* and *curr* nodes. At this stage, the *pred* and *curr* nodes can only be changed because of some previous local operations. This is because the transaction already finished the lock acquisition and validation, which prevents any conflicting transaction from proceeding.

Using these two points, the issue in Figure 1(a) is solved without re-scanning the write-set. The first point enforces that node 3 is inserted first. Subsequently, according to the second point, when 2 is inserted, the transaction will resume its traversal from node 1 (which is guaranteed to be locked and non-deleted). Then, it will detect that node 3 is its new *succ*, and will correctly link node 2.

The removal case is shown in Figure 1(b), in which node 5 is removed and node 4 is inserted. Again, 5 must be removed as first (even if 4 is added earlier during the transaction execution), so that when 4 is added, it will correctly link to 6 and not to 5. Two subsequent `remove` operations follow the same procedure.

Skip-list uses the same procedure but at all levels. This is because each level is independent from the others, which means that the *preds* of the same node in two or more levels may be different. For this reason, the same procedure described above is repeated at each level, independently.

4 Correctness

In this section, we discuss the arguments that we use for assessing the correctness of OTB-Set, and, due to space constraints, we report the detailed correctness proof in Appendix A.

The correctness of OTB-Set can be proved in two incremental steps. The first step is to show that, after the modifications needed for supporting the execution of transactions, each single operation on the *set* is still linearizable, like the lazy *set*. The second step consists of showing that the whole transaction is opaque [9].

A) Linearizability: Each operation traverses the *set* following the same rules as in the lazy *set*. After the traversal, we can distinguish between write and read operations' behavior. A write operation, instead of acquiring the locks on the involved nodes instantaneously after the traversal, it acquires the same locks, but at transaction commit time. Since the transaction is validated after the locks acquisition using the same validation done by the lazy *set*, the linearization points of each write operation is just shifted to the commit phase of the transaction (rather than after the operation as in the lazy *set*). We cannot use the same arguments for defining the linearization point of the read operations in our OTB-Set. In fact, in lazy *set*, a `contains` operation is wait-free, which implies that its linearization point is when the *curr* node is checked[4]. In OTB-Set,

[4] In some exceptional cases, discussed in [13], the linearization point of the unsuccessful *contains* operation becomes earlier. However, those special cases are not relevant when we discuss the correctness of our OTB-Set.

where `contains` operations are no longer wait-free, this point is replaced with the point when each operation is re-validated during the transaction commit.

B) Opacity: Considering the transaction as a whole, the combination of *lazy writes*, *post-validation*, and *commit-time-validation* is sufficient for guaranteeing opacity. In fact, this is the same approach used at memory level in many lazy STM algorithms such as NOrec [5] to enforce opacity. Specifically, all operations are linearized at the transaction's commit time and after acquiring all the semantic locks. This allows the committed transactions to appear as happened at a single indivisible point in time. Aborted transactions do not expose any write to other transactions, because, in general, transactions never write in the shared *set* unless they are sure that they will not eventually abort. Live transactions (whether they will eventually commit or abort) never observe an inconsistent state because they validate their entire read-set after each read (in the *post-validation* routine) and during the transaction commit (in the *commit-time validation* routine). Finally, the effect of interfering operations of the same transaction is preserved leveraging the points *G2.1* and *G2.4* of OTB guidelines.

