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
Novi Research, 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, Nesting

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2021.30

Related Version *Full Version*: `https://arxiv.org/abs/2001.00363` [2]

Supplementary Material *Software (Library)*: `https://github.com/galassatech/TDSL-Nesting-Java/`
Software (Benchmark): `https://github.com/galassatech/NIDS-Java/`

Funding *Gal Assa*: Funded in part by the Hasso Plattner Institute.

1 Introduction

1.1 Transactional Libraries

The concept of memory transactions [25] is broadly considered to be a programmer-friendly paradigm for writing concurrent code [22, 39]. 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 [6, 24, 37] 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 [28]), and second, STMs abort transactions due to conflicts. Instead, programmers widely use concurrent data structure libraries [40, 30, 21, 5], 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. [41] 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 [41] was written in C++, we implement our version in Java. We offer more background on TDSL in Section 2.

Quite a few works [29, 9, 46, 31] have used and extended TDSL and similar approaches like STO [26] and transactional boosting [23]. 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.

1.2 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, presented in Section 3, is facilitating long transactions via *nesting* [33]. 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* [42], which, in contrast to so-called flat nesting, limits the scope of aborts, and unlike open nesting [35], 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.

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 shared state
```

NIDS benchmark. In Section 4 we introduce a new benchmark of a *network intrusion detection system (NIDS)* [19], 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 multiple points of contention. It follows one of the designs suggested in [19] and executes significant computational operations within transactions, making it more realistic than existing intrusion-detection benchmarks (e.g., [27, 32]).

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 Section 3 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. In Section 5, 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.

Summary of contributions. This paper is the first to bring nesting into transactional data structure libraries and also the first to implement closed nesting in sequential STMs. We implement a Java version of TDSL with built-in support for nesting. Via microbenchmarks, we explore when nesting is beneficial and show that in some scenarios, it can greatly reduce abort rates and improve performance. We build a complex network intrusion detection application, while enriching our library with the data structures required to support it. We show that nesting yields significant improvements in performance and abort rates.

2 A Walk down Transactional Data Structure Lane

Our algorithm builds on ideas used in TL2 [6], which is a generic STM framework, and in TDSL [41], which suggests forgoing generality for increased efficiency. We briefly overview their modus operandi as background for our work.

The TL2 [6] algorithm introduced a version-based approach to STM. The algorithm's building blocks are version clocks, read-sets, write-sets, and a per-object lock. A *global version clock (GVC)* is shared among all threads. A transaction has its own *version clock (VC)*, which is the value of GVC when the transaction begins. A shared object has a version, which is the VC of the last transaction that modified it. The read- and write-sets consist of references to objects that were read and written, respectively, in a transaction's execution.

Version clocks are used for *validation*: Upon read, the algorithm first checks if the object is locked and then the VC of the read object is compared to the transaction's VC. If the object is locked or its VC is larger than the transaction's, then we say the validation *fails*, and the transaction aborts. Intuitively, this indicates that there is a *conflict* between the current transaction, which is reading the object, and a concurrent transaction that writes to it.

At the end of a transaction, all the objects in its write-set are locked and then every object in the read-set is revalidated. If this succeeds, the transaction commits and its write-set is reflected to shared memory. If any lock cannot be obtained or any of the objects in the read-set does not pass validation, then the transaction aborts and retries.

Opacity [18] is a safety property that requires every transaction (including aborted ones) to observe only *consistent* states of the system that could have been observed in a sequential execution. TL2's read-time validation (described above) ensures opacity.

In TDSL, the TL2 approach was tailored to specific data structures (skiplists and queues) so as to benefit from their internal organization and semantics. TDSL's skiplists use small read- and write-sets capturing only accesses that induce conflicts at the data strucutre's semantic level. For example, whereas TL2's read-set holds all nodes traversed during the lookup of a particular key, TDSL's read-set keeps only the node holding this key. In addition, whereas TL2 uses only optimistic concurrency-control (with commit-time locking), TDSL's queue uses a semi-pessimistic approach. Since the head of a queue is a point of contention, *deq* immediately locks the shared queue (although the actual removal of the object from the queue is deferred to commit time); the *enq* operation remains optimistic.

Note that TDSL is less general than generic STMs: STM transactions span all memory accesses within a transaction, which is enabled, e.g., by instrumentation of binary code [1] and results in large read- and write-sets. TDSL provides transactional semantics within the confines of the library's data structures while other memory locations are not accessed transactionally. This eliminates the need for instrumenting code.

3 Adding Nesting to TDSL

We introduce nesting into TDSL. Section 3.1 describes the correct behavior of nesting and offers a general scheme for making a transactional *data structure (DS)* nestable. Section 3.2 then demonstrates this technique in the two DSs supported by the original TDSL – queue and skiplist. We restrict our attention to a single level of nesting for clarity, as we could not find any example where deeper nesting is useful. However, deeper nesting could be supported along the same lines if required, via migrating the descendant's local state to its ancestor as described in *nCommit* below. In Section 3.3 we use microbenchmarks to investigate when nesting is useful and when less so, and to compare our library's performance with transactional data structures used on top of general purpose STMs. The nestable log, stack, and producer-consumer pool are described in Section 3.4

3.1 Nesting Semantics and General Scheme

Nesting is a technique for defining child sub-transactions within a transaction. A child has its own local state (read- and write-sets), and it may also observe its parent's local state. A child transaction's commit migrates its local state to its parent but not to shared memory visible by other threads. Thus, the child's operations take effect when the parent commits, and until then remain unobservable.

