Brief Announcement: Semantic Conflict Detection for Transactional Data Structure Libraries

Yaodong Sheng
Lehigh University
Bethlehem, PA, USA
yas616@lehigh.edu

Ahmed Hassan
Lehigh University
Bethlehem, PA, USA
ahh319@lehigh.edu

Michael Spear
Lehigh University
Bethlehem, PA, USA
spear@cse.lehigh.edu

ABSTRACT

The Transactional Data Structure Library (TDSL) methodology improves the programmability and performance of concurrent software by making it possible for programmers to compose multiple concurrent data structure operations into coarse-grained transactions. Like transactional memory, TDSL enables arbitrarily many operations on arbitrarily many data structures to appear to other threads as a single atomic, isolated transaction. Like concurrent data structures, the individual operations on a TDSL data structure are optimized to avoid artificial contention.

We introduce techniques for reducing false conflicts in TDSL implementations. Our approach allows expressing the postconditions of operations entirely via semantic properties, instead of through low-level structural properties. Our design is general enough to support lists, deques, ordered and unordered maps, and vectors. It supports richer programming interfaces than are available in existing TDSL implementations. It is also capable of precise memory management, which is necessary in low-level languages like C++.

CCS CONCEPTS

• **Information systems** → Data structures.

KEYWORDS

Transactional memory; Concurrent data structures; Synchronization

ACM Reference Format:
Yaodong Sheng, Ahmed Hassan, and Michael Spear. 2021. Brief Announcement: Semantic Conflict Detection for Transactional Data Structure Libraries. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '21), July 6–8, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3409964.3461826

1 INTRODUCTION

Composition is a significant challenge when writing concurrent software. Transactional Memory (TM) provides an easy mechanism for combining several data structure operations into coarse, serializable *transactions* [5, 7]. However, these composed transactions can struggle to scale, because TM treats *any* memory conflict among transactions as a true conflict, which causes at least one of the conflicting transactions to abort.

Some TM systems can "forget" certain memory conflicts [2, 4]. However, these techniques require more cleverness than is appropriate for a language mechanism that is supposed to make it *easy* to write correct concurrent code. A more promising technique is to use a transactional *style* of programming, but not TM. The Transactional Data Structure Library (TDSL) [8] provides several hand-crafted data structures. A programmer can start a transaction, interact with multiple TDSL data structures, and then commit. The data structure implementations themselves are responsible for ensuring correctness while avoiding conflicts for operations that ought to commute. They also are designed such that operations can be rolled back. A complementary approach, the Lock-Free Transactional Transformation (LFTT), provides a similar functionality, with nonblocking progress guarantees [10].

These works tightly coupled their data structures with the transaction management code. For example, LFTT's transaction management only knows about the semantics of insert, lookup, and remove operations, and uses this knowledge to detect conflicts. TDSL's transaction management has explicit functionality for managing a skip list's index nodes outside of the logical transaction. Tight coupling makes it difficult to add new features (e.g., range queries) and new data structures (e.g., hash tables).

We present a refined approach to composable concurrent data structures with three key components:

- The use of TM as an implementation technique within data structures (§2): This ensures deadlock freedom for complex operations (e.g., multiple writes in a doubly linked list), memory safety, and straightforward reasoning about correctness.
- Enhanced metadata that expresses ownership, structural versioning, and semantic versioning (§3): This avoids false aborts for commutative operations, and naturally supports range queries and in-place modification of shared data.
- Lightweight complier support, based on techniques from TM [9] (§4): This enables our data structures to support arbitrary data types without increasing programmer burden.

2 TRANSACTIONAL DOUBLY LINKED LISTS

Concurrent data structures often rely on a single compare-and-swap as the linearization point [6] for an operation that entails several memory updates [3]. Our technique uses TM to implement *multi-word* compare-and-swap. This simplifies the synchronization protocol. Furthermore, by varying the TM implementation, we can explore different tradeoffs between latency, precision of memory reclamation, and progress guarantees.

SPAA '21, July 6–8, 2021, Virtual Event, USA.

ACM ISBN 978-1-4503-8070-6/21/07.
https://doi.org/10.1145/3409964.3461826

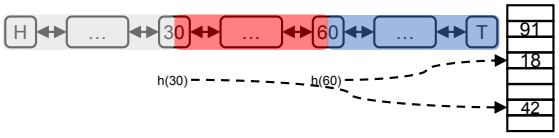

Figure 1: List traversal with multiple transactions.

This design makes complex data structures (e.g., doubly linked lists) easy to synchronize. However, TM does not allow a datum to be read nontransactionally if a concurrent thread is using a transaction to update it. Thus the read-only traversal phase of our data structure operations must also use TM. This can create false contention and false aborts. We extend the idea of telescoping transactions [1, 11]. The general idea, as show in Figure 1, is to employ a sequence of traversals of contiguous segments of the data structure. In the figure, the first segment iterates from the head sentinel to node 30, the second iterates from node 30 to node 60, and the third iterates from node 60 to the tail sentinel.