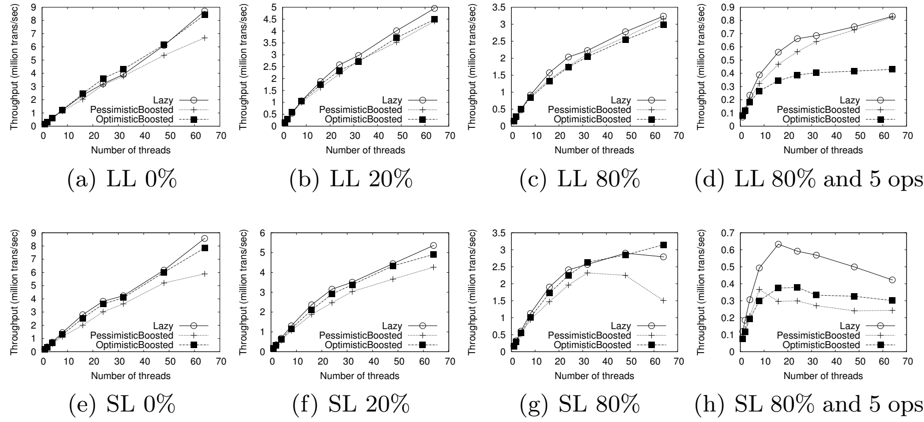
The optimizations described in Section 3.3 do not break opacity simply because they do not contradict with any of the previously mentioned evidences. It is also straightforward to prove that composing the operations on two different OTB-Set instances does not break the opacity of the transaction as a whole. This is because each read/write-set entry will be validated and/or published independently.

5 Experimental Evaluation

In this section we evaluate the performance of our OTB-Set's Java implementation equipped with the optimizations described in Section 3.3. We compared it with lazy *set* [13] and PTB *set* [14]. In order to conduct a fair comparison, the percentage of the writes in all of the experiments is the percentage of the successful ones, because 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 item added by previous transactions as a parameter, such that it will probably succeed. In each experiment, the number of `add` and `remove` operations are kept equal to avoid significant fluctuations of the data structure size during the experiments.

The experiments were conducted on a 64-core machine, which has four AMD Opteron (TM) Processors, each with 16 cores running at 1400 MHz, 32 GB of memory, and 16KB L1 data cache. 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.

We use transactional throughput as our key performance indicator. Although abort rate is another important parameter to measure and analyze, it is meaningless in our case. Both lazy *set* and PTB *set* do not explicitly abort the transaction. However, there is an internal retry for each operation if validation fails.



(a) LL 0% (b) LL 20% (c) LL 80% (d) LL 80% and 5 ops

(e) SL 0% (f) SL 20% (g) SL 80% (h) SL 80% and 5 ops

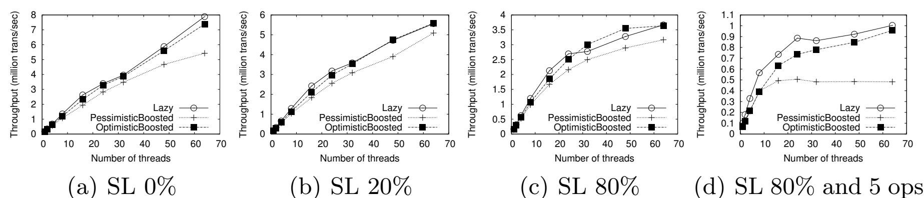
Fig. 2. Throughput of linked-list-based (LL) and skip-list-based (SL) set with 512 elements (labels indicate % write transactions). Four different workloads: read-only (0% writes), read-intensive (20% writes), write-intensive (80% writes), and high contention (80% writes and 5 operations per transaction).

Additionally, PTB aborts only if it fails to acquire the semantic locks, which is less frequent than validation failures in the OTB-Set. We recall that the lazy *set* is not capable to run transactions at all (i.e., it is a concurrent data structure, not transactional). We only show it as a rough upper bound for the OTB-Set and PTB, but it actually does not support transactional operations.

We first show the results for a linked-list implementation of the *set*. In this experiments, we used a linked-list with 512 nodes. In order to conduct a comprehensive evaluation of OTB-Set's performance, in the first row of Figure 2 we show the results for four different linked-list workloads: read-only (0% writes and 1 operation per transaction), read-intensive (20% writes and 1 operation per transaction), write-intensive (80% writes and 1 operation per transaction), and high contention (80% writes and 5 operations per transaction). In both read-only and read-intensive workloads, OTB-Set performs closer to the (upper bound) performance of the lazy list than PTB-Set. This is expected, because PTB incurs locking overhead even for read operations. In contrast, OTB-Set, like lazy linked-list, does not acquire locks on read operations, although it still has a small overhead for validating the read-set. For the write-intensive workload, PTB starts to be slightly better than OTB-Set, and the gap increases in high contention workloads. This is also expected, because contention becomes very high, which increases abort rate (recall that aborts have high overhead due to re-traversing the list in linear time). In these high/very high contention scenarios, the "pessimism" of PTB pays off more than the "optimism" of OTB-Set. For example, in the high contention scenario, five operations are executed per transaction. In PTB, each operation (pessimistically) locks its semantic items before executing each operation and then it keeps trying to execute the opera-