Correctness. A nested transaction implementation ought to ensure that (1) nested operations are not visible in the shared state until the parent commits; and (2) upon a child's commit, its operations are correctly reflected in the parent's state exactly as if all these operations were executed as part of the parent. In other words, nesting part of a transaction does not change its externally visible behavior.

Implementation scheme. In our approach, the child uses its parent's VC. This way, the child and the parent observe the shared state at the same "logical time" and so read validations ensure that the combined state observed by both of them is consistent, as required for opacity.

Algorithm 2 introduces general primitives for nesting arbitrary DSs. The *nTXbegin* and *nCommit* primitives are exposed by the library and may be called by the user as in Algorithm 1. When user code operates on a transactional DS managed by the library for the first time, it is registered in the transaction's childObjectList, and its local state and lockSet are initialized empty. *nTryLock* may be called from within the library, e.g., a nested dequeue calls *nTryLock*. Finally, *nAbort* may be called by both the user and the library.

We offer the *nTryLock* function to facilitate pessimistic concurrency control (as in TDSL's queues), where a lock is acquired before the object is accessed. This function (1) locks the object if it is not yet locked; and (2) distinguishes newly acquired locks from ones that were acquired by the parent. The latter allows the child to release its locks without releasing ones acquired by its parent.

A nested commit, *nCommit*, validates the child's read-set in all the transaction's DSs *without* locking the write-set. If validation is successful, the child migrates its local state to the parent, again, in all DSs, and also makes its parent the owner of all the locks it holds. To this end, every nestable DS must support *migrate* and *validate* functions, in addition to nested versions of all its methods.

Algorithm 2 Nested begin, lock, commit, and abort.

```
1:  procedure nTXbegin
2:      alloc childObjectList, init empty
3:  On first access to
              obj in child transaction
4:      add obj to childObjectList
5:  procedure nAbort
6:      for each obj in childObjectList do
7:          release locks in lockSet
8:      parent VC ← GVC
9:      for each obj in childObjectList do
10:         validate parent
                                  ▷ DS specific code
11:         if validation fails
12:             abort                ▷ Retry parent
13:     Restart child
14: procedure nCommit
15:     for each obj in childObjectList do
16:         validate obj with parent's VC
17:         if validation fails
18:             nAbort
19:     for each obj in childObjectList do
20:         obj.migrate              ▷ DS specific code
21:     for each lock in lockSet do
22:         transfer lock to parent
23: procedure nTryLock(obj)
24:     if obj is unlocked
25:         lock obj with child id
26:         add obj to lockSet
27:     if obj is locked but not by parent
28:         nAbort                   ▷ Abort child
```

In case the child aborts, it releases all of its locks. Then, we need to decide whether to retry the child or abort the parent too. Simply retrying the child without changing the VC is liable to fail because it would re-check the same condition during validation, namely, comparing read object VCs to the transaction's VC. We therefore update the VC to the current GVC value (line 8) before retrying. This ensures that the child will not re-encounter past conflicts. But in order to preserve opacity, we must verify that the state the parent observed is still consistent at the *new* logical time (in which the child will be retried) because operations within a child transaction ought to be seen as if they were executed as part of

the parent. To this end, we revalidate the parent's read-set against the new VC (line 10). This is done without locking its write-set. Note that if this validation fails then the parent is deemed to abort in any case, and the early abort improves performance. If the revalidation is successful, we restart only the child (line 13).

Recall that retrying the child is only done for performance reasons and it is always safe to abort the parent. Specific implementations may thus choose to limit the number of times a child is retried.

3.2 Queue and Skiplist

We extend TDSL's queue with nested transactional operations in Algorithm 3. The original queue's local state includes a list of nodes to enqueue and a reference to the last node to have been dequeued (together, they replace the read- and write-sets). We refer to these components as the parent's *local queue*, or *parent queue* for short. Nested transactions hold an additional *child queue* in the same format.

Algorithm 3 Nested operations on queues.

```
1:  Queue
2:      sharedQ                          ▷ Shared among all threads
3:      parentQ, childQ                  ▷ Thread local
4:  procedure nEnq(val)
5:      childQ.append(val)
6:  procedure migrate
7:      parentQ.appendAll(childQ)
8:  procedure validate
9:      return true
10: procedure nDeq()
11:     nTryLock()
12:     val ← next node in sharedQ
                                         ▷ stays in sharedQ
13:     if val = ⊥
14:         val ← next node in parentQ
                                         ▷ stays in parentQ
15:         if val = ⊥
16:             val ← childQ.deq()
                                         ▷ Removed from childQ
        return val
```

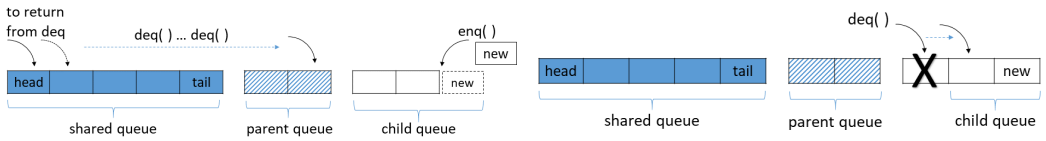


Figure 1 Nested queue operations: dequeue returns objects from the shared, and then parent states without dequeuing them, and when they are exhausted, dequeues from the child's queue; enqueue always enqueues to the child's queue.

