Brief Announcement: Using Nesting to Push the Limits of Transactional Data Structure Libraries

Gal Assa
Technion – Israel Institute of Technology, Haifa, Israel

Hagar Meir
IBM Research, Haifa, Israel

Guy Golan-Gueta
Independent researcher, Israel

Idit Keidar
Technion – Israel Institute of Technology, Haifa, Israel

Alexander Spiegelman
Independent researcher, CA, USA

Abstract

Transactional data structure libraries (TDSL) combine the ease-of-programming of transactions with the high performance and scalability of custom-tailored concurrent data structures. They can be very efficient thanks to their ability to exploit data structure semantics in order to reduce overhead, aborts, and wasted work compared to general-purpose software transactional memory. However, TDSLs were not previously used for complex use-cases involving long transactions and a variety of data structures.

In this paper, we boost the performance and usability of a TDSL, towards allowing it to support complex applications. A key idea is *nesting*. Nested transactions create checkpoints within a longer transaction, so as to limit the scope of abort, without changing the semantics of the original transaction. We build a Java TDSL with built-in support for nested transactions over a number of data structures. We conduct a case study of a complex network intrusion detection system that invests a significant amount of work to process each packet. Our study shows that our library outperforms publicly available STMs twofold without nesting, and by up to 16x when nesting is used.

2012 ACM Subject Classification Computing methodologies → Concurrent algorithms

Keywords and phrases Transactional Libraries

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.45

Related Version *Full Version*: `https://arxiv.org/abs/2001.00363` [1]

1 Transactional Libraries

The concept of memory transactions [9] is broadly considered to be a programmer-friendly paradigm for writing concurrent code [6, 19]. A transaction spans multiple operations, which appear to execute atomically and in isolation, meaning that either all operations commit and affect the shared state or the transaction aborts. Either way, no partial effects of on-going transactions are observed.

Despite their appealing ease-of-programming, software transactional memory (STM) toolkits [3, 8] are seldom deployed in real systems due to their huge performance overhead. The source of this overhead is twofold. First, an STM needs to monitor all random memory accesses made in the course of a transaction (e.g., via instrumentation in VM-based languages [12]), and second, STMs abort transactions due to conflicts.

Instead, programmers widely use concurrent data structure libraries [20, 14, 5, 2], which are much faster but guarantee atomicity only at the level of a single operation on a single data structure.

To mitigate this tradeoff, Spiegelman et al. [21] have proposed *transactional data structure libraries (TDSL)*. In a nutshell, the idea is to trade generality for performance. A TDSL restricts transactional access to a pre-defined set of data structures rather than arbitrary memory locations, which eliminates the need for instrumentation. Thus, a TDSL can exploit the data structures' semantics and structure to get efficient transactions bundling a sequence of data structure operations. It may further manage aborts on a semantic level, e.g., two concurrent transactions can simultaneously change two different locations in the same list without aborting. While the original TDSL library [21] was written in C++, we implement our version in Java.

Quite a few works [13, 23, 15] have used and extended TDSL and similar approaches like STO [10] and transactional boosting [7]. These efforts have shown good performance for fairly short transactions on a small number of data structures. Yet, despite their improved scalability compared to general purpose STMs, TDSLs have also not been applied to long transactions or complex use-cases.

A key challenge arising in long transactions is the high potential for aborts and the large penalty that such aborts induce as much work is wasted.

2 Our Contribution

Transactional nesting. In this paper we push the limits of the TDSL concept in an attempt to make it more broadly applicable. Our main contribution, is facilitating long transactions via *nesting* [17]. Nesting allows the programmer to define nested *child* transactions as self-contained parts of larger *parent* transactions. This controls the program flow by creating *checkpoints*; upon abort of a nested child transaction, the checkpoint enables retrying only the child's part and not the preceding code of the parent. This reduces wasted work, which, in turn, improves performance. At the same time, nesting does not relax consistency or isolation, and continues to ensure that the entire parent transaction is executed atomically. We focus on *closed nesting* [22], which, in contrast to so-called flat nesting, limits the scope of aborts, and unlike open nesting [18], is generic and does not require semantic constructs.

The flow of nesting is shown in Algorithm 1. When a child commits, its local state is migrated to the parent but is not yet reflected in shared memory. If the child aborts, then the parent transaction is checked for conflicts. And if the parent incurs no conflicts in its part of the code, then only the child transaction retries. Otherwise, the entire transaction does. It is important to note that the semantics provided by the parent transaction are not altered by nesting. Rather, nesting allows programmers to identify parts of the code that are more likely to cause aborts and encapsulate them in child transactions in order to reduce the abort rate of the parent.

Yet nesting induces an overhead which is not always offset by its benefits. We investigate this tradeoff using microbenchmarks. We find that nesting is helpful for highly contended operations that are likely to succeed if retried. We also find that nested variants of TDSL improve performance of state-of-the-art STMs with transaction friendly data structures.

