Transactional Data Structure Libraries: Extended Abstract*

Alexander Spiegelman†
Technion, Haifa, Israel
sashas@tx.technion.ac.il

Guy Golan-Gueta
Yahoo research, Haifa, Israel
ggolan@yahoo-inc.com

Idit Keidar
Technion and Yahoo research, Haifa, Israel
idish@ee.technion.ac.il

Abstract

We introduce *transactions* into libraries of concurrent data structures; such transactions can be used to ensure atomicity of sequences of data structure operations. By restricting transactional access to a well-defined set of data structure operations, we strike a balance between the ease-of-programming of transactions and the efficiency of custom-tailored data structures. We exemplify this concept by designing and implementing a library supporting transactions on any number of maps, sets (implemented as skiplists), and queues. Our library offers efficient and scalable transactions, which are an order of magnitude faster than state-of-the-art transactional memory toolkits. Moreover, our approach treats stand-alone data structure operations (like *put* and *enqueue*) as first class citizens, and allows them to execute with virtually no overhead, at the speed of the original data structure library.

*The full paper has been accepted to PLDI 2016 [25].

†This work was done when Alexander Spiegelman was an intern in Yahoo Labs, Haifa. Alexander Spiegelman is grateful to the Azrieli Foundation for the award of an Azrieli Fellowship.

Motivation

Data structures are the bricks and mortar of computer programs. They are generally provided via highly optimized libraries. Since the advent of the multi-core revolution, many efforts have been dedicated to building *concurrent data structure libraries (CDSLs)* [18, 16, 9, 4, 24, 15], which are so-called "thread-safe". Thread-safety is usually interpreted to mean that each individual data structure operation (e.g., *insert, contains, push, pop*, and so on) executes atomically, in isolation from other operations on the same data structure.

Unfortunately, simply using atomic operations is not always "safe". Many concurrent programs require a number of data structure operations to jointly execute atomically [23]. As an example, consider a server that processes requests to transfer money to bank accounts managed in a CDSL. If several threads process requests in parallel, then clearly, atomicity of individual CDSL operations does not suffice for safety: two concurrent threads processing transfers to the same account may read the same balance at the start of their respective operations, causing one of the transfers to be lost.

This predicament has motivated the concept of memory *transactions* [17], which appear to execute atomically (all-or-nothing) and in isolation (so no partial effects of on-going transactions are observed). A transaction can either *commit*, in which case all of its updates are reflected to the rest of the system, or *abort*, whereby none of its updates take effect. Transactions have been used in DBMSs for decades, and are broadly considered to be a programmer-friendly paradigm for writing concurrent code [22, 14]. Numerous academic works have developed *software transactional memory (STM)* toolkits [7]. Moreover, some (limited) hardware support for transactions is already available [6].

Nevertheless, as of today, general-purpose transactions are not practical. STM incurs too high a overhead [5] and hardware transactions are only "best effort" [6]. And in both cases, abort rates can be an issue. Thus, with the exception of eliding locks [21] in short critical sections, transactions are hardly used in industry today. CDSLs, despite their more limited semantics, are far more popular. Efficient CDSL implementations are available for many programming languages [2, 20, 19, 1] and are widely adopted [23].

Contributions

Our goal in this paper is to provide transaction semantics for CDSLs without sacrificing performance. We introduce the concept of a *transactional CDSL (TDSL)*, which supports bundling sequences of data structure operations into atomic transactions. Individual operations are seen as *singleton transactions* (*singletons* for short). TDSLs provide *composability*; for example, a transaction may invoke operations on two different maps and a queue. But unlike STM approaches, atomicity only encompasses the TDSL's operations, whereas other memory accesses are not protected.

