POSTER: Nesting and Composition in Transactional Data Structure Libraries

Gal Assa
Technion
galassa@campus.technion.ac.il

Hagar Meir
IBM Research
hagar.meir@ibm.com

Guy Golan-Gueta
VMWare Research
ggolangueta@vmware.com

Idit Keidar
Technion
idish@ee.technion.ac.il

Alexander Spiegelman
VMWare Research
spiegelmans@vmware.com

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 work, we boost the performance and usability of a TDSL, 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 nesting in 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 TL2 twofold without nesting, and by up to 16x when nesting is used. Finally, we discuss cross-library nesting, namely dynamic composition of transactions from multiple libraries.

CCS Concepts • **Computing methodologies → Concurrent algorithms**.

This work was partially supported by an Israel Innovation Authority Magnet Consortium (GenPro).

PPoPP '20, February 22–26, 2020, San Diego, CA, USA

ACM ISBN 978-1-4503-6818-6/20/02.
https://doi.org/10.1145/3332466.3374514

1 Introduction

1.1 Transactional Libraries

The concept of memory transactions is broadly considered to be a programmer-friendly paradigm for writing concurrent code. 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] are seldom deployed in real systems due to their huge performance overhead [2]. The source of this overhead is twofold. First, an STM needs to monitor all random memory accesses made in the course of a transaction , and second, STMs abort transactions due to conflicts. Instead, programmers widely use concurrent data structure libraries 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. [9] 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 and allows it to exploit the data structures' semantics and structure to get efficient transactions bundling a sequence of data structure operations. A TDSL can mange aborts on a semantic level, e.g., two concurrent transactions can simultaneously change two different locations in the same list without aborting.

Since its publication, quite a few works have used and extended the TDSL approach [5, 11]. 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 not been applied to long transactions or complex use-cases. A key challenge arising in long transactions is the high potential for aborts along with the large penalty that such aborts induce as much work is wasted.

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
```

1.2 Our Contribution

Transactional nesting. This work pushes the limits of the TDSL concept in an attempt to make it more broadly applicable. Our main contribution is facilitating long transactions via *nesting* [7]. 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, improves performance and reduces energy consumption.

We focus on *closed nesting* [10], which, in contrast to flat nesting, limits the scope of aborts, and unlike open nesting [8], is generic and does not require semantic constructs. Nesting does not relax consistency or isolation, and ensures that the entire parent transaction is executed atomically.

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.

NIDS benchmark. We introduce a new benchmark of a *network intrusion detection system (NIDS)*, which invests a fair amount of work to process each packet. It features a pipelined architecture with long transactions, a variety of data structures, and 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 IDS benchmarks (e.g., [6]).

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 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 per stack versus one per slot in the producer-consumer pool).

Evaluation. We evaluate our NIDS application. 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.

Composition. While most of this work considers nesting in the context of a single library, programmers often wish to access data structures from multiple libraries within the same atomic transaction. We discusses dynamic composition of nested transactions from distinct libraries.
A full version of this work is available via [1].

References

[1] Gal Assa, Hagar Meir, Guy Golan-Gueta, Idit Keidar, and Alexander Speigelman. 2020. Using Nesting to Push the Limits of Transactional Data Structure Libraries. *arXiv preprint arXiv:2001.00363* (2020).

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

[3] Dave Dice, Ori Shalev, and Nir Shavit. 2006. Transactional locking II. In *International Symposium on Distributed Computing*. Springer.

[4] Bart Haagdorens, Tim Vermeiren, and Marnix Goossens. 2004. Improving the performance of signature-based network intrusion detection sensors by multi-threading. In *International Workshop on Information Security Applications*. Springer.

[5] Jaeho Kim, Ajit Mathew, Sanidhya Kashyap, Madhava Krishnan Ramanathan, and Changwoo Min. 2019. MV-RLU: Scaling Read-Log-Update with Multi-Versioning. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM.

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

[7] John Eliot Blakeslee Moss. 1981. *Nested Transactions: An Approach to Reliable Distributed Computing*. Technical Report. Massachusetts inst of tech Cambridge lab for computer science.

[8] Yang Ni, Vijay S Menon, Ali-Reza Adl-Tabatabai, Antony L Hosking, Richard L Hudson, J Eliot B Moss, Bratin Saha, and Tatiana Shpeisman. 2007. Open nesting in software transactional memory. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM.

[9] Alexander Spiegelman, Guy Golan-Gueta, and Idit Keidar. 2016. Transactional Data Structure Libraries. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA.

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

[11] Deli Zhang, Pierre Laborde, Lance Lebanoff, and Damian Dechev. 2018. Lock-free transactional transformation for linked data structures. *ACM Transactions on Parallel Computing (TOPC)* 5, 1 (2018).