The nested enqueue operation remains simple: it appends the new node to the tail of the child queue (line 5). The nested dequeue first locks the shared queue. Then, the next node to return from dequeue is determined in lines 12 – 16, as illustrated in Figure 1. As long as there are nodes in the shared queue that have not been dequeued, dequeue returns the value of the next such node but does not yet remove it from the queue (line 12). Whenever the shared queue has been exploited, we proceed to traverse the parent transaction's local queue (line 14), and upon exploiting it, perform the actual dequeue from the nested transaction's local queue (line 16). A commit appends (migrates) the entire local queue of the child to the tail of the parent's local queue. The queue's validation always returns true: if it never invoked dequeue, its read set is empty, and otherwise, it had locked the queue.

We note that acquiring locks within nested transactions may result in deadlock. Consider the following scenario: Transaction T_1 dequeues from Q_1 and T_2 dequeues from Q_2, and then both of them initiate nested transactions that dequeue from the other queue (T_2 from Q_1 and vice versa). In this scenario, both child transactions will inevitably fail no matter

how many times they are tried. To avoid this, we retry the child transaction only a bounded number of times, and if it exceeds this limit, the parent aborts as well and releases the locks acquired by it. Livelock at the parent level can be addressed using standard mechanisms (backoff, etc.).

To extend TDSL's skiplist with nesting we preserve its optimistic design. A child transaction maintains read- and write-sets of its own, and upon commit, merges them into its parent's sets. As in the queue, read operations of child transactions can read values written by the parent. Validation of the child's read-set verifies that the versions of the read objects have not changed. The skiplist's implementation is straightforward, and its pseudo-code is presented in the full version.

3.3 To Nest, or Not to Nest

Nesting limits the scope of abort and thus reduces the overall abort rate. On the other hand, nesting introduces additional overhead. We now investigate this tradeoff using a synthetic microbenchmark and further provide guidelines for nesting in transactional data structure libraries.

Experiment setup. We run our experiments and measure throughput on an AWS m5.24xlarge instance with 2 sockets with 24 cores each, for a total of 48 physical cores. We disable hyperthreading.

We use a synthetic workload, where every thread runs 50,000 transactions, each consisting of 10 random operations on a shared skiplist followed by 2 random operations on a shared queue. Operations are chosen uniformly at random, and so are the keys for the skiplist operations. We examine three different nesting policies: (1) flat transactions (no nesting); (2) nesting skiplist operations and queue operations; and (3) nesting only queue operations.

We examine two scenarios in terms of contention on the skiplist. In the low contention scenario, the skiplist's key range is from 0 to 50,000. In the second scenario, it is from 0 to 50, so there is high contention. Every experiment is repeated 10 times.

Compared systems. We use the Synchrobench [15] framework in order to compare our TDSL to existing data structures optimized for running within transactions. Specifically, we run ε-STM (Elastic STM [13]) with the three transactional skiplists available as part of Synchrobench – transactional friendly skiplist set, transational friendly optimized skiplist set, and transactional Pugh skiplist set – and to the (single) available transactional queue therein. In all experiments we ran, the friendly optimized skiplist performed better than the other two, and so we present only the results of this data structure. This skiplist requires a dedicated maintenance thread in addition to the worker threads. To provide an upper bound on the performance of ε-STM, we allow it to use the same number of worker threads as TDSL plus an additional maintenance thread, e.g., we compare TDSL with eight threads to ε-STM with a total of nine. We note that ε-STM requires one maintenance thread per skip list; again, to favor ε-STM, we use a single skiplist in the benchmarks.

Synchrobench supports elastic transactions in addition to regular (opaque) ones, and also optionally supports multi-version concurrency control (MVCC) [16, 17], which reduces abort rates on read-only transactions. We experiment with these two modes as well.

We also ran our experiments on TL2 with the transactional friendly skiplist, but it was markedly slower than the alternatives, and in many experiments failed to commit transactions within the internally defined maximum number of attempts. We therefore omit these results.

Results. Figure 2 shows the average throughput obtained.

In the low contention scenario (Figure 2a), nesting both queue and skiplist operations yields the best performance in the vast majority of data points. It improves throughput by 1.6x on average compared to flat transactions on 48 threads. It is worth noting that nesting provides the highest throughput without relaxing opacity like elastic transactions, and without keeping track of multiple versions of memory objects like MVCC. This is due to less work being wasted upon abort. Nesting queue operations seems to be the main reason for the performance gain compared to flat transactions, as nesting only queue operations yields comparable performance. In fact, nesting only the operations on the contended object may be preferable, as it provides the best of both worlds: low abort rates, as discussed later in this section, and less overhead around skiplist sub-transactions. The overhead difference is seen clearly when examining the performance of the two nesting variants of TDSL with a single thread, when nesting induces overhead and offers no benefits.