tion on the underlying (black-box) concurrent data structure. On the other hand, OTB suffers from aborting the whole transaction even if the last operation of the transaction fails.

In the second row of Figure 2, the same results are shown for the skip-list-based *set* of the same size (512 nodes). The results show that OTB-Set performs better in all cases, including the high contention case. This confirms that OTB-Set gains because of the reduced overhead of aborts. Although the semantic contention is almost the same (for a *set* with 512 nodes, contention is relatively high), using a skip-list instead of a linked-list supports OTB-Set more than PTB. This is mainly because skip-list traverses less nodes of the set through the higher levels of the skip-list. Thus, even if the whole transaction aborts, re-executing skip-list operations is less costly than linked-list.



(a) SL 0% (b) SL 20% (c) SL 80% (d) SL 80% and 5 ops

Fig. 3. Throughput of skip-list-based set with 64K elements (labels indicate % write transactions). Four different workloads: read-only (0% writes), read-intensive (20% writes), write-intensive (80% writes), and high-contention (80% writes and 5 operations per transaction).

The last set of experiments (Figure 3), shows the performance when the contention is significantly lower. We used a skip-list of size 64K and measured throughput for the same four workloads. The results show that in such cases, which however are still practical, OTB-Set is up to 2× better, even in write-intensive and high contention workloads. This is mainly because in the very low contention scenario, the PTB's eager locking mechanism becomes ineffective and a more optimistic algorithm, such as OTB-Set, is preferable.

6 Conclusions

In this paper we provided a detailed design and implementation of a transactional optimistic *set* data structure (OTB-Set). We presented two versions of OTB-Set: one "non-optimized", derived from the implementation of general guidelines; and one "optimized", which aims at further enhancing the performance. We also proved the correctness of the designed set and showed that OTB-Set operations guarantee opacity. Our evaluation revealed that the performance of OTB-Set is closer to highly concurrent (non-transactional) lazy *set* than the original transactional boosting version in most of the cases.

Acknowledgments

This work is supported in part by US National Science Foundation under grant CNS-1116190.

References

1. Y. Afek, H. Avni, and N. Shavit. Towards consistency oblivious programming. In *OPODIS*, pages 65–79, 2011.
2. H. Avni and B. C. Kuszmaul. Improving htm scaling with consistency-oblivious programming. In *TRANSACT*, 2014.
3. N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. Transactional predication: high-performance concurrent sets and maps for stm. In *PODC*, pages 6–15, 2010.
4. B. D. Carlstrom, A. McDonald, M. Carbin, C. Kozyrakis, and K. Olukotun. Transactional collection classes. In *PPOPP*, pages 56–67, 2007.
5. L. Dalessandro, M. F. Spear, and M. L. Scott. NOrec: streamlining STM by abolishing ownership records. In *PPOPP*, pages 67–78, 2010.
6. D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Distributed Computing*, pages 194–208. Springer, 2006.
7. N. L. Diegues and P. Romano. Time-warp: lightweight abort minimization in transactional memory. In *PPoPP*, pages 167–178, 2014.
8. P. Felber, V. Gramoli, and R. Guerraoui. Elastic transactions. In *DISC*, pages 93–107. 2009.
9. R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *PPOPP*, pages 175–184, 2008.
10. A. Hassan, R. Palmieri, and B. Ravindran. Integrating transactionally boosted data structures with stm frameworks: A case study on set. In *TRANSACT*, 2014.
11. A. Hassan, R. Palmieri, and B. Ravindran. Optimistic transactional boosting. In *PPOPP*, pages 387–388, 2014.
12. A. Hassan, R. Palmieri, and B. Ravindran. Remote invalidation: Optimizing the critical path of memory transactions. In *IPDPS*, pages 187–197, 2014.
13. S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. Scherer, and N. Shavit. A lazy concurrent list-based set algorithm. In *OPODIS*, pages 3–16. Springer, 2005.
14. M. Herlihy and E. Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *PPOPP*, pages 207–216, 2008.
15. M. Herlihy, V. Luchangco, M. Moir, and W. N. S. III. Software transactional memory for dynamic-sized data structures. In *PODC*, pages 92–101, 2003.
16. M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Elsevier, 2012.
17. M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.
18. Y. Ni, V. 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 *PPOPP*, pages 68–78, 2007.
19. T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In *DISC*, pages 284–298, 2006.
20. N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.
21. L. Xiang and M. L. Scott. Composable partitioned transactions. In *WTTM*, 2013.