Restricting the transactional alphabet to a well-defined set of operations (e.g., *enqueue, dequeue, insert, remove*, and *contains*) is the key to avoiding the notorious overhead associated with STM. We show that we can benefit from this restriction in three ways:

1. First, while a TDSL implementation *may* use standard STM techniques, it can also apply CDSL-like custom-tailored optimizations that rely on the specific data structure's semantics and organization in order to improve efficiency and reduce the abort rate. For example, it can employ STM-like read-set tracking and validation [7], but reduce the read-set size to include only memory locations that induce real semantic conflicts. Another example is to use transactional access to a core data structure that ensures correctness but does not support fast lookup, and complement it with a non-transactional index for fast lookup.

2. Second, a TDSL can employ different STM strategies for managing different data structures within the same library. For example, transactional access to maps is amenable to optimistic concurrency control, since operations in concurrent transactions are unlikely to conflict. But when queues are used inside transactions, contention is frequent, and so a pessimistic solution is often more efficient.

3. Third, a TDSL can treat singletons as first class citizens – it can spare them the transaction management overhead altogether, and save programmers the need to deal with their aborts.

In our full paper [25] we exemplify these three ideas by presenting example TDSL algorithms for popular data structures – maps, sets (implemented as skiplists), and queues – as well as compositions thereof. In addition, we generalize this concept, and discuss a generic approach for composing TDSLs with each other as well as with STM toolkits such as TL2 [7]. Such a composition can provide, on the one hand, high performance transactions comprised of data structure operations, and on the other hand, fully general transactions, including ones that access scalars.

API and semantics

A TDSL is simply a CDSL with added support for transactions. Its API provides, in addition to CDSL operations (like *insert* and *enqueue*), *TX-begin* and *TX-commit* operations. The added operations are delineations – library operations invoked between a *TX-begin* and the ensuing *TX-commit* pertain to the same transaction[1]. However, other memory accesses made in this span to locations outside the library do not constitute part of the transaction. We assume that the shared data structure's state is only manipulated via the TDSL's API.

The library may *abort* a transaction during any of the operations, resulting in an exception. In case of abort, none of the transaction's operations is reflected in the data structure. Applications using the library need to catch abort exceptions, at which point they typically restart the aborted transaction.

We consider transactions that further provide *opacity* [11], meaning that even transactions that are deemed to abort are not allowed to see inconsistent states of the data structure partially reflecting concurrent transactions' updates.

It is important to note that a TDSL is, in particular, a CDSL, and legacy code may continue to use its operations outside of transactions. Library operations invoked outside transactions are treated as singletons. Singletons cannot abort, and so legacy code can continue to use the original thread-safe library operations. The semantics of singletons relative to other transactions is preserved. In other words, each run has a *linearization* encompassing all of its transactions and singletons.

Support for Library Composition

Here we generalize our concept to allow for composing libraries, that is, supporting transactions that access multiple TDSLs as well as STM toolkits such as TL2 [7]. Our composition framework is based on the theory of Ziv et al. [26]. Such composition can provide support for fully general memory transactions, including ones that access scalars.

API. Generally speaking, a transaction that spans multiple TDSLs begins by calling *TX-begin* in all of them, then accesses objects in the TDSLs via their APIs, and finally attempts to commit in all of them. However, to ensure atomic commitment, we need to split the *TX-commit* operation into three phases – *TX-lock*, *TX-verify*, and *TX-finalize* – and perform each phase for all involved TDSLs before moving to the next phase. Any standard TDSL operation (like *get* or *enqueue*) and any *TX-lock* or *TX-verify* phase of an individual TDSL may throw an abort exception, in which case *TX-abort* is called in all TDSLs partaking in the transaction (for uniformity, we call *TX-abort* also in the library that initiated the abort via an exception). The API that needs to be supported by a composable TDSL is summarized in Table 1.

Intuitively, the first and last phases correspond to the two phases of *strict two-phase locking* [8], where *TX-lock* ensures that the transaction will be able to commit in the *TX-finalize* phase if deemed successful. In case commit-time locking is used, it occurs in *TX-lock*, while with encounter-time locking (as in the case of our queue implementation's dequeue), *TX-lock* simply does nothing. The *TX-verify* phase is required when transactional reads are optimisitic (as in our skiplist TDSL and in TL2).