We further investigate the effect of nesting via the abort rate, shown in Figure 2c (for the low contention scenario). We see the dramatic impact of nesting on the abort rate. This sheds light on the throughput results. Nesting both skiplist and queue operations indeed minimizes the abort rate. However, the gap in abort rate does not directly translate to throughput, as it is offset by the increased overhead.

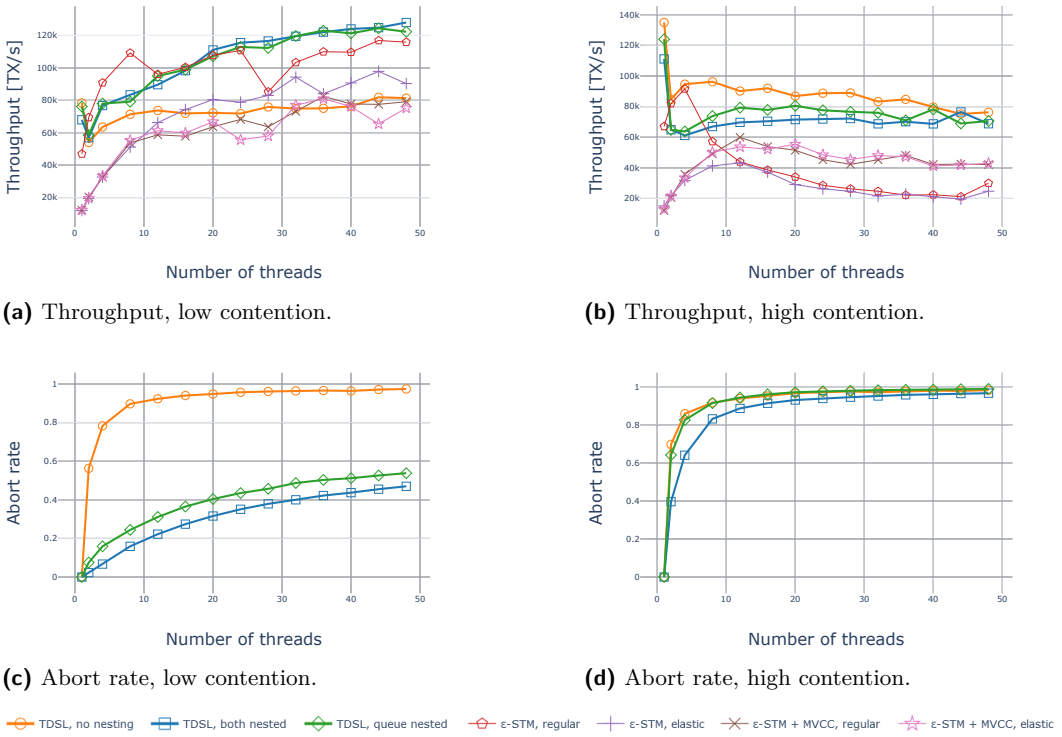
In the high contention scenario (Figure 2b), both DSs are highly contended, and nesting is harmful. The high contention prevents the throughput from scaling with the number of threads, and we observe degradation in performance starting from as little as 2 concurrent threads for TDSL, and between 4-12 concurrent threads for the other variants. From the abort rate point of view (Figure 2d), the majority of transactions abort with as little as 4 threads regardless of nesting, and 80-90% abort with 8 threads. Despite exhibiting the lowest abort rate, nesting all operations performs worse than other TDSL variants. In this scenario, too, nested TDSL performs better than the ε-STM variants despite being unfruitful compared to flat transactions.

Aborts on queue operations occur due to failures of *nTryLock*, which has a good chance of succeeding if retried. On the other hand, aborts on nested skiplist operations are due to reading a higher version than the parent's VC. In such scenarios, the parent is likely to abort as well since multiple threads modify a narrow range of skiplist elements, hence an aborted child is not very likely to commit even if given another chance. Overall, we find that nesting the highly contended queue operations is more useful than nesting map operations – even when contended. Thus, contention alone is not a sufficient predictor for the utility of nesting. Rather, the key is the likelihood of the failed operation to succeed if retried.

3.4 Additional Data Structures

Transactions may span multiple objects of different types. Every data structure implements the methods defined by its type (e.g., dequeue for queue), as well as methods for validation, migrating a child transaction's state to its parent, and committing changes to shared memory. We extend our Java TDSL with three widely used data structures – a producer-consumer pool (supports *produce* and *consume*), a log (*read*, *append*), and a stack (*push*, *pop*). We briefly describe their modus operandi next. We refer the reader to the full version for the specifics of the nesting support transactional implementation and its correctness.

Both the log and the stack resemble with the queue, as both have single points of contention: the log's tail and the stacks head. Accessing elements preceding the log's tail may always succeed and does not generate contention, but appending to it's tail requires acquiring the log's lock upon first append, in a similar manner to the queue's dequeue, as competing



(a) Throughput, low contention.

(b) Throughput, high contention.

(c) Abort rate, low contention.

(d) Abort rate, high contention.

Figure 2 The impact of nesting in TDSL, compared to transactional friendly data structures on ε-STM.