A Detailed Correctness of OTB-Set

In this appendix, we give more details about the correctness of OTB-Set. In Section A.1 we assume, without loss of generality, transactions composed of only one OTB-Set operation and we prove that this operation is linearizable. Then, in Section A.2, we prove that if the transaction contains more than one operation of a *non-optimized* OTB-Set, the implementation of OTB guidelines guarantees opacity [9]. Finally, Section A.3 discusses the correctness of the *optimized* OTB-Set.

A.1 Linearization

Although OTB-Set does not use the lazy *set* as a black box, it uses similar mechanisms for traversing the *set* and validating the nodes accessed. For this reason, we can assess the correctness of our OTB-Set implementations by relying on the correctness of the lazy *set*. Again, we focus on a linked-list-based *set*, assuming that applying the same concepts to a skip-list-based *set* is straightforward.

- *Successful `add`*. As described in [13], a successful `add` operation in a lazy linked-list is linearized in the moment when *pred*.next is set. This is because, at this point, the `add` operation already *i)* acquired the lock on *pred*, *ii)* validated that *pred* and *curr* are not deleted and that *pred* still links to *curr*. This means that the operation is still valid and no other concurrent operation can interfere with it. In OTB-Set, the same linearization point is selected. We now assume that transactions consist of only one operation. This means that, until the moment of acquiring the lock on *pred*, both the lazy *set* and OTB-Set behave in a similar way, even though the lock acquisition itself is delayed to the commit phase in OTB-Set. Once the lock is acquired in OTB-Set (at commit), it uses the same validation as the lazy *set* which allows having the same linearization point.
- *Successful `remove`*. In the lazy linked-list, the successful `remove` operation is linearized when the entry is marked as *deleted*. We can safely select the same linearization point in OTB-Set because, although this point is shifted to the transaction's commit, OTB-Set still behaves similar to the lazy *set*, like the `add` operation.
- *Successful `contains`*. Here, the linearization points of lazy *set* and OTB-Set are different. In lazy *set*, the `contains` operation is wait-free and the successful `contains` operation is linearized immediately when the *deleted* flag of a matching entry is observed to be false. In OTB-Set, however, this point cannot be selected as the linearization point for two reasons. First, the linearization point should be selected somewhere in the commit phase, thus allowing the further extension of having more than one operation in the same transaction (that we will discuss in Section A.2). Second, according to the point *G2.4* of the OTB guidelines, all operations, including the `contains` operation, should re-validate their results at commit, which means also that the operation may fail during the commit and therefore the transaction may abort. For this reason, we select

the linearization point of the `contains` operations when those operations are successfully re-validated at commit time.