Semantics. A developer building a TDSL has to ensure that the implementation satisfies certain properties with respect to this API. Defining these formally is beyond the scope of the current paper; instead

[1] In principle, a transaction may end in a programmer-initiated abort, but we omit this option for simplicity of the exposition.

Table 1 API supported by a composable TDSL. In certain implementations, some functions will do nothing.

B	*TX-begin*()	start a transaction
L	*TX-lock*()	make transaction's updates committable
V	*TX-verify*()	verify earlier optimistic operations
F	*TX-finalize*()	commit and end the current transaction
A	*TX-abort*()	abort and end the current transaction

we give here a semi-formal description of the required properties, and refer the reader to a more formal treatment by Ziv et al. [26].

For succinctness, when defining the semantics we refer to calls to *TX-begin*, *TX-lock*, *TX-verify*, *TX-finalize*, and *TX-abort* in TDSL i as B^i, L^i, V^i, F^i, and A^i, respectively. We refer to operations invoked during the transaction via the different TDSLs as $op_1, op_2, \dots op_k$. Using this notation, the *history* of a committed transaction T accessing two TDSLs is the following sequence of steps:

$$B^1, B^2, op_1, op_2, \dots, op_k, L^1, L^2, V^1, V^2, F^1, F^2.$$

A transaction may abort at any point before V successfully returns, for example, during some operation op_j, resulting in a history of the form:

$$B^1, B^2, op_1, op_2, \dots, op_j, A^1, A^2.$$

Note that the above histories involve a single thread; multiple threads can run transactions concurrently, so their history sequences are interleaved. In particular, between any two steps (standard TDSL operations or begin/commit phases) of a given transaction, other transactions (running in other threads) can invoke arbitrary sequences of steps in the same libraries.

We say that a transaction T is *committed* in TDSL i and history h if h includes an F^i step of T. Given a history h, we define the *clean history* of transaction T in TDSL i, denoted $h_c^i(T)$, as the subseqeunce of h consisting of steps involving TDSL i by (1) transactions that are committed in h and i; and (2) T. Henceforth we refer to the requirements a single TDSL needs to satisfy, and so omit the superscript i and simply refer to steps executed in that TDSL only.

A composable TDSL implementation is required to guarantee the following properties with respect to its clean histories:

$C1$: **Atomicity window:** If F has been invoked for transaction T in history h, then for every point p between the completion of L and the invocation of V in $h_c(T)$, T can be seen as if it has been atomically executed in p (i.e., p is a linearization point of T). In the terminology of Ziv et al. [26], this condition requires every committed transaction to have a *serialization window* between the completion of L and the invocation of V.

Note that although the transaction itself does not invoke any operations on the same TDSL during this window, other concurrent transactions may access the TDSL during this time. Intuitively, the L phase "locks" the relevant data objects to avoid interference by concurrent threads, and V verifies that this is indeed a linearization point for the transaction.

$C2$: **Opacity window:** Consider an operation op_m by a transaction T that occurs before the transaction either commits or aborts. Then for every point p in $h_c(T)$ between the invocation of B and the invocation of op_1 (the first TDSL operation invoked by T), the sequence $op_1, \dots, op_m$ can be seen as if it has been atomically executed in p. In other words, this condition requires every active transaction to have a serialization window between B and the invocation of op_1.

This condition guarantees opacity, since $op_1, \dots, op_m$ can be seen as if it is executed immediately after the linearization point of the previous committed transaction; hence the values returned to the client code are based on a consistent shared state (i.e., these values may be returned in a non-concurrent execution of the transaction).