transactions will surely abort if the appending transaction commits. For the stack, reading (i.e., popping) is similar in nature to the dequeue operation, and is performed pessimistically. However, a stack may employ its semantics to create some degree of optimism: Since push and pop operations cancel out with each other, a transaction is not required to operate pessimistically and acquire a lock until it had popped more elements than it had pushed. If at any time during the transaction there was at least as many pushed items as popped ones, the stack's lock will only be acquired at the end of the transaction to append the local stack to its top. In both data structures, child transactions maintain local logs and stacks with elements to be added to the parent's local structures, and eventually to the shared structures upon commit.

The producer-consumer pool is unlike any other data structure implemented in TDSL so far: its sequential specification does not require contained elements be ordered. It has multiple potential points of contention, and has no read-only operations. Transactions (both parent and nested) mark slots in the shared pool as that are accessed withing the transaction, so other threads do not access them. The transactional implementation includes a cancellation mechanism that releases nodes that were produced and then consumed in the same transaction, as well a migration mechanism to apply operations from a nested transaction to the parent's state.

4 NIDS Case Study

We conduct a case study of parallelizing a full-fledged network intrusion detection system using memory transactions. In this section we provide essential background for multi-threaded IDS systems, describe our NIDS software and point out candidates for nesting.

Intrusion detection is a basic security feature in modern networks, implemented by popular systems such as Snort [38], Suricata [14], and Zeek [36]. As network speeds increase and bandwidth grows, NIDS performance becomes paramount, and multi-threading becomes instrumental [19].

Multi-threaded NIDS. We develop a multi-threaded NIDS benchmark. The processing steps executed by the benchmark follow the description in [19]. As illustrated in Figure 3, our design employs two types of threads. First, *producers* simulate the *packet capture* process of reading packet fragments off a network interface. In our benchmark, we do not use an actual network, and so the producers generate the packets and push packet fragments into a shared producer-consumer pool called the *fragments pool*. The rationale for using dedicated threads for packet capture is that – in a real system – the amount of work these threads have scales with network resources rather than compute and DRAM resources. In our implementation, the producers simply drive the benchmark and do not do any actual work.

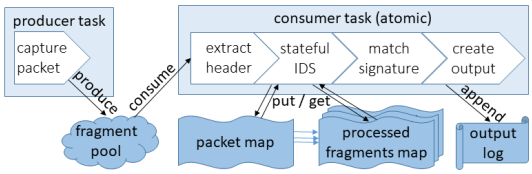


Figure 3 Our NIDS benchmark: tasks and data structures.

Packet processing is done exclusively by the *consumer* threads, each of which consumes and processes a single packet fragment from the shared pool. Algorithm 4 describes the consumer's code. To ensure consistency, each consumer executes as a single atomic transaction. It begins by performing *header extraction*, namely, extracting information from the link layer header. The next step is called *stateful IDS*; it consists of packet reassembly and detecting violations of protocol rules. Reassembly uses a shared *packet map* associating each packet with its own shared *processed fragment map*. The first thread to process a fragment pertaining to a particular packet creates the packet's fragment map whereas other threads append fragments to it. Similarly, only the thread that processes a packet's last fragment continues to process the packet, while the remaining threads move on to process other fragments from the pool. By using atomic transactions, we guarantee that indeed there are unique "first" and "last" threads and so consistency is preserved.

Algorithm 4 Consumer code.

```
1:  f ← fragmentPool.consume()
2:  process headers of f
3:  fragmentMap ← packetMap.get(f)
                                  ▷ Start nested TX
4:  if fragmentMap = ⊥
5:      fragmentMap ← new map
6:      packetMap.put(f, fragmentMap)
                                  ▷ End nested TX
7:  fragmentMap.put(f.id, f)
8:  if f is the last fragment in packet
9:      reassemble and inspect packet
                                  ▷ Long computation
10:     log the result            ▷ Nested TX
```

The thread that puts together the packet proceeds to the *signature matching* phase, whence the reassembled packet's content is tested against a set of logical predicates; if all are satisfied, the signature matches. This is the most computationally expensive stage [19]. Finally, the thread generates a packet trace and writes it to a shared log.

As an aside, we note that our benchmark performs five of the six processing steps detailed in [19]; the only step we skip is *content normalization*, which unifies the representations of packets that use different application-layer protocols. This phase is redundant in our solution since we use a unified packet representation to begin with. In contrast, the *intruder* benchmark in STAMP [32] implements a more limited functionality, consisting of packet reassembly and naïve signature matching: threads obtain fragments from their local states (rather than a shared pool), signature matching is lightweight, and no packet traces are logged. This results in significantly shorter transactions than in our solution.

Nesting. We identify two candidates for nesting. The first is the logging operation given that logs are prone to be highly contended. Because in this application the logs are write-only, transactions abort only when they contend to write at the tail and not because of consistency issues. Therefore, retrying the nested transaction amounts to retrying to acquire a lock on the tail, which is much more efficient than restarting the transaction.

Second, when a packet consists of multiple fragments, its entry in the packet map is contended. In particular, for every fragment, a transaction checks whether an entry for its packet exists in the map, and creates it if it is absent. Nesting lines 3 - 6 of Algorithm 4 may thus prevent aborts.

