# Integrating Transactionally Boosted Data Structures with STM Frameworks: A Case Study on Set

Ahmed Hassan Roberto Palmieri Binoy Ravindran
Virginia Tech
hassan84@vt.edu robertop@vt.edu binoy@vt.edu

## Abstract

Providing transactional collections of data structures with the same performance of highly concurrent data structures enables performance-competitive transactional composability. Although Software Transactional Memory (STM) is increasingly becoming a promising technology for designing and implementing transactional applications, concurrent data structures still do not exploit STM's advantages. Recently, Optimistic Transactional Boosting (OTB) has been proposed as a methodology to implement transactional versions of highly concurrent data structures. OTB works in a similar way to STM algorithms, but on the level of data structure semantics. This similarity is a motivation for finding a way to integrate operations of transactional data structures with STM frameworks. In this paper, we extend the design of DEUCE, a Java STM framework, to support OTB integration. Using our extension, programmers can include both OTB data structure operations and traditional memory reads/writes in the same transaction, and the framework will guarantee that both will execute safely as an atomic block. While keeping the same simple interface and the same independence from the JVM as the original DEUCE framework, we allow developers to easily integrate more OTB data structures. As a case study, we show the implementation details of OTB-Set, a transactionally boosted linked-list-based set, and we show how different STM algorithms like NOrec and TL2 can interact with it. Our experiments show up to 10x improvement in the performance of micro-benchmarks over the original DEUCE framework.

***Categories and Subject Descriptors*** D.1.3 [*Programming Techniques*]: Concurrent Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features; E.1 [*Data Structures*]: Concurrent Data Structures

***General Terms*** Algorithms, Performance

***Keywords*** Optimistic Transactional Boosting, Transactional Data Structures, STM frameworks, DEUCE

## 1. Introduction

In-memory transactions can be a potentially significant optimization for managing concurrent requests on shared data structures. The current widely used concurrent collections of elements (e.g., linked-list, skip-list, Tree) are well optimized for preserving isolation of atomic operations, but they do not support transactional access to the objects. Java's *Concurrent Collections* yield high performance for concurrent accesses, but require programmer-defined *synchronized* blocks for demarcating transactions. Such blocks are trivially implemented using coarse-grain locks that significantly limit concurrency.

Software Transactional Memory (STM) [12] is increasingly becoming a promising technology for designing and implementing concurrent applications. They provide a simple interface to develop concurrent applications with strong correctness and progress guarantees (e.g., strong isolation, deadlock freedom). They can also be used to implement transactional data structures and collections. However, pure STM-based transactional collections perform inferior to their optimized, concurrent (non-transactional) counterparts. This is mainly because STM monitors all of the memory locations accessed by a transaction, which results in false conflicts when accessing data structures. For example, if two transactions are trying to insert two different items into a linked-list, these two insertions are commutative, and they are supposed to be executed concurrently without breaking consistency. However, STM may not be able to detect this commutativity, and can raise a false conflict, aborting one of them. In some cases, like long linked-lists, these false conflicts dominate any other overheads in the system.

Recent works in literature propose different ways to implement transactional data structures other than the traditional use of STM algorithms, either by adapting STM algorithms and/or using them more efficiently [1, 4], or by boosting the highly concurrent data structures to be transactional [6, 8].

Transactional boosting was firstly proposed in [8], which builds a layer of semantic locks on top of concurrent data structures. Semantic locks prevent non-commutative operations from conflicting. This reduces false conflicts because an efficient underlying data structure is used instead of monitoring all reads and writes with STM. Recently, the idea of optimistic boosting (OTB) has been proposed as an alternative to the original boosting [6]. Unlike the first proposal of boosting, OTB does not use the underlying concurrent data structures as black boxes. Instead, OTB proposes implementing new transactional versions of the (highly concurrent) lazy data structures, like lazy set [7] and priority queue [9], following the same idea of the original concurrent versions.

All previous proposals, including boosting, do not give details on how to integrate the proposed transactional data structures with STM frameworks. Addressing this issue is important because it allows programmers to combine operations of the efficient transactional data structures with traditional memory reads/writes in the same transaction.

In this paper, we show how to integrate transactionally boosted data structures with the current STM frameworks. One of the

main benefits of optimistic boosting (compared to the original *pessimistic* boosting) is that it uses the terms validation and commit in the same way as many STM algorithms [2, 3], but in the semantic layer. Thus, OTB allows building a system which combines both semantic-based and memory-based validation/commit techniques in a unified consistent framework. More specifically, we show in this paper how to implement OTB data structures in a standard way that can integrate with STM frameworks. We also show how to modify STM frameworks to allow such integration while maintaining the consistency and programmability of the framework.

Using the proposed integration, OTB transactional data structures are supposed to work in the context of generic transactions. That is why the proposed integration gains the benefits of both STM and boosting. On one hand, it uses OTB data structures with their minimal false conflicts and optimal data structure-specific design, which increases their performance. On the other hand, it keeps the same simple STM interface, which increases programmability. To the best of our knowledge, this linking between transactional data structures and STM algorithms has not been investigated in literature before.

