Optimistic Transactional Boosting

Ahmed Hassan
Virginia Tech
hassan84@vt.edu

Roberto Palmieri
Virginia Tech
robertop@vt.edu

Binoy Ravindran
Virginia Tech
binoy@vt.edu

Abstract

Herlihy and Koskinen's transactional boosting methodology addressed the challenge of converting concurrent data structures into transactional ones. We present an optimistic methodology for boosting concurrent collections. Optimistic boosting allows greater data structure-specific optimizations, easier integration with STM frameworks, and lower restrictions on the boosted operations than the original boosting methodology.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming – Parallel Programming; E.1 [*Data Structures*]: Concurrent Data Structures

Keywords STM, Transactional Boosting, Transactional Data Structures

1. Introduction

Concurrent collections of elements [3] are well optimized for preserving isolation of concurrent operations, but they do not support transactional accesses to objects. Software Transactional Memory (STM) [4] is increasingly becoming a promising technology for designing and implementing concurrent applications. STM can be trivially used for implementing transactional data structures and collections. However, the performance of STM-based transactional collections is significantly lower than their optimized, concurrent (non-transactional) counterparts.

As an alternative to using STM, Herlihy and Koskinen introduced the technique of *Transactional Boosting* [2], which converts concurrent data structures to transactional ones by providing a *semantic layer* of abstract locks on top of concurrent objects. However, it has some downsides. First, abstract lock acquisition and modifications in memory are eager. This does not natively provide opacity [1] at memory level and contradicts with the methodology of most STM algorithms, which makes the integration between "boosted" data structures and any STM framework difficult. Second, the technique uses the underlying concurrent data structure as a black box, which prevents further optimizations. Finally, the methodology requires defining an inverse for each operation, which is not necessarily supported in all data structures.

Motivated by these observations, we present *Optimistic Transactional Boosting* (or OTB), an optimistic methodology for converting concurrent data structures into transactional ones. In OTB, transactional operations do not eagerly acquire the semantic locks and modify the shared data structure. Instead, they populate their changes in local logs during their execution, defering any physical modifications to commit time. This way, OTB combines the benefits of lazy concurrent data structures (i.e., un-monitored traversals), boosting (i.e., semantic validation), and transactional memory (i.e., optimistic concurrency control).

OTB gains significant advantages over Herlihy and Koskinen's boosting, which we call "pessimistic" boosting hereafter due to its pessimistic behavior on lock acquisition. First, it avoids the need for defining inverse operations. Second, it uses the concepts of validation, commit, and abort in the same way as general (optimistic) STM algorithms, but at the semantic layer, which enables easy integration with STM frameworks. Finally, it uses highly concurrent collections as white boxes (rather than black boxes as in pessimistic boosting) for designing new transactional versions of each concurrent (non-transactional) data structure, which allows more data structure-specific optimizations.

2. OTB Methodology

Each operation in OTB is divided into three steps:

Traversal. This step scans the objects and computes what the operation's results should be (its postcondition), and what it depends on (its precondition). This raises the need to define (in each transaction), what we call, *semantic read-set* and *semantic write-set*, which store these information.

Validation. This step checks the validity of preconditions. Specifically, the entities stored in the semantic read-set are validated in this step. The step is repeated after each new read (to guarantee opacity), and at commit time.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

PPoPP '14, February 15–19, 2014, Orlando, Florida, USA.
Copyright is held by the owner/author(s).
ACM 978-1-4503-2656-8/14/02.
http://dx.doi.org/10.1145/2555243.2555283

Commit. This step performs the modifications to the shared data structure. The step is not done at the end of each operation, but is deferred to 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 write-sets, classical two-phase locking is used (but at the semantic layer). This semantic (or abstract) locking prevents semantic conflicts at commit.

Unlike the classical meaning of read-sets and write-sets in STM, not all memory reads and writes are saved in the semantic read-sets and write-sets. Instead, only those reads and writes that affect linearization of the object and consistency of the transaction are saved. This avoids *false conflicts* – i.e., concurrent operations that conflict at the memory level but are independent at the semantic level; thus, they do not require any abort, which degrades the performance of several STM-based data structures.