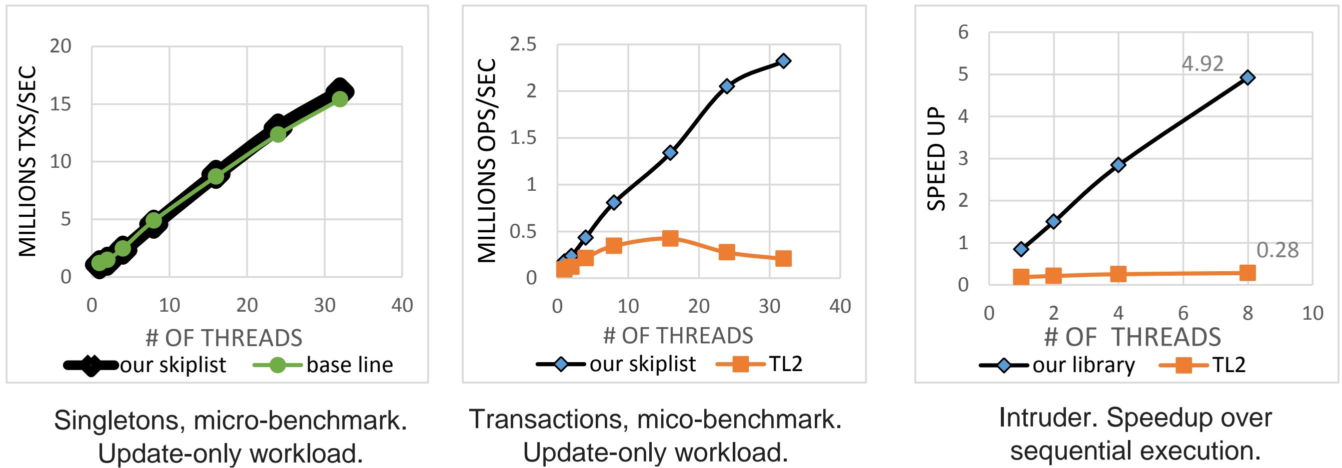

Figure 1: Representative results of our TDSL evaluation.

$C3$: **Aborts:** A transaction can be aborted by invoking A at any time after B is invoked and until F is invoked. It cannot abort after F is invoked; in particular, F should never throw an abort exception. An aborted transaction T leaves the library's state as if T had never been executed.

$C4$: **Non-blocking:** Each operation is non-blocking. This means, for example, that a TDSL operation should never wait for a lock.

Evaluation

We implement our example transactional library and evaluate its performance. We compare the singletons of our transactional skiplist to custom-tailored concurrent skiplist CDSL operations, and evaluate transactions of skiplist operations using our TDSL relative to a state-of-the-art STM. In both cases, we run synthetic workloads using the Synchrobench micro-benchmark suite [10]. In addition, we evaluate our library with Intruder, a standard transactional benchmark that performs signature-based network intrusion detection. It uses transactions that span multiple queues, maps, and skiplists, as well as singletons.

All implementations are in C++. For our library's index, we use a standard textbook concurrent skiplist [18], which is based on [13] and [9], with epoch-based memory reclamation [9]. This is also our *baseline* solution, because our transactional skiplist is a direct extension of this skiplist, and can be similarly implemented atop other baseline skiplists.

For synthetic micro-benchmarking we use the Synchrobench framework [10] configured as follows: Each experiment is a 10 second run in which each thread continuously executes operations or transactions thereof. Keys are selected uniformly at random from the range [1, 1,000,000]. Each experiment is preceded by a warm-up period where 100,000 randomly selected keys are inserted into the skiplist.

We compare our singletons to the baseline in order to assess the overhead inflicted on stand-alone operations by our transaction support. In the full paper [25], we further compare them to all custom-tailored implementations available in Synchrobench, which were shown in [10] to outperform alternative solutions. We compare our transactions to the TL2 implementation provided with Synchrobench, and we compare an Intruder implementation using our library to the STAMP Intruder, which uses the TL2 implementation from [3].

Our evaluation shows that we can get ten-fold faster transactions than STMs in update-dominated workloads accessing sets, and at the same time cater stand-alone operations, (i.e., singletons), in par with state-of-the-art CDSLs. In addition, it shows that Intruder [12] runs up to 17x faster than using a state-of-the-art STM.