We use DEUCE [10] as our base framework. DEUCE is a Java STM framework with a simple programming interface. It allows users to define *@Atomic* functions for the parts of code that are required to be executed transactionally. However, like all other frameworks, using transactional data structures inside *@Atomic* blocks requires implementing pure STM versions, which dramatically degrades the performance. We extend the design of DEUCE to support OTB transactional data structures (with the ability to use the original pure STM way as well). To do so, we integrate two main components into the DEUCE agent. The first component is OTB-DS (or OTB data structure), which is an interface to implement any optimistically boosted data structure. The second component is OTB-STM Context, which extends the original STM context in DEUCE. This new context is used to implement new STM algorithms which are able to communicate with OTB data structures. The new STM algorithms should typically be an extension of the current memory-based STM algorithms in literature.

As a case study, we implement OTB-Set, an optimistically boosted set based on both linked-list and skip-list, inside OTB-DS. Also, we extend two STM algorithms to communicate with OTB-Set (NOrec [2] and TL2 [3]). We select NOrec and TL2 as examples of STM algorithms which use different levels of lock granularity. NOrec is a coarse-grained locking algorithm, which uses a single global lock at commit time to synchronize transactions. TL2, on the other hand, is a fine-grained locking algorithm, which uses ownership records for each memory block. We show in detail how to make the extended design of DEUCE general enough to support both levels of lock granularity.

Our evaluation shows that performance is improved by up to an order of magnitude when we use OTB-Set instead of a pure STM set. Similar performance gain is achieved for both OTB-NOrec and OTB-TL2, especially when false conflicts are frequent, like in linked-list-based sets. Even if false conflicts are rare, like in skip-lists, OTB still performs better, which indicates that the overhead of OTB integration is dominated by the gain of boosting in most cases.

The paper makes the following contributions:

- To the best of our knowledge, this is the first proposal to use optimized transactional data structures inside STM transactions.
- We extend the design of the DEUCE framework to support integration of the optimistically boosted data structures with STM algorithms.
- We describe in detail the design and the implementation of a transactional linked-list-based set using the idea of OTB.
- We extend two STM algorithms in DEUCE (NOrec and TL2) to support integration with boosted data structures.
- Through experiments, we achieve up to 10x improvement in the performance of micro-benchmarks over the original pure STM solutions.

The rest of the paper is organized as follows. In Section 2, we describe the background needed on OTB. In Section 3, we describe the design of the extended DEUCE framework. Section 4 is a case study of integrating OTB-Set with DEUCE using both OTB-NOrec and OTB-TL2. We evaluate our new framework in Section 5, and conclude the paper in Section 6.

## 2. Background: Optimistic Transactional Boosting

Transactional boosting was firstly proposed in [8]. To execute an operation in a boosted data structure, abstract locks are (pessimistically) acquired on the operation's item(s), and then an underlying highly concurrent data structure is used as a black-box to complete the operation. Undo (semantic) logs are used to rollback these operations if the transaction aborts.

The idea of optimistic transactional boosting (OTB) has been introduced in [6][1] as an alternative to pessimistic boosting. It gives guidelines to implement transactional versions of the previously implemented lazy concurrent data structures, like set [7] and priority queue [9] (instead of using lazy data structures as black boxes as in the original boosting).

Like concurrent data structures, operations in OTB versions traverse data structures without instrumentation and validate only the nodes that are semantically involved in the logic of the operation. For example, to insert an item in a concurrent linked-list, the list is traversed without any instrumentation. When the traversal reaches the place in which the new node has to be inserted, abstract locks (or semantic locks)[2] are acquired only on the predecessor and the successor nodes, then the semantic of the list is validated, and finally the new node is physically added.

The main difference between concurrent versions and OTB (transactional) versions is in the steps after data structure traversal. Concurrent versions complete the operation immediately by acquiring semantic locks, validating the data structure semantics, and applying modifications on the shared data structure. On the contrary, OTB data structures delay acquiring semantic locks and any physical modifications on the shared data structure to commit time. To achieve that, OTB saves any necessary information locally in the so-called *semantic read-sets* and *semantic write-sets*. Thus, OTB uses validation and commit procedures in a similar way to most STM algorithms [2, 3, 11], but on the semantic level. In Section 4, we show in detail how to implement an optimistically boosted set using this idea.

OTB data structures are expected to perform better than pure STM data structures, because false conflicts are reduced[3]. Unlike read/write sets in STM, not all memory reads and writes are saved in the semantic read/write sets. Instead, only those reads and writes that affect linearization of the object and consistency of the transaction are saved. For example, in STM-based linked-lists, all traversed nodes are instrumented (while not needed) and may result in false conflicts, especially if the linked-list is relatively long.

[1] We provide more details about OTB in a technical report in the following link: `http://www.hyflow.org/pubs/ppopp_14_TR.pdf`

[2] Semantic locks means locks on data structure nodes, rather than the traditional STM locks on memory locations.

[3] False conflicts are the conflicts that occur in memory when there is no semantic conflict and there is no need to abort.

OTB has been shown to have some benefits over the original boosting methodology. For example, OTB does not require the existence of an inverse for each operation, because operations are not eagerly executed. Moreover, OTB can be easily integrated with STM frameworks, because OTB uses the same terms of validation and commit as most STM algorithms. If a transaction contains both object-level semantic operations and memory-level transactional reads/writes, the whole transaction can be synchronized by monitoring both memory-level and semantic-level read-sets and write-sets, as we show in our proposed framework.

## 3. Extension of DEUCE Framework

DEUCE [10] is a Java STM framework which provides a simple programming interface without any additions to the JVM. It allows programmers to define *atomic* blocks, and guarantees executing these blocks atomically using an underlying set of common STM algorithms (e.g. NOrec [2], TL2 [3], and LSA [11]). We extend DEUCE to support calling OTB data structures' operations along with traditional memory reads and writes in the same transaction, without breaking transaction consistency.