- *Unsuccessful* `contains`. Like successful `contains`, the unsuccessful `contains` operations are re-validated at commit time and we select this re-validation point as our linearization point. It is worth to note that the definition of the linearization point of the unsuccessful `contains` operations in the lazy *set* is not straightforward. In fact, in the lazy linked-list, the linearization point of a logically (but not physically) deleted node has to be before that any other transaction occasionally added a new node with the same item. However, this case in our OTB-Set is not relevant because we abort the transaction in this case (remember that we validate that both the *pred* and *curr* nodes are not logically deleted). Although this abort can be seen as a false conflict, we allow it for the sake of making a simple validation process, and assuming that such case is rare. The lazy *set* also suffers from a similar false conflict in the case of unsuccessful `remove`, and they also chose not to optimize this case for the sake of a simple and clear validation process.
- *Unsuccessful* `add/remove`. In the non-optimized version of the OTB-Set, described in Section 3.2, the linearization points of the unsuccessful `add/remove` operations are the same as in the lazy *set*. This is because, in this non-optimized version, the same locks (on the *pred* and *curr* nodes) are acquired and the same validation is used. In the optimized version of the OTB-Set, as described in Section 3.3, those linearization points change because we now consider the operations as read operations and we do not acquire any locks for them during commit. As we mentioned in Section 3.3, those operations are considered during commit as successful/unsuccessful `contains` operations, which motivates using the same linearization points as the `contains` operation in this case, rather than the linearization points mentioned in [13].

A.2 Opacity - Non-Optimized OTB-Set

Opacity was proposed in [9] to formally proof the correctness of TM implementations, and most STM algorithms are proven to guarantee opacity [5,6,19,15]. Intuitively, as mentioned in [9], opacity is guaranteed if three requirements are captured: *i)* every committed transaction atomically appears in a single indivisible point in the history of the committed transactions, *ii)* live transactions do not see the intermediate results of any aborted transaction, *iii)* transactions are always consistent even if they will eventually abort. In this section and in the following section, we show that those three requirements are preserved in both the "non-optimized" and the "optimized" OTB-Set, respectively. The correctness of the non-optimized OTB-Set can be also used for any other lazy data structure as far as it follows the same two guidelines mentioned in Section 2 (i.e., *G1* and *G2*).

In the following, we borrow the same terminology used in [9]. However, for brevity, instead of having two points in the history for each operation (the *invocation* point and the *return* point), we will only show one point which reflects the *return* point. This is acceptable because any transaction is serial, meaning

that it does not invoke a new operation until the previous operation returns its value, and the *invocation* point is not relevant to the execution of any other transaction.

- *Equivalence to a legal sequential history.* The first requirement for a history H to be opaque is that if we remove all non-committed transaction, the resulting sub-history H' is equivalent to a legal sequential history S that preserves the real-time order of the transactions in H'. In OTB-Set, H' preserves the real-time order because all operations are linearized during the commit phases of their transactions. For that reason, a committing transaction can be serialized in one point, right after the transaction successfully acquires its semantic locks. After this serialization point, if the transaction successfully validates its read-set, all conflicting transactions in H' will be serialized after it. If it fails in validation, it will simply abort.
  Precisely, we have five cases to cover for proving the legality of any sub-history H' of some committed OTB-Set transactions $T_1, T_2, ..., T_n$:
  1. Transaction are executed serially: which means that each transaction starts after the previous transaction commits. The real-time order in this case is natively preserved because after T_i commits, all its writes are immediately visible to the following transactions (threads are not caching any state of the objects).
  2. Concurrent transactions are independent (which means that they have no intersection in their read/write-sets or the intersection is only between read-sets). Natively, they can be serialized in any order. The history of each transaction as a standalone transaction is kept legal using the guidelines *G2.1* and *G2.4*. For example, in the following history:
     $H1 =< add(T_i, x, true), contains(T_i, x, true), remove(T_i, x, true), tryC_{T_i}, C_{T_i} >$
     G2.1 guarantees that both the `contains` and the `remove` operations cannot return an illegal value (which is false in this case) during the execution of the transaction, and *G2.4* guarantees that the `remove` operation will be executed correctly at commit (remember that we are now proving the non-optimized version of OTB-Set which means that operations are executed according to their invocation order and without any local elimination during commit).
  3. The write-sets of two concurrent transactions, T_i and T_j, intersect. Clearly the commit phases of those transactions can never execute concurrently. Either one of them will fail in acquiring the semantic locks and thus will abort, or T_j will start its commit after T_i entirely finishes its commit and releases its locks, which allows serializing T_i before T_j.
  4. The read-set of T_i intersects with the write-set of T_j and the read-set of T_j does not intersect with the write-set of T_i. In this case, T_i will either abort during the commit-time validation, or it will successfully finish its validation before T_j acquires the "conflicting" semantic locks. In the latter case, T_i can be safely serialized before T_j.