The experiments were run on a dedicated machine with four Intel Xeon E5-4650 processors, each with 8 cores, for a total of 32 threads (with hyper-threading disabled). Figure 1 presents a few of our evaluation results. More experiments are reported in the full paper [25].

References

[1] c++ concurrent data structures from libcd. `http://libcds.sourceforge.net/`.

[2] Concurrentskiplistmap from java.util.concurrent. `http://docs.oracle.com/javase/7/docs/api/java/\util/concurrent/ConcurrentSkipListMap.html`.

[3] Tl2 implementation. `https://github.com/daveboutcher/tl2-x86-mp`.

[4] Nathan G Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. In *ACM Sigplan Notices*, volume 45, pages 257–268. ACM, 2010.

[5] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software transactional memory: Why is it only a research toy? *Queue*, 6(5):40:46–40:58, September 2008.

[6] Intel Corporation. Intel transactional synchronization extensions. In *Intel architecture instruction set extensions programming reference*, chapter 8. 2012.

[7] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking ii. In *Distributed Computing*, pages 194–208. Springer, 2006.

[8] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 1976.

[9] Keir Fraser. *Practical lock-freedom.* PhD thesis, University of Cambridge, 2004.

[10] Vincent Gramoli. More than you ever wanted to know about synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorithms. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 1–10, New York, NY, USA, 2015. ACM.

[11] Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 175–184, New York, NY, USA, 2008. ACM.

[12] Bart Haagdorens, Tim Vermeiren, and Marnix Goossens. Improving the performance of signature-based network intrusion detection sensors by multi-threading. In *Proceedings of the 5th International Conference on Information Security Applications*, WISA'04, pages 188–203, Berlin, Heidelberg, 2005. Springer-Verlag.

[13] M Herlihy, Y Lev, and N Shavit. A lock-free concurrent skiplist with wait-free search. *Unpublished Manuscript, Sun Microsystems Laboratories, Burlington, Massachusetts*, 2007.

[14] Maurice Herlihy. The transactional manifesto: Software engineering and non-blocking synchronization. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 280–280, New York, NY, USA, 2005. ACM.

[15] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. A provably correct scalable concurrent skip list. In *Conference On Principles of Distributed Systems (OPODIS).* Citeseer, 2006.

[16] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. A simple optimistic skiplist algorithm. In *Structural Information and Communication Complexity*, pages 124–138. Springer, 2007.

[17] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM.

[18] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming.* Morgan Kaufmann, 2008.

[19] Doug Lea. Overview of package util. concurrent release 1.3. 4. `http://gee.cs.oswego.edu/dl/classes/EDU/oswego/cs/dl/util/concurrent/intro.html`, Retrieved 2011-01-01.

[20] Douglas Lea. *Concurrent programming in Java: design principles and patterns.* Addison-Wesley Professional, 2000.

[21] Ravi Rajwar and James R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 34, pages 294–305, Washington, DC, USA, 2001. IEEE Computer Society.

[22] Michael Scott. Transactional memory today. *SIGACT News*, 46(2):96–104, June 2015.

[23] Ohad Shacham, Nathan Grasso Bronson, Alex Aiken, Mooly Sagiv, Martin T. Vechev, and Eran Yahav. Testing atomicity of composed concurrent operations. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, pages 51–64, 2011.

[24] Nir Shavit and Itay Lotan. Skiplist-based concurrent priority queues. In *Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International*, pages 263–268. IEEE, 2000.

[25] Alexander Spiegelman, Guy Golan-Gueta, and Idit Keidar. Transactional data structure libraries. To appear in PLDI 2016.

[26] Ofri Ziv, Alex Aiken, Guy Golan-Gueta, G. Ramalingam, and Mooly Sagiv. Composing concurrency control. *SIGPLAN Not.*, 50(6):240–249, June 2015.