### 3.1 Programming Model

Our framework is designed in a way that integration between data structures' operations and memory accesses is completely hidden from the programmer. For example, a programmer can write an atomic block like that shown in Algorithm 1. In this example, all transactions access a shared set (*set1*), and two shared integers (*n1* and *n2*) which hold the number of successful and unsuccessful add operations on *set1*, respectively. Each thread atomically calls *method1* as a transaction using the *@Atomic* annotation. In *method1*, both set operation (add operation) and traditional memory access (incrementing the shared integers) have to be executed atomically as one block, without breaking consistency, atomicity, or isolation of the transaction.

**Algorithm 1** An example of using Atomic blocks in the new DEUCE framework.

```
1:  Set set1 = new(OTBSet)
2:  integer n1 = 0
3:  integer n2 = 0

4:  @Atomic
5:  procedure METHOD1(x)
6:      if set1.add(x) == true then
7:          n1++
8:      else
9:          n2++
10: end procedure
```

To execute such an atomic block, all previous proposals use a pure STM-based implementation of *set1*, to allow STM frameworks to instrument the whole transaction. Our extended framework, conversely, is the first proposal that uses a more efficient (transactionally boosted) implementation of *set1*, rather than an inefficient STM implementation, and at the same time allows an atomic execution of this kind of transaction.

### 3.2 Framework Design

Figure 1 shows the DEUCE framework with the proposed modifications needed to support OTB integration. For the sake of a complete presentation, we briefly describe in Section 3.2.1 the original building blocks of the DEUCE framework (the white blocks with numbers 1-3). Then, in Section 3.2.2, we describe our additions to the framework to allow OTB integration (gray blocks with numbers 4-7). More details about the original DEUCE framework can be found in [10].

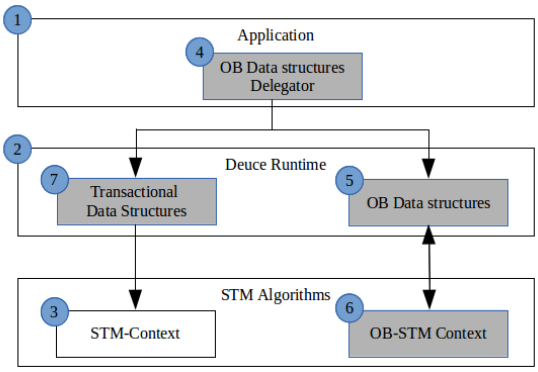



**Figure 1.** New design of DEUCE framework.

#### 3.2.1 Original DEUCE Building Blocks

The original DEUCE framework consists of three layers:

**Application layer.** DEUCE applications do not use any new keywords or any addition to the language. Programmers need only to put an *@Atomic* annotation on the methods that they need to execute as transactions. If programmers want to include in the *@Atomic* blocks some operations that are not transactional by nature, like system calls and I/O operations, DEUCE allows that by using an *@Exclude* annotation. Classes marked as excluded are not instrumented by DEUCE runtime.

**DEUCE runtime layer.** Given this simple application interface (only *@Atomic* and *@Exclude* annotations), DEUCE runtime guarantees that atomic methods will be executed in the context of a transaction. To achieve that, all methods (even if they are not *@Atomic*) are duplicated with an instrumented version, except those in an excluded class. Also, *@Atomic* methods are modified to the form of a retry loop calling the instrumented versions. Some optimizations are made to build these instrumented versions. More details about these optimizations can be found in [10].

**STM context layer.** STM context is an interface which allows programmers to extend the framework with more STM algorithms. DEUCE runtime interacts with STM algorithms using only this interface. Thus, the context interface includes the basic methods for any STM algorithm, like *init*, *commit*, *rollback*, *onReadAccess*, and *onWriteAccess*.

#### 3.2.2 New Building Blocks to Support OTB

The design of our framework extension has three goals: *1)* keeping the simple programming interface of DEUCE; *2)* allowing programmers to integrate OTB data structures' operations with memory reads/writes in the same transaction; and *3)* giving developers a simple API to plug in their own OTB data structures and/or OTB-STM algorithms. To achieve that, we added the following four building blocks to DEUCE framework.

**OTB Data Structures Delegator.** In our new framework design, the application interface is extended with the ability of calling OTB data structures' operations. For example, in Algorithm 1, the user should be able to instantiate *set1*, and call its operations from outside the DEUCE runtime agent. At the same time, OTB data structures have to communicate with STM algorithms to guarantee consistency of the transaction as a whole. This means that OTB data structures have to interact with both the application layer and the DEUCE runtime layer.

To isolate the applications interface from the DEUCE runtime agent, we use two classes for each OTB data structure. The main class, which contains the logic of the data structure, exists in the runtime layer (inside the DEUCE agent). The other class exists at

the application layer (outside the DEUCE agent), and it is just a delegator class which wraps calls to the internal class operations.

This way, the proposed extension in the applications interface does not affect programmability. There is no need for any addition to the language or any modifications in the JVM (like the original DEUCE interface). Also, all synchronization overheads are hidden from the programmer. The only addition is that the programmer should include delegator classes in his application code and call OTB operations through them.