Breaking the transaction into segments is not trivial. When the first segment commits, another thread could remove node 30, which would invalidate the second segment of the traversal. We adopt the approach in [11]: Each node hashes to an entry in an array of integers, which is incremented when the node is unlinked from the table. Traversals can sample the appropriate table entry before ending one segment, and then check it upon starting the next. If the value is unchanged, the operation can pick up where it left off. We call these integers *fenceposts*.

Our design decouples the traversal segments of an operation from the final segment that reads or modifies the node that it finds. That is, in [11], an insertion's final search segment would stitch a node into the data structure, whereas our stitch operation is a separate, subsequent transaction. To make this work, we require fencepost increments during removal and insertion (prior work only required increments during removal). That is, our fenceposts do not only indicate that some node is *present* in the list; they also indicate that some nodes are *absent*.

3 CONFLICT DETECTION WITH VARIABLE PRECISION

§ 2 provides a scalable, low-contention list. We now extend it with a transactional data structure interface that defines conflicts in terms of semantic information, instead of structural information. We augment data structure nodes with five fields:

- owner: the thread currently performing a semantic operation on the node
- state: one of 10 values, discussed below
- semantic: a counter that tracks the number of *semantic* changes to the node.
- structural: a counter that tracks the number of *structural* changes to the node.
- subscribers: a count of transactions that are monitoring changes to semantic and structural.

In TM and TDSL, ownership is a binary property: either a location is owned, in which case the owner has exclusive access to that location, or else that location is not owned, in which case all threads may read that location. We expand ownership to ten states (four are related to range queries), which express semantic information. This allows operations to *eagerly* acquire locations, without impeding non-conflicting concurrent operations.

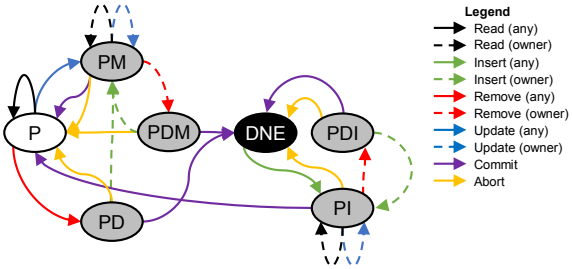

Figure 2: Node ownership state transitions.

Figure 2 shows the transitions among states. We use DNE as a shorthand for when a key does not exist. The states are as follows:

- P (present): the node is not owned. A transaction may read it, but may not remove it or modify it.
- PI (pending-insert): the node is being inserted by a pending transaction. To all transactions other than the owner, the node is effectively not present.
- PD (pending-delete): the node is being removed by a pending transaction. To all transactions other than the owner, the node is effectively present.
- PM (pending-modify): the node being modified by a pending transaction (e.g., modify the value of a key/value pair).
- PDI (pending-delete-insert): a transaction intended to insert this node but subsequently decided to remove it.
- PDM (pending-delete-modify): a transaction modified this node but subsequently decided to remove it.

These states preserve concurrency in the face of contention. For example, if some transaction owned a node in a sorted list with key 60, TM and TDSL would forbid other transactions from accessing that node, or traversing past it. In our design, another transaction searching for 64 could traverse through that node. Indeed, if the owner was modifying the value at key 60, even a concurrent existence check for key 60 could be allowed to succeed.

We illustrate the benefit of these states by considering operations that return false (e.g., insertion of a key that already exists, lookup or removal of a key that does not exist). In these cases, there is no natural location in the data structure for storing semantic information. Figure 3 depicts three alternatives that we support. In all cases, a thread is searching for node 62, which does not exist. The first way to represent this absence (Struct) is structurally, by recording the fact that 60 points to 64. This is the only way to represent non-existence in TM and TDSL. In our design, a failed lookup can monitor the structural version of 60. If 60 is removed, or if anything is inserted between 60 and 64, the structural version will change, causing an abort.

In the row labeled Range, we introduce a range node type. Range nodes hold two keys and no value. The state pattern $F, L_{E,I}$ encodes whether the First key of the range is Inclusive or Exclusive, and

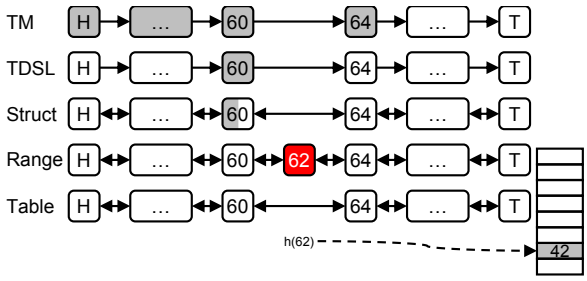

Figure 3: Unsuccessful `list::contains(62)`

whether the Last key is Inclusive or Exclusive. Expressing inclusivity/exclusivity is necessary when keys are not enumerable (e.g., floats, strings). A $F_I L_I$ range node where key1 = key2 = 62 can be used to encode that 62 is not in the list. An operation that returns false can insert such a node, which will prevent other transactions from inserting 62. This approach has the benefit of avoiding false conflicts: in contrast, with structural nonexistence, if a set only contains 2 and 2^{20}, then any insertion and lookup of values in the range $\{3 \ldots 2^{20} - 1\}$ conflict.