5 NIDS Evaluation

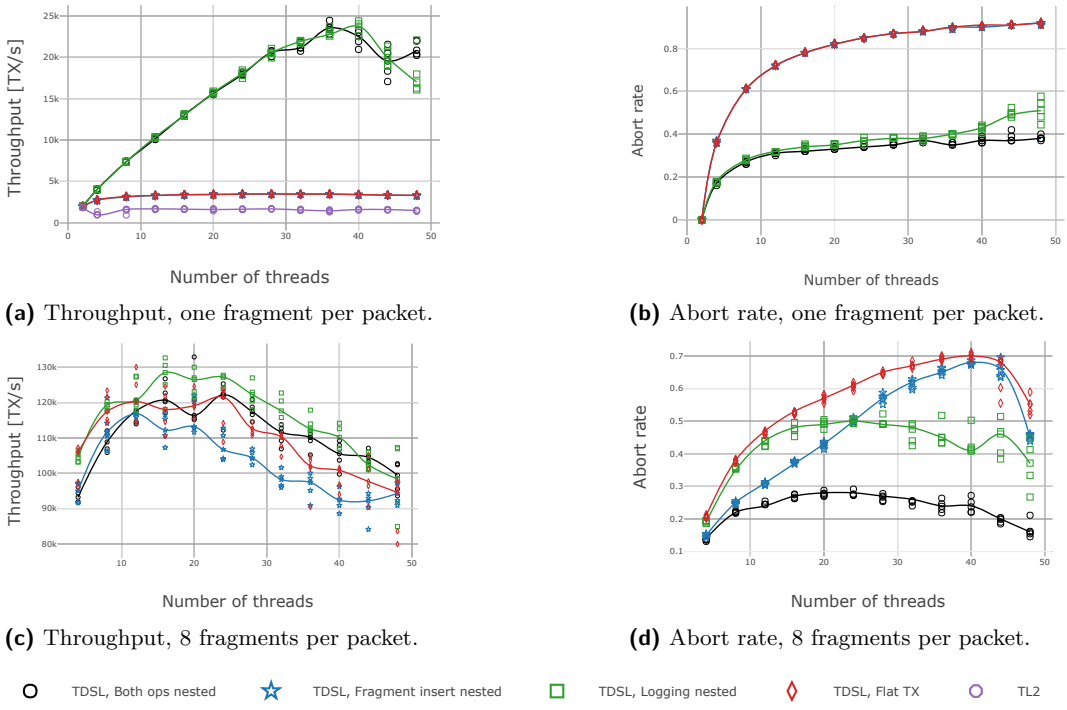
We now experiment with nesting in the NIDS benchmark. We detail our evaluation methodology in Section 5.1 and present quantitative results in Section 5.2.

5.1 Experiment Setup

Our baseline is TDSL without nesting, which is the starting point of this research. We also compare to the open source Java STM implementation of TL2 by Korland et al. [28], as well as ε-STM [13] and PSTM [16, 17]. The results we obtained for ε-STM and PSTM were very similar to those of TL2 and are omitted from the figures to avoid clutter. Note that the open-source implementations of ε-STM and PSTM optimize only data structures that contain integers; they use bare-STM implementations for data structures holding general objects, as the data structures in our benchmark do. This explains why their performance is sub-optimal in this benchmark.

We experiment with nesting each of the candidates identified in Section 4 (put-if-absent to the packetMap and updating the log), and also with nesting both. Our baseline executes *flat transactions*, i.e., with no nesting. In TDSL, the packet pool is a producer-consumer pool, the map of processed packets is a skiplist of skiplists, and the output block is a set of logs. For TL2, the packet pool is implemented with a fixed-size queue, the packet map is an RB-tree of RB-trees, and the output log is a set of vectors. We use the implementations provided in [27] without modification.

The experiment environment is the same as for the microbenchmark described in Section 3.3. We repeated the experiment on an in-house 32-core Xeon machine and observed similar trends; these results are omitted. We run each experiment 5 times and plot all data points, connecting the median values with a curve.



(a) Throughput, one fragment per packet.

(b) Abort rate, one fragment per packet.

(c) Throughput, 8 fragments per packet.

(d) Abort rate, 8 fragments per packet.

Figure 4 NIDS experiments results.

We conduct two experiments. In the first, each packet consists of a single fragment, there is one producer thread, and we scale the number of consumers. In the second experiment, there are 8 fragments per packet and as we scale the number of threads, we designate half the threads as producers. We experimented also with different ratios of producers to consumers, but this did not seem to have a significant effect on performance or abort rates, so we stick to one configuration in each experiment. The number of fragments per packet governs contention: If there are fewer fragments then more threads try to write to logs simultaneously. More fragments, on the other hand, induce more put-if-absent attempts to create maps.