**OTB Data Structures.** Calls from the application interface are of two types. The first type is traditional memory reads/writes, which are directly handled by the OTB-STM context (as described in the next block). The second type is OTB operations, which are handled by a new block added to DEUCE runtime, called OTB-DS (or OTB data structures). The design of an OTB data structure should satisfy the following three points:

- The semantics of the data structure should be preserved. For example, set operations should follow the same logic as if they are executed serially. This is usually guaranteed in optimistic boosting using a validation/commit procedure as shown in [6]. As a case study, in Section 4.1, we show in detail how the semantics of linked-list-based set are satisfied using such a validation/commit procedure.
- Communication between OTB-DS and OTB-STM algorithms. As shown in Figure 1, OTB data structures communicate with STM algorithms in both directions. On one hand, when an OTB operation is executed, it has to validate the previous memory accesses of the transaction, which requires calling routines inside the STM context. On the other hand, if a transaction executes memory reads and/or writes, it may need to validate the OTB operations previously called in the transaction.
- The logic of the underlying STM algorithm, which affects the way of interaction between OTB-DS and OTB-STM context. For example, as we will show in detail in Section 4, OTB-Set interacts with NOrec [2] and TL2 [3] in different ways. In the case of NOrec, which uses a global coarse-grained lock, acquiring semantic locks in OTB-DS may be useless because all transactions are synchronized using the global lock. On the contrary, TL2 uses a fine-grained locking mechanism, which requires OTB-DS to handle fine-grained semantic locks as well. It is worth noting that although a general way of interaction between OTB-DS and OTB-STM can be found, this generality may nullify some optimizations which are specific to each STM algorithm (and each data structure). In this paper we focus on the specific optimizations that can be achieved separately on the two case-study STM algorithms (NOrec and TL2), and we keep the design of a general interaction methodology that works with all STM algorithms as a future work.

To satisfy all of the previous points, while providing a common interface, OTB-DS implements an interface of small sub-routines. These subroutines allow flexible integration between OTB operations and memory reads/writes.

- *preCommit:* which acquires any semantic locks before commit.
- *onCommit:* which commits writes saved in the semantic write-sets.
- *postCommit:* which releases semantic locks after commit.
- *validate-without-locks:* which validates semantics of the data structure without checking the semantic locks' status.
- *validate-with-locks:* which validates both the semantic locks and the semantics of the data structure.

Each OTB-STM context calls these subroutines inside its contexts in a different way, according to the logic of the STM algorithm itself. If a developer designs a new OTB-STM algorithm which needs a different way of interaction, he can extend this interface by adding new subroutines. It is worth noting that an *@Exclude* annotation is used for all OTB-DS classes to inform DEUCE runtime not to instrument their methods.

**OTB-STM Context.** As we showed in Section 3.2.1, STM context is the context in which each transaction will execute. OTB-STM context inherits the original DEUCE STM context to support OTB integration. We use a different context for OTB to preserve the validity of the applications which use the original DEUCE path (through block 7). OTB-STM context adds the following to the original STM context:

- An array of *attached* OTB data structures, which are references to the OTB-DS instances that have to be instrumented inside the transaction. Usually, an OTB data structure is attached when its first operation is called inside the transaction.
- Semantic read-sets and write-sets of each attached OTB data structure. As the context is the handler of the transaction, it has to include all thread local variables, like the semantic read-sets and write-sets.
- Some abstract subroutines which are used to communicate with the OTB-DS layer. In our case study described in Section 4, we only need two new subroutines: *attachSet*, which informs the OTB-STM context to consider the set for any further instrumentation, and *onOperationValidate*, which makes the appropriate validation (at both memory level and semantic level) when an OTB operation is called.

To implement a new OTB-STM algorithm (which is usually a new version of an already existing STM algorithm like NOrec and TL2, not a new STM algorithm from scratch), developers define an OTB-STM context for this algorithm and do the following:

- Modify the methods of the original STM algorithm to cope with the new OTB characteristics. Basically, *init*, *onReadAccess*, *commit*, and *rollback* are modified.
- Implement the new subroutines (*attachSet* and *onOperationValidate*) according to the logic of the STM algorithm.

Like OTB-DS, all OTB-STM contexts have to be annotated with *@Exclude* annotations.

**Transactional Data Structures.** This block is only used to support a unified application interface for both OTB-STM algorithms and traditional STM algorithms. If the programmer uses a traditional (non-OTB) STM algorithm and calls an OTB-DS operation inside the transaction, DEUCE runtime will use a traditional pure STM implementation of the data structure to handle this operation, and it will not use optimistic boosting anymore.

## 4. Integrating OTB-Set with NOrec and TL2

Following the framework design in Section 3, we show a case study on how to integrate an optimistically boosted version of a linked-list-based set in the modified DEUCE framework[4]. This is done using the following two steps:

- Implementing OTB-Set (in Section 4.1), which inherits OTB-DS interface and follows its guidelines. We already showed some general guidelines on how to implement OTB-Set in [6]. But here we give more implementation details, more optimiza-

[4] Skip-list-based OTB-Set is implemented in a similar way with few modifications. For space limitations, we only show the linked-list version. More details can be found in the technical report.

tions and special cases, and more details about integration with DEUCE[5].

- Implementing OTB-STM algorithms (Sections 4.2 and 4.3), which interact with the new OTB-Set. We use two algorithms in this case study, NOrec and TL2. As we showed in Section 3, we will need to implement a new OTB-STM context for both algorithms[6].

### 4.1 Optimistic Boosted Set

Sets are collections of items which have three basic operations: `add`, `remove`, and `contains`, with the familiar meanings. 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 query operations on the same item are commutative as well. Such a high degree of commutativity between operations enables fine-grained semantic synchronization.