5. The read-set of T_i intersects with the write-set of T_j and the read-set of T_j intersects with the write-set of T_i. In this case, any scenario where both transactions concurrently commit is illegal. For example, in the following two histories[5]:
$H2 =< remove(T_k, x, true), remove(T_k, y, true), tryC_{T_k}, C_{T_k}, add(T_i, x, true), contains(T_j, x, false), add(T_j, y, true), contains(T_i, y, false), tryC_{T_i}, C_{T_i}, tryC_{T_j}, C_{T_j} >$
$H3 =< remove(T_k, x, true), remove(T_k, y, true), tryC_{T_k}, C_{T_k}, add(T_i, x, true), contains(T_j, x, true), add(T_j, y, true), contains(T_i, y, true), tryC_{T_i}, C_{T_i}, tryC_{T_j}, C_{T_j} >$
Both histories are illegal because the `contains` operations in T_i and T_j cannot return both false or both true[6]. A possible legal case is that the `contains` operation of T_i returns false and the one of T_j returns true (which allows T_i to be legally serialized before T_j).
Our validation process in Algorithm 2 prevents that all these illegal scenarios can happen. As we validate that the nodes in the read-set are both *unlocked* and *valid*. T_i and T_j cannot both successfully acquire the semantic locks and then successfully validate their read-sets before starting to write. At least one transaction will abort because some entries in its read-set is locked by the other transaction.

- *The effect of the aborted transactions*. Aborted transactions in OTB-Set have no effect on the live transactions. This is simply because transactions do not publish any writes until their commit phase. During commit, if a transaction successfully acquires the semantic locks and then it successfully validates its read-set, it cannot abort anymore. Accordingly, it is safe at this point to start writing on the shared *set*.
- *Consistency of live transactions*. Transactions which guarantee opacity should always observe a consistent state. This also includes the *live* transactions, which are the transactions that did not yet commit or abort. Theoretically, as mentioned in [9], we can transform any history which contains some live transactions to a complete history by either committing or aborting those live transactions. The challenge here is to prove that this completed history is still legal (which means that the operations executed so far inside the live transactions are legal). In OTB-Set we guarantee that live transactions always observe a consistent state by the *post-validation* procedure which validates, after each operation, that the entire read-set is still valid. Precisely, in a history H, an operation $< op(T_i, x, true/false) >$, can be implicitly extended to either $< op(T_i, x, true/false), validate(T_i, succeeded) >$ or $< op(T_i, x, true/false), validate(T_i, failed), A_{T_i} >$ according to whether its validation succeeds or fails, *which guarantees preserving the legality of* H.

[5] We put the first two operations of T_k to enforce that x and y are both in the *set* before T_i and T_j start.

[6] This case is an example of producing a cyclic opacity graph which is mentioned in [9].

A.3 Opacity - Optimized OTB-Set

In this section we show how each optimization discussed in Section 3.3 does not prevent transactions to guarantee opacity.