5.2 Results

Performance. Figures 4a and 4b show the throughput and abort rate in a run with 1 fragment per packet and a single producer. Whereas the performance of all solutions is similar when we run a single consumer, performance differences become apparent as the number of threads increases. For flat transactions (red diamonds), TDSL's throughput is consistently double that of TL2 (purple octagons), as can be observed in Figure 5, which zooms in on these two curves in the same experiment. We note that the TDSL work [41] reported better performance improvements over TL2, but they ran shorter transactions that did not write to a contended log at the end, where TDSL's abort rate remained low. In contrast, our benchmark's long transactions result in high abort rates in the absence of nesting. Nesting the log writes (green squares) improves throughput by an additional factor of up to 6, which is in line with the improvement of TDSL over TL2 reported in [41], and also reduces the abort rate by a factor of 2. The packet map is not contended in this experiment, and so transactions with nested insertion to the map behave similarly to flat ones (in terms of both throughput and abort rate).

Table 1 Scalability: peak performance (tx/sec) / number of threads where it is achieved.

Algorithm	1 Fragment	8 Fragments
TL2	1.6K / 8	24K / 4
TDSL flat	3.5K / 28	122K / 24
TDSL nesting log	23.5K / 40	127K / 24
TDSL nesting put-if-absent	3.5K / 28	113K / 20
TDSL nesting both	23.5K / 36	122K / 24

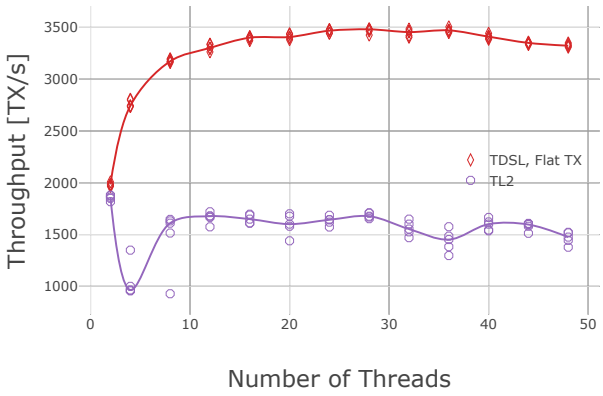


Figure 5 Throughput of TL2 and flat transactions in TDSL, a single producer and one fragment per packet.

Figure 4c shows the results in experiments with 8 fragments per packet. For clarity, we omit TL2 from this graph because it performs 6 times worse than the lowest alternative. Here, too, the best approach is to nest only log updates, but the impact of such nesting is less significant in this scenario, improving throughput only by about 20%. This is because with one fragment per packet, every transaction tries to write to the log, whereas with 8, only the last fragment's transaction does, reducing contention on the log. Nevertheless, the effect of nesting log updates is more significant as it reduces the number of aborts by a factor of 3, and thus saves work.

Unlike in the 1-thread scenario, with 8 threads, there is contention on the put-if-absent to the fragment map, and so nesting this operation reduces aborts. At first, it might be surprising that flat transactions perform better than ones that nest the put-if-absent despite their higher abort rate. However, the abort reduction has a fairly low impact since this operation is performed early in the transaction. Thus, the overhead induced by nesting exceeds the benefit of not repeating the earlier part of the computation. The effect of this overhead is demonstrated in the difference in performance between nesting both candidates (black circles) and nesting only the log writes (green squares).

Scaling. Not only does nesting have a positive effect on performance, it improves scalability as well. For instance, Figure 4a shows that throughput increases linearly all the way up to 40 threads when nesting the logging operation, whereas flat nesting, as can be seen in Figure 5, peaks at 28 threads but saturates already at 16. Table 1 summarizes the scaling factor in both experiments.

6 Related Work

Transactional data structures. Since the introduction of TDSL [41] and STO [26], transactional libraries got a fair bit of attention [29, 9, 46, 31]. Other works have focused on wait-free [29] and lock-free [9, 46] implementations (as opposed to TDSL and STO's lock-based approach). Such algorithms are interesting from a theoretical point of view, but provide very little performance benefits, and in some cases can even yield worse results than lock-based solutions [10, 7]. Lebanoff et al. [31] introduce a trade-off between low abort rate and high computational overhead. By restricting their attention to static transactions, they are able to perform scheduling analyses in order to reduce the overall system abort rate. We, in contrast, support dynamic transactions.

Transactional boosting and its follow-ups [20, 23] offer generic approaches for making concurrent data structures transactional. However, they do not exploit the structure of the transformed data structure, and instead rely on semantic abstractions like compensating actions and abstract locks.

Some full-fledged STMs incorporate optimization for specific data structures. For instance, ε-STM [13] and PSTM [16, 17] support elastic transactions on search data structures. Note, however, that unlike closed nesting, elastic transactions relax transactional semantics. PSTM allows programmers to select the concurrency control mechanism (MVCC or single-version) and the required semantics (elastic or regular) for each transaction. While this offers a potential for performance gains, our results in Section 3.3 have shown that nesting outperforms all of the approaches.

PSTM improves on SwissTM [8], which has featured other optimizations in order to support longer transactions, like a contention manager and mixed concurrency control, and showed 2-3x better performance compared to TL2 [6] and TinySTM [12] and good scalability up to 8 threads. These optimizations are orthogonal to nesting.

Chopping and nesting. Recent works introduced the concept of *chopping* [43, 11, 45], which splits up transactions in order to reduce abort rates. Chopping and the similar concept of elastic transactions [13] were recently adopted in transactional memory [34, 44, 31]. The high-level idea of chopping is to divide a transaction into a sequence of smaller ones and commit them one at a time. While atomicity is eventually satisfied (provided that all transactions eventually commit), this approach forgoes isolation, which nesting preserves.