Linked-list-based OTB-Set is the transactional version of the lazy concurrent linked-list-based set [7]. For each operation, the list is traversed without any instrumentation until the involved nodes are reached. Each operation in OTB-Set involves two nodes (which are used during validation and commit): $pred$, 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 (sentinel nodes are added as head and tail of the list to handle special cases). To save needed information about these nodes, we use the underlying OTB-DS semantic read-sets and write-sets. In particular, each read-set entry contains the two involved nodes in the operation and the type of the operation. Each write-set entry contains the same items, and also includes the new value to be added in case of a successful `add` operation. The `add` and `remove` operations are not necessarily considered as writing operations, because duplicated items are not allowed in the set. This means that both `contains` and unsuccessful `add/remove` operations are considered as read operations (which just add entries to the semantic read-set). Only successful `add` and `remove` operations are considered read/write operations (which add entries to both the read-set and the write-set).

Algorithm 2 shows the pseudo code of the linked-list operations. We can isolate five steps of each operation:

- **Attaching set to STM context** (line 2). This is done by calling the underlying OTB-STM context's operation *attachSet*.
- **Local writes check** (lines 3-14). Since writes are buffered and deferred to the commit phase, this step guarantees consistency of further reads and writes. 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. Furthermore, if a transaction adds an item and then removes the same item, or vice versa, operations locally eliminate each other (lines 9 and 14). This elimination can further improve optimistic boosting's performance. Elimination only removes the entries from the write-set and leaves the read-set entries as they are, to maintain transaction isolation by validating the eliminated operations at commit.
- **Traversal** (lines 15-18). This step is exactly the same as in 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.

**Algorithm 2** Linked-list: `add`, `remove`, and `contains` operations.

```
1:  procedure OPERATION(x)
                                          ▷ Step 1: Attach set to OTB-STM context
2:      OTB-STM-Context.attachSet(this)
                                          ▷ Step 2: search local write-sets
3:      if x ∈ write-set and write-set entry is add then
4:          if operation = add then
5:              return false
6:          else if operation = contains then
7:              return true
8:          else                                                  ▷ remove
9:              delete write-set entry & 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 & return true
                                          ▷ Step 3: Traversal
15:     pred = head and curr = head.next
16:     while curr.item < x do
17:         pred = curr
18:         curr = curr.nerxt
                                          ▷ Step 4: Post Validation
19:     OTB-STM-Context.onOTBOperationValidate()
                                          ▷ Step 5: Save reads and writes
20:     Compare curr.item with x and check curr.deleted
21:     if Successful add/remove then
22:         read-set.add(new ReadSetEntry(pred, curr, operation))
23:         write-set.add(new WriteSetEntry(pred, curr, operation, x))
24:         return true
25:     else if Successful contains then
26:         read-set.add(new ReadSetEntry(pred, curr, operation))
27:         return true
28:     else if Unsuccessful operation then
29:         read-set.add(new ReadSetEntry(pred, curr, operation))
30:         return false
31: end procedure
```

- **Validation** (line 19). At the end of the traversal step, the involved nodes are found in local variables (i.e., $pred$ and $curr$). At this point, to avoid breaking opacity [5], the transaction must be post-validated to ensure that it does not see any inconsistent snapshot. This validation is done by calling the OTB-STM context method *onOperationValidate*, which guarantees validating all of the attached OTB-Sets as well as making the necessary memory-level validations.
- **Saving reads and writes** (lines 20-30). At this point, the transaction can decide on the return value of the operation (line 20). Then, it modifies its read and write sets. Although all operations must add the appropriate read-set entry, only the successful `add/remove` operations modify the write-set (line 23).

#### 4.1.1 OTB-DS interface methods

OTB-Set is communicating with the context of the underlying OTB-STM algorithm using the subroutines of the OTB-DS interface. OTB-Set implements these subroutines as follows:

**Validation:** Transactions validate that read-set entries are semantically valid. In addition, to maintain isolation, a transaction has to ensure that all nodes in its semantic read-set are not locked by another writing transaction during validation. As it is not always the case (in some cases, semantic locks are not validated, as shown in the next section), it is important to make two versions of validation:

- *validate-without-locks*: This method validates the read-set entries in a similar way to lazy linked-list. Both $pred$ and $curr$ should not be deleted, and $pred$ should still link to $curr$. The only difference is in the case of successful `contains` and unsuccessful `add`, in which a simpler validation is used. In these particular cases, the transaction only needs to check that $curr$ is still not deleted, since that is sufficient to guarantee that the

[5] The full implementation details are in the technical report.

[6] Note that the original STM contexts of NOrec and TL2 can still be used in our framework (using block 7 in Figure 1), but they will use an STM-based implementation of the set rather than our optimized OTB-Set.

returned value is still valid (recall that in lazy list, as well as in OTB list, 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.

- *validate-with-locks*: This version did exactly the same function as the previous one, with the addition of a validation on the semantic locks as well. This is achieved by implementing locks as *sequence locks* (i.e., locks with version numbers). Before validation, a transaction takes a snapshot of the locks and makes sure that they are unlocked. After validation, it ensures that the lock versions are still the same.