- *Unsuccessful `add/remove`*. OTB-Set validates the unsuccessful `add/remove` operations as `contains` operations. It can be easily shown that this does not break transactions' consistency. Although operations are semantically different, handling them in the same way at commit (at the memory level) does not break the semantics with any means, as far as the same result is validated during commit.
- *Eliminating Operations*. Elimination does not break consistency because operations are eliminated only from the write-sets. If the operations were also eliminated from the read-sets, opacity may be broken because another transaction may modify the *set* and then commit successfully before the commit of the former transaction, which violates the serialization points we mentioned in the previous section. For example, in the following history: $H4 =< add(T_i, x, true), remove(T_i, x, true), add(T_j, x, true), tryC(T_j), C(T_j), tryC(T_i), C(T_i) >$ transaction T_i becomes illegal. This is because at the serialization point of T_i, when it commits, the `add` operation cannot return true because T_j already added x to the *set* and committed. In our OTB-Set implementation, T_i will detect during its commit that T_j added x because the eliminated operations are still in the read-set and they will be validated during commit, and thus triggering the abort of T_i.
- *Simpler Validation*. For the successful `contains` and the unsuccessful `add` operations, the *curr* node is detected to match the searched item x, and to be not *deleted*. During the commit phase of a transaction T_i, it is sufficient to check the *deleted* mark of the *curr* node. This is because any other transaction cannot execute any new writing operation on x before deleting the previous node, and deleting x is done first by logically mark the node as *deleted*. This means that, if T_i observes at commit that x is not logically deleted, then the operation is still valid. In this case, there is no need to validate the *pred* node.
- *Optimized Commit*. The correctness of this optimization is based on two facts. First, as write operations on the same item are eliminating each other, we cannot observe two entries of a transaction T_i's write-set, which add (or remove) the same item x. This means that all operations in the write-set are commutative (i.e. not semantically conflicting), and can be (semantically) executed in any order. Second, at memory level, it becomes also unnecessary to execute those write operations in the same order as their original order, because each operation does not change the *pred* of any subsequent operation (as they are sorted in a descending order). As the *pred* node is not changed, it is safe for any operation to start from that *pred* node to reach the new *pred* and *curr* nodes.

B Detailed Optimized OTB-Set Algorithms

In Section 3.3, we showed four optimizations on the basic implementation of OTB-Set. In this appendix, we show more details about how to modify Algorithms 1, 2, and 3 to maintain these optimizations.

Algorithm 4 OTB Linked-list: Optimized `add`, `remove`, and `contains` operations.

```
 1: procedure Operation(x)
                                        ▷ Step 1: search local write-sets
 2:     if x ∈ write-set and write-set entry is add then
 3:         if operation = add then
 4:             return false
 5:         else if operation = contains then
 6:             return true
 7:         else                                              ▷ remove
 8:             delete write-set entry
 9:             return true
10:     else if x ∈ write-set and write-set entry is remove then
11:         if operation = remove or operation = contains then
12:             return false
13:         else                                              ▷ add
14:             delete write-set entry
15:             return true
16:     ...
                                        ▷ Step 3: Save reads and writes
17:     read-set.add(new ReadSetEntry(pred, curr, operation))
18:     if Successful add/remove then
19:         write-set.add(new WriteSetEntry(pred, curr, operation, x))
20:     if Successful operation then
21:         return true
22:     else
23:         return false
24: end procedure
```

Algorithm 4 shows the modification to steps 1 and 3 of Algorithm 1 to achieve the first two optimizations. In step 1, eliminated operations are removed from the write-set, while still keeping them in the read-set (lines 8 and 14). In step 3, only successful `add` and `remove` operations are added to the write-set (line 19)

Algorithm 5 OTB Linked-list: optimized validation.

```
 1: procedure Validate(read-set)
 2:     ...
 3:     for all entries in read-sets do
 4:         if successful contains or unsuccessful add then
 5:             if curr.deleted then
 6:                 return false
 7:         else
 8:             if pred.deleted or curr.deleted or pred.next ≠ curr then
 9:                 return false
10:         return true
11:     ...
12: end procedure
```

Algorithm 5 shows the optimized validation procedure. Lines 3-10 replace lines 6-8 in Algorithm 2.

Algorithm 6 OTB Linked-list: optimized commit.

```
1: procedure COMMIT
2:     ...
3:     sort write-set descending on items
4:     for all entries in write-sets do
5:         curr = pred.next
6:         while curr.item < x do
7:             pred = curr
8:             curr = curr.next
9:     ...
10: end procedure
```

Algorithm 6 describes the modified commit procedure, which replace lines 20-22 and 26-30 of Algorithm 3). Line 3 applies the first guideline point in Section 3.3. Lines 5-8 apply the second guideline point.