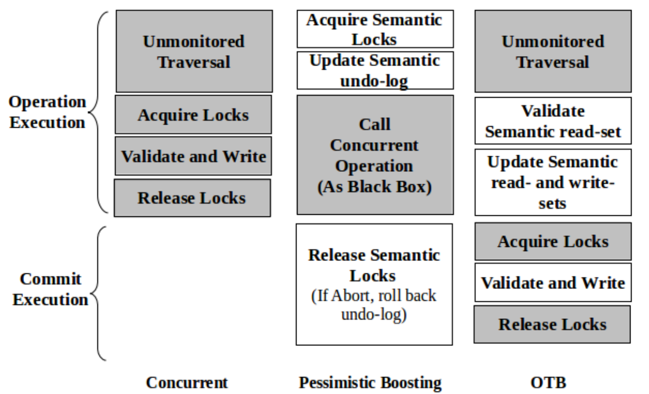

Figure 1. Execution flow of: concurrent (lock-based or lock-free) data structures; pessimistic boosting; OTB.

Figure 1 shows the execution flow of: concurrent (lazy) data structures, pessimistic boosting, and OTB.

Concurrent (non-transactional) data structures yield high performance because they traverse the data structure without instrumentation, and they only acquire locks (or use CAS operations in case of lock-free objects) at late phases.

To add transactional capabilities, pessimistic boosting acquires semantic locks eagerly, and saves the inverse operations in an undo-log (to rollback the transaction in case of abort). Then, it uses the underlying concurrent data structure as a black box without any modifications. (In both pessimistic boosting and OTB, dark blocks in Figure 1 are the same as the concurrent versions, while white blocks are added/modified.) At commit time, the only task to be accomplished is the release of semantic locks, because operations have already been executed eagerly.

In contrast to pessimistic boosting, OTB acquires semantic locks lazily, and uses the underlying data structure as a white box. Similar to concurrent data structures, OTB traverses objects without instrumentation. However, it differs from them in three aspects: *i)* lock acquisition and actual writes are shifted to commit time; *ii)* the validation procedure is modified to satisfy the new transactional requirements; and *iii)* the necessary information is saved in local semantic read-sets and write-sets.

Thus, OTB gains the following benefits over pessimistic boosting. First, it does not require well defined commutativity rules or inverse operations. Second, integration with STM frameworks is easy, as OTB uses the same phases of validation and commit (with the same meaning as in STM). Third, it uses highly concurrent collections as white boxes to design new transactional versions of each concurrent (non-transactional) data structure. This allows greater optimizations according to the new transactional features, with minimal re-engineering overhead.

3. Example: OTB Set

Following the steps in Section 2, applying optimistic boosting on lazy linked list-based set [3] is straightforward. We briefly explain the guidelines for applying the three steps of optimistic boosting on set.

Traversal. Any operation (add, remove, or contains) on an item X traverses the list without instrumentation until it reaches the involved nodes. As in a lazy set, two nodes are involved in each operation: *pred*, the largest node lower than X; and *curr*, the successor of *pred*. These nodes are saved in the read-set (and the write-set if the operation is a successful add/remove operation). To cover the read-after-write hazard, the write-set is scanned before the list traversal.

Validation. List is validated in a similar way as its concurrent version [3]. The validity of *pred* and *curr* is checked: they should not have been deleted, and the reference of *pred* should point to *curr*. The only difference is that the entire read-set must be post-validated after each operation, and not just the current operation's nodes (to guarantee opacity).

Commit. Semantic locks on all *pred* and *curr* nodes in the write-set are acquired. Then, read-set is validated again, and the write-set is published. Finally, locks are released.

Acknowledgments

The authors would like to thank Maurice Herlihy for his comments and suggestions. This work is supported in part by US National Science Foundation under grant CNS 1116190.

References

[1] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *PPoPP*, 2008.

[2] M. Herlihy and E. Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *PPoPP*, 2008.

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

[4] N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 10(2), 1997.