**Commit:** To be flexible when integrating with the STM contexts, commit consists of the following subroutines:

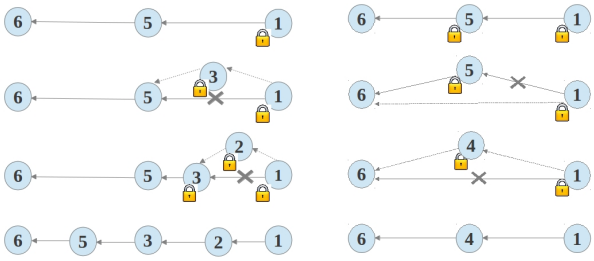
**Algorithm 3** Linked-list: onCommit.

```
1:  procedure ONCOMMIT
2:      sort write-set descending on items
3:      for all entries in write-sets do
4:          curr = pred.next
5:          while curr.item < x do
6:              pred = curr
7:              curr = curr.next
8:          if operation = add then
9:              n = new Node(item)
10:             n.locked = true
11:             n.next = curr
12:             pred.next = n
13:         else                                   ▷ remove
14:             curr.deleted = true
15:             pred.next = curr.next
16: end procedure
```

- *preCommit*: which acquires the necessary semantic locks on the write-sets. Like lazy linked-list: any `add` operation only needs to lock $pred$, while `remove` operations lock both $pred$ and $curr$. This can be easily proven to guarantee consistency, as described in [7].
- *postCommit*: which releases the acquired semantic locks after commit.
- *onAbort*: which releases any acquired semantic locks not yet released when abort is called.
- *onCommit*: which publishes writes on the shared linked-list. This step is not trivial, because each node may be involved in more than one operation in the same transaction. In these cases, the saved $pred$ and $curr$ of these operations may change according to which operation commits first. For example, in Figure 2(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 saves node 1 as the predecessor of node 3.
  To ensure that operations in this case are executed correctly, three guideline points are followed (described in Algorithm 3):
  1. Inserted nodes are locked until the whole commit procedure is finished. Then they are unlocked along with the other $pred$ and $curr$ nodes (line 10).
  2. The items are added/removed in descending order of their values, regardless of their order in the transaction execution (line 2). This guarantees that each operation starts the commit phase from a valid non-deleted node.
  3. Operations resume traversal from the saved $pred$ to the new $pred$ and $curr$ nodes (which can only be changed by operations in the same transaction, otherwise the transaction will abort). Lines 4-7 encapsulate the logic for this case.

Using these three points, the issue in Figure 2(a) will be solved. According to the first point, all nodes (1, 2, 3, 5) are locked and no transaction can access them until commit is finished (any transaction will abort if it tries to access these nodes). The second point enforces that node 3 is inserted first. Subsequently, according to the third point, when 2 is inserted, the transaction will resume its traversal from node 1 (which is guaranteed to be locked and non-deleted). It will then detect that node 3 is its new $succ$, and will correctly link node 2.

The removal case is shown in Figure 2(b), in which node 5 is removed and node 4 is inserted. Again, 5 must be removed 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. The same procedure holds for the case of the two subsequent `remove` operations.



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

**Figure 2.** Executing more than one operation that involves the same node in the same transaction.

### 4.2 Integration with NOrec

NOrec [2] is an STM algorithm which uses a single global lock at commit time to synchronize transactions. Validation on NOrec is incremental and value-based. Each transaction validates its read-set after each read if it finds that the global timestamp has changed, and then it extends its local timestamp if validation succeeds. At commit time, the write-set is published on the shared memory.

To integrate NOrec with OTB-Set, two main observations have to be taken into consideration. First, using a single global lock to synchronize memory reads/writes can be exploited to remove the overhead of the fine-grained semantic locks as well. Semantic validation has to use the same global lock because in some cases the validation process includes both semantic operations and memory-level reads. As a result, there is no need to use any semantic locks given that the whole process is synchronized using the global lock. Second, both NOrec and OTB-Set use some kind of value-based validation. There are no timestamps attached with each memory block (like TL2 for example). This means that both NOrec and OTB-Set require an incremental validation to guarantee opacity [5]. They both do the incremental validation in a similar way, which makes the integration straightforward.

The implementation of OTB-NOrec context subroutines is as follows[7]:

***init:*** In addition to clearing the memory-based read-set and write-set, each transaction should clear the semantic read-sets and write-sets of all previously attached OTB-Sets, and then it detaches all of these OTB-Sets to start a new empty transaction.

***attachSet:*** This procedure is called in the beginning of each set operation (which is previously shown in Algorithm 2). It simply checks if the set is previously attached, and adds it to the local array of the attached sets if it is not yet attached.

***onOperationValidate:*** As both memory reads and semantic operations are synchronized and validated in the same way (using the

[7] We skipped the implementation details of NOrec itself (and TL2 in the next section), and concentrate only on the modifications we made on the context to support OTB.

global lock and a value based validation), this method executes the same procedure as *onReadAccess*, which loops until the global lock is not acquired by any transaction, and then it calls the *validate* subroutine.

***validate:*** This private method is called on both *onReadAccess* and *onOperationValidate*. It simply validates the memory-based read-set as usual, and then validates the semantic read-sets of all the attached OTB-Sets. This validation is done using the *validate-without-locks* subroutine, which is described in Section 3.2.1, because there is no use of the semantic locks in OTB-NOrec context. If validation fails in any step, an abort exception is thrown.

***commit:*** There is no need to call the attached OTB-Sets' *preCommit* and *postCommit* subroutines during transaction commit. Again, this is because these subroutines deal with semantic locks, which are useless here. The commit routine simply acquires the global lock, validates read-sets (both memory and semantic read-sets) using the *validate* subroutine, and then starts publishing the writes in the shared memory. After the transaction publishes the memory-based write-set, it calls the *onCommit* subroutine in all of the attached OTB-Sets, and then it releases the global lock.