Finally, the row labeled Table uses a table for representing nonexistence. This is like the Fenceposts table from Section 2: keys in the set hash to locations in the table. On any insertion, we increment the corresponding location in the table. This approach has the benefit of avoiding writes for read-only operations.

Each technique provides value. For a failed get, remove, or modify, we use Table. For a range query, use Struct and Range *within the same range query*. When the first element in the range is not present, we insert an $F_I L_*$ node. When the last element in the range is not present, we insert an $F_* L_I$ node. By using these Range nodes instead of the Struct technique, we avoid the false conflicts that would arise upon a concurrent insertion or removal outside of the range, but immediately preceding (following) the first (last) present node of the range. When two keys K_i and K_j are both present, but the keys between them are not, inserting an $F_E L_E$ node between them does not provide any increased precision over the structural technique. Instead, we record the structural version of the node holding K_i. This representation of missing nodes $\{K_i + 1 \ldots K_j - 1\}$ does not cause false aborts vis-a-vis Range: inserting any of those nodes would invalidate the range query for either technique.

In summary, our strategy avoids using structure to express semantics: when a key is present in the data structure, a transaction will take ownership of the corresponding node, or track the semantic version of that node. False aborts are never due to artificial structural conflicts; they only arise due to hash collisions in the Table. By tuning the size of this table, programmers can strike a balance between false conflicts and space overheads.

4 COMPILER SUPPORT

A practical data structure must support update operations. If the transaction subsequently aborts, the writes performed as part of an update must be undone. We re-purpose compiler support for TM to provide this rollback. The data structure designer writes map::update so that it takes two arguments: a key, and a lambda function to run on the value associated with that key. The designer also annotates the update function to indicate that its lambdas may perform operations that require rollback. At compile time, we locate every lambda that is passed to update. Within each, we (1) replace every store with a call to our library, (2) replace every allocation and deallocation with a call to our library, and (3) make a clone of every function reachable from the lambda, and transform the clone. The store instrumentation logs the old value before updating it. The allocation and deallocation instrumentation ensure that frees do not happen until the transaction commits, and allocations are undone on abort. Lastly, the compiler warns the programmer if there is a call to a function whose body cannot be instrumented (e.g., due to system calls or third-party libraries).

5 CONCLUSIONS AND FUTURE WORK

In this paper, we presented a refined approach to composable concurrent data structures. The key idea behind our work is to make semantic and structural operations explicit, so that structural and semantic operations can commute. Implementation and evaluation is currently in progress. To date, we have implemented scalable lists, deques, stacks, queues, vectors, ordered maps (skip lists), and unordered maps (hash tables). The last of these is perhaps the most exciting: a resize (structural) can commute with arbitrarily many elemental (semantic) operations.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. CNS-1814974. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] Nachshon Cohen, Maurice Herlihy, Erez Petrank, and Elias Wald. 2017. The Teleportation Design Pattern for Hardware Transactional Memory. In *Proceedings of the 21st International Conference on Principles of Distributed Systems*. Lisbon, Portugal.

[2] Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. 2017. Elastic Transactions. *J. Parallel and Distrib. Comput.* 100 (Feb. 2017), 103–127. Issue C.

[3] Maurice Herlihy, Nir Shavit, Victor Luchangco, and Michael Spear. 2021. *The Art of Multiprocessor Programming, 2nd edition.* Morgan Kaufmann.

[4] Maurice P. Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer III. 2003. Software Transactional Memory for Dynamic-sized Data Structures. In *Proceedings of the 22nd ACM Symposium on Principles of Distributed Computing*. Boston, MA.

[5] Maurice P. Herlihy and J. Eliot B. Moss. 1993. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th International Symposium on Computer Architecture*. San Diego, CA.

[6] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: a Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems* 12, 3 (1990), 463–492.

[7] Nir Shavit and Dan Touitou. 1995. Software Transactional Memory. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*. Ottawa, ON, Canada.

[8] Alexander Spiegelman, Guy Golan-Gueta, and Idit Keidar. 2016. Transactional Data Structure Libraries. In *Proceedings of the 37th ACM Conference on Programming Language Design and Implementation*. Santa Barbara, CA.

[9] Pantea Zardoshti, Tingzhe Zhou, Pavithra Balaji, Michael Scott, and Michael Spear. 2019. Simplifying Transactional Memory Support in C++. *ACM Transactions on Architecture and Code Optimization* 16, 3 (July 2019), 25:1–25:24.

[10] Deli Zhang, Pierre Laborde, Lance Lebanoff, and Damian Dechev. 2018. Lock-Free Transactional Transformation for Linked Data Structures. *ACM Transactions on Parallel Computing* 5, 1 (June 2018).

[11] Tingzhe Zhou, Victor Luchangco, and Michael Spear. 2017. Hand-Over-Hand Transactions with Precise Memory Reclamation. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*. Washington, DC.