NIDS benchmark. We introduce a new benchmark of a *network intrusion detection system (NIDS)* [4], which invests a fair amount of work to process each packet. This benchmark features a pipelined architecture with long transactions, a variety of data structures, and

Algorithm 1 Transaction flow with nesting.

```
1: TXbegin()
2:     [Parent code]                  ▷ On abort – retry parent
3:     nTXbegin()                     ▷ Begin child transaction
4:         [Child code]               ▷ On abort – retry child or parent
5:     nTXend()                       ▷ On commit – migrate changes to parent
6:     [Parent code]                  ▷ On abort – retry parent
7: TXend()                            ▷ On commit – apply changes to thread state
```

multiple points of contention. It follows one of the designs suggested in [4] and executes significant computational operations within transactions, making it more realistic than existing intrusion-detection benchmarks (e.g., [11, 16]).

Enriching the library. In order to support complex applications like NIDS, and more generally, to increase the usability of TDSLs, we enrich our transactional library in with additional data structures – producer-consumer pool, log, and stack – all of which support nesting. The TDSL framework allows us to custom-tailor to each data structure its own concurrency control mechanism. We mix optimism and pessimism (e.g., stack operations are optimistic as long as a child has popped no more than it pushed, and then they become pessimistic), and also fine tune the granularity of locks (e.g., one lock for the whole stack versus one per slot in the producer-consumer pool).

Evaluation. We evaluate our library using our NIDS application, and compare it against existing general purpose STMs. We find that nesting can improve performance by up to 8x. Moreover, nesting improves scalability, reaching peak performance with as many as 40 threads as opposed to 28 without nesting.

References

1 Gal Assa, Hagar Meir, Guy Golan-Gueta, Idit Keidar, and Alexander Spiegelman. Using nesting to push the limits of transactional data structure libraries. *arXiv preprint*, 2021. `arXiv:2001.00363`.

2 Nathan G Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. In *ACM Sigplan Notices*, volume 45. ACM, 2010.

3 Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *DISC*. Springer, 2006.

4 Bart Haagdorens, Tim Vermeiren, and Marnix Goossens. Improving the performance of signature-based network intrusion detection sensors by multi-threading. In *WISA*. Springer, 2004.

5 Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N Scherer, and Nir Shavit. A lazy concurrent list-based set algorithm. In *OPODIS*. Springer, 2005.

6 Maurice Herlihy. The transactional manifesto: software engineering and non-blocking synchronization. In *PLDI 2005*. ACM, 2005.

7 Maurice Herlihy and Eric Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *PPoPP*, 2008.

8 Maurice Herlihy, Victor Luchangco, Mark Moir, and William N Scherer III. Software transactional memory for dynamic-sized data structures. In *PODC*. ACM, 2003.

9 Maurice Herlihy and J Eliot B Moss. *Transactional memory: Architectural support for lock-free data structures*, volume 21. ACM, 1993.

10 Nathaniel Herman, Jeevana Priya Inala, Yihe Huang, Lillian Tsai, Eddie Kohler, Barbara Liskov, and Liuba Shrira. Type-aware transactions for faster concurrent code. In *Eurosys*, 2016.

11 Guy Korland. Jstamp, 2014. URL: `https://github.com/DeuceSTM/DeuceSTM/tree/master/src/test/jstamp`.

12 Guy Korland, Nir Shavit, and Pascal Felber. Noninvasive concurrency with java STM. In *MULTIPROG*, 2010.

13 Pierre LaBorde, Lance Lebanoff, Christina Peterson, Deli Zhang, and Damian Dechev. Wait-free dynamic transactions for linked data structures. In *PMAM*, 2019.

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

15 Lance Lebanoff, Christina Peterson, and Damian Dechev. Check-wait-pounce: Increasing transactional data structure throughput by delaying transactions. In *IFIP International Conference on Distributed Applications and Interoperable Systems*. Springer, 2019.

16 Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *2008 IEEE International Symposium on Workload Characterization*, 2008.

17 John Eliot Blakeslee Moss. Nested transactions: An approach to reliable distributed computing. Technical report, MIT Cambridge lab, 1981.

18 Yang Ni, Vijay S Menon, Ali-Reza Adl-Tabatabai, Antony L Hosking, Richard L Hudson, J Eliot B Moss, Bratin Saha, and Tatiana Shpeisman. Open nesting in software transactional memory. In *PPoPP*, 2007.

19 Michael Scott. Transactional memory today. *SIGACT News*, 46(2), 2015.

20 Nir Shavit and Itay Lotan. Skiplist-based concurrent priority queues. In *IPDPS 2000*. IEEE, 2000.

21 Alexander Spiegelman, Guy Golan-Gueta, and Idit Keidar. Transactional data structure libraries. In *PLDI 2016*. ACM, 2016.

22 Alexandru Turcu, Binoy Ravindran, and Mohamed M Saad. On closed nesting in distributed transactional memory. In *Seventh ACM SIGPLAN workshop on Transactional Computing*, 2012.

23 Deli Zhang, Pierre Laborde, Lance Lebanoff, and Damian Dechev. Lock-free transactional transformation for linked data structures. *TOPC*, 5(1), 2018.