***rollback:*** Like *preCommit* and *postCommit*, there is no need to call OTB-Set's *onAbort* subroutine during the rollback.

### 4.3 Integration with TL2

TL2 [3], as opposed to NOrec, uses a fine-grained locking mechanism. Each memory block has a different lock. Reads and writes are synchronized by comparing these locks with a global version-clock. Also, unlike NOrec, validation after each read is not incremental. There is no need to validate the whole read-set after each read. Only the lock version of the currently read memory block is validated. The whole read-set is validated only at commit time and after acquiring all locks on the write-set.

Thus, the integration with OTB-Set requires validation and acquisition of the semantic locks in all steps. That is why we provide two versions of validation (with and without locks) in the layer of OTB-DS. The implementation of OTB-TL2 context subroutines is as follows:

***init:*** It is extended in the same way as OTB-NOrec.

***attachSet:*** It is implemented in the same way as OTB-NOrec.

***onOperationValidate:*** There are two differences between OTB-NOrec and OTB-TL2 in the validation process. First, there is no need to validate the memory-based read-set when an OTB-Set operation is called. This is basically because TL2 does not use an incremental validation, and OTB-Set operations are independent from memory reads. Second, OTB-Sets should use the *validate-with-locks* subroutine instead of *validate-without-locks*, because semantic locks are acquired during commit.

***onReadAccess:*** Like *onOperationValidate*, this subroutine has to call the *validation-with-locks* subroutines of all of the attached sets in addition to the original memory-based validation.

***commit:*** Unlike OTB-NOrec, semantic locks have to be considered for the attached OTB-Sets. Thus, *preCommit* is called for of all the attached OTB-Sets right after acquiring the memory-based locks, so as to acquire the semantic locks as well. If *preCommit* of any set fails, an abort exception is thrown. During validation, OTB-TL2 context calls the *validate-with-locks* subroutine of the attached sets, instead of *validate-without-locks*. Finally, *postCommit* subroutines are called to release semantic locks.

***rollback:*** Unlike OTB-NOrec, *onAbort* subroutines of the attached sets have to be called to release any semantic locks that are not yet released.

## 5. Evaluation

We now evaluate the performance of the modified framework using a set micro-benchmark. In each experiment, 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 experiment was run five times and the arithmetic average is reported as the final result.

The experiments were conducted on a 48-core machine, which has four AMD Opteron (TM) Processors, each with 12 cores running at 1400 MHz, 32 GB of memory, and 16KB L1 data cache. The machine runs Ubuntu Linux 10.04 LTS 64-bit.

In each experiment, we compare the modified OTB-NOrec and OTB-TL2 algorithms (which are internally calling OTB-Set operations) with the traditional NOrec and TL2 algorithms (which internally call a pure STM version of the set).

We run two different benchmarks. The first one is the default set benchmark in DEUCE. In this benchmark, each set operation is executed in a transaction. This benchmark evaluates the gains from using OTB data structures instead of the pure STM versions. However, they do not test transactions which call both OTB operations and memory reads/writes. We developed another benchmark (which is a modified version of the previous one) to test such cases. In this second benchmark, as shown in Algorithm 1, each transaction calls an OTB-Set operation (add, remove, or contains), and increment some shared variables to calculate the number of successful and unsuccessful operations. As a result, both OTB-Set operations and increment statements are executed atomically. We justify the correctness of the transaction execution by comparing the calculated variables with the (non-transactionally calculated) results from DEUCE benchmark.

### 5.1 Linked-List Micro-Benchmark

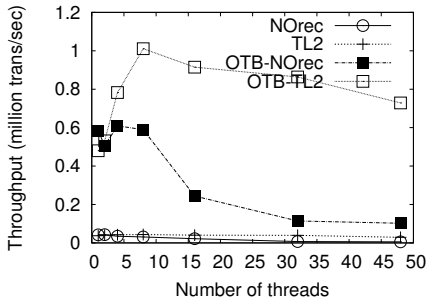
Figure 3 shows the results for a linked-list with size 512. Both OTB-NOrec and OTB-TL2 show a significant improvement over their original algorithms, up to an order of magnitude of improvement. This is reasonable because pure STM-based linked-lists have a lot of false conflicts, as we described earlier. Avoiding false conflicts in OTB-Set is the main reason for this significant improvement. The gap is more clear in the single-thread case, as a consequence of the significant decrease in the instrumented (and hence logged) reads and writes. It is worth noting that in both versions (with and without OTB), TL2 scales better than NOrec, because NOrec is a conservative algorithm which serializes commit phases using a single lock.

### 5.2 Skip-List Micro-Benchmark

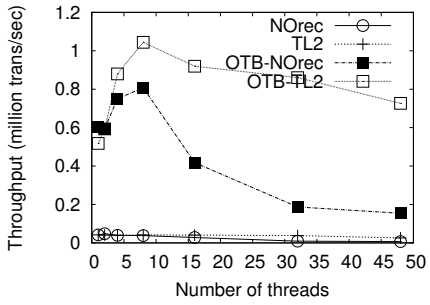
Results for skip-list are shown in Figure 4. Skip-lists do not usually have the same number of false conflicts as linked-lists. This is because traversing a skip-list is logarithmic, and the probability of modifying the higher levels in a skip-list is very small. That is why the gap between OTB versions and the original STM versions is not as large as for linked-lists. However, OTB versions still perform better in general. OTB-NOrec is better in all cases, and it performs up to 5x better than NOrec for a small number of threads. OTB-TL2 is better than TL2 for a small number of threads, and it is almost the same (or slightly worse) for a high number of threads. Performance gain for a small number of threads is better because false conflicts still have an effect on the performance. For higher numbers of threads contention increases, which reduces the ratio of false conflicts compared to the real conflicts. This reduction in false conflicts reduces the impact of boosting, which increases the influence of the integration mechanism itself. That's why OTB-TL2 is slightly worse. However, plots in general show that the gain of saving false conflicts dominates this overhead in most cases.

### 5.3 Integration Test Case

In this benchmark, we have six shared variables in addition to the shared OTB-Set (number of successful/unsuccessful adds, re-

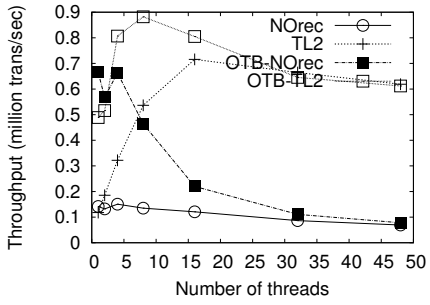



(a) 80% add/remove, 20% contains

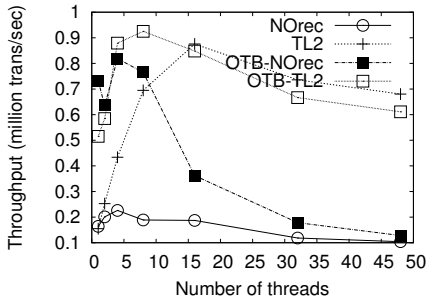



(b) 50% add/remove, 50% contains

**Figure 3.** Throughput of linked-list-based set with 512 elements, for two different workloads.



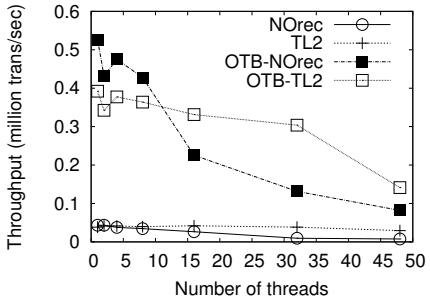

(a) 80% add/remove, 20% contains



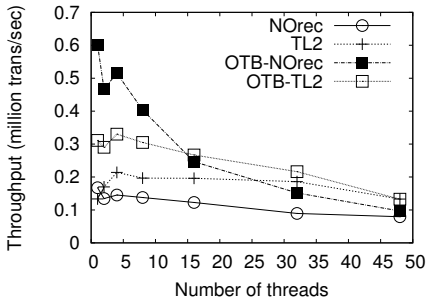

(b) 50% add/remove, 50% contains

**Figure 4.** Throughput of skip-list-based set with 4K elements, for two different workloads.

moves and contains). Each transaction executes a set operation (50% reads) and then it increments one of these six variables according to the type of the operation and its return value. As all transactions are now executing writes on few memory locations, contention increases and performance degrades on all algorithms. However, OTB-NOrec and OTB-TL2 still give better performance than their corresponding algorithms. The calculated numbers match the summaries of DEUCE, which justifies the correctness of the transactions. Also, NOrec versions relatively perform like the previous case (without increment statements), compared to TL2 versions. This is because NOrec (like all coarse-grained algorithms) works well when transactions are conflicting by nature.




(a) linked-list




(b) skip-list

**Figure 5.** Throughput of Algorithm 1 (a test case for integrating OTB-Set operations with memory reads/writes). Set operations are 50% add/remove and 50% contains.

## 6. Conclusions

We presented an extension of the DEUCE framework to support integration with transactional data structures that are implemented using the idea of *Optimistic Transactional Boosting*. As a case study, we implemented OTB-Set, an optimistically boosted linked-list-based set, and showed how it can be integrated in the modified framework. We then show how to adapt two different STM algorithms (NOrec and TL2) to support this integration. As a future work, this case study can be generalized with some slight modifications and some relaxations in the STM-algorithm-specific optimizations proposed in this paper. Performance of micro-benchmarks using the modified framework is improved by up to 10x over the original framework.

## References

[1] 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 (PPoPP)*, pages 161–170. ACM, 2012.

[2] L. Dalessandro, M. Spear, and M. Scott. Norec: streamlining stm by abolishing ownership records. In *Proceedings of the 15th ACM SIG-*

*PLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 67–78. ACM, 2010.

[3] D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. In *Proceedings of the 20th international symposium on Distributed Computing (DISC)*, pages 194–208. Springer, 2006.

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

[5] 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 (PPoPP)*, pages 175–184. ACM, 2008.

[6] A. Hassan, R. Palmieri, and B. Ravindran. Optimistic transactional boosting. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and practice of parallel programming (PPoPP), Poster paper*, 2014.

[7] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. Scherer, and N. Shavit. A lazy concurrent list-based set algorithm. *Proceedings of the 9th International Conference on Principles of Distributed Systems*, pages 3–16, 2006.

[8] 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 (PPoPP)*, pages 207–216. ACM, 2008.

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

[10] G. Korland, N. Shavit, and P. Felber. Noninvasive concurrency with java stm. In *Third Workshop on Programmability Issues for Multi-Core Computers (MULTIPROG)*, 2010.

[11] T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. *Proceedings of the 20th international symposium on Distributed Computing (DISC)*, pages 284–298, 2006.

[12] N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.