While some previous work on supporting nesting in generic STMs was done in the past [4, 42, 35, 3], we are not aware of any previous work implementing *closed nesting* in a non-distributed sequential STM. This might be due to the fact that the benefit of closed nesting is in allowing longer transactions whereas STMs are not particularly suitable for long transactions in any case, and the extra overhead associated with nesting might be excessive when read- and write-sets are large as in general purpose STMs. Our solution is also the first to introduce nesting into transactional data structure libraries, and thus the first to exploit the specific structure and semantics of data structures for efficient nesting. Because our data-structures use diverse concurrency control approaches, we had to develop nesting support for each of them. An STM using any of these approaches (e.g., fine-grain commit-time locking with read-/write-sets) can mimic our relevant technique (e.g., closed-nesting can be supported in TL2 using a similar scheme to the one we use in maps).

7 Conclusion

The TDSL approach enables high-performance software transactions by restricting transactional access to a well-defined set of data structure operations. Yet in order to be usable in practice, a TDSL needs to be able to sustain long transactions, and to offer a variety of data structures. In this work, we took a step towards boosting the performance and usability of TDSLs, allowing them to support complex applications. A key enabler for long transactions is nesting, which limits the scope of aborts without changing the transactional semantics.

We have implemented a Java TDSL with built-in support for nesting in a number of data structures. We conducted a case study of a complex network intrusion detection system running long transactions. We found that nesting improves performance by up to 8x, and the nested TDSL approach outperforms the general-purpose STM by up to 16x. We plan to make our code (both the library and the benchmark) available in open-source.

References

1 David J Angel, James R Kumorek, Farokh Morshed, and David A Seidel. Byte code instrumentation, November 6 2001. US Patent 6,314,558.

2 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 arXiv:2001.00363*, 2021.

3 Joao Barreto, Aleksandar Dragojević, Paulo Ferreira, Rachid Guerraoui, and Michal Kapalka. Leveraging parallel nesting in transactional memory. In *ACM Sigplan Notices*, volume 45 (5), 2010.

4 Martin Bättig and Thomas R Gross. Encapsulated open nesting for STM: fine-grained higher-level conflict detection. In *PPoPP*, 2019.

5 Nathan G Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. In *ACM Sigplan Notices*, volume 45 (5). ACM, 2010.

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

7 Dave Dice and Nir Shavit. Understanding tradeoffs in software transactional memory. In *CGO*. IEEE, 2007.

8 Aleksandar Dragojević, Rachid Guerraoui, and Michal Kapalka. Stretching transactional memory. *ACM sigplan notices*, 44, 2009.

9 Avner Elizarov and Erez Petrank. Loft: lock-free transactional data structures. Master's thesis, Computer Science Department, Technion, 2019.

10 Robert Ennals. Software transactional memory should not be obstruction-free. Technical report, IRC-TR-06-052, Intel Research Cambridge, 2006.

11 Jose M Faleiro, Daniel J Abadi, and Joseph M Hellerstein. High performance transactions via early write visibility. *VLDB*, 10(5), 2017.

12 Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *PPoPP*, 2008.

13 Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. Elastic transactions. In Idit Keidar, editor, *Distributed Computing*, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

14 OIS Foundation. Suricata. URL: `https://suricata-ids.org/`.

15 Vincent Gramoli. More than you ever wanted to know about synchronization: synchrobench, measuring the impact of the synchronization on concurrent algorithms. In *PPoPP*, 2015.

16 Vincent Gramoli and Rachid Guerraoui. Democratizing transactional programming. *Commun. ACM*, 57(1), January 2014.

17 Vincent Gramoli and Rachid Guerraoui. Reusable concurrent data types. In *ECOOP 2014 – Object-Oriented Programming*. Springer, 2014.

18 Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In *PPoPP*. ACM, 2008.

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

20 Ahmed Hassan, Roberto Palmieri, and Binoy Ravindran. Optimistic transactional boosting. In *PPoPP*, 2014.

21 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.

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

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

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

25 Maurice Herlihy and J Eliot B Moss. *Transactional memory: Architectural support for lock-free data structures*, volume 21 (2). ACM, 1993.

26 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.

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

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

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

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

31 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.

32 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.

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

34 Shuai Mu, Sebastian Angel, and Dennis Shasha. Deferred runtime pipelining for contentious multicore software transactions. In *EuroSys*. ACM, 2019.

35 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.

36 Vern Paxson. Bro: a System for Detecting Network Intruders in Real-Time. *Computer Networks*, 31(23-24):2435–2463, 1999. URL: `http://www.icir.org/vern/papers/bro-CN99.pdf`.

37 Dmitri Perelman, Anton Byshevsky, Oleg Litmanovich, and Idit Keidar. Smv: selective multi-versioning stm. In *DISC*. Springer, 2011.

38 Martin Roesch et al. Snort: Lightweight intrusion detection for networks. In *Lisa*, volume 99, 1999.

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

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

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

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

43 Zhaoguo Wang, Shuai Mu, Yang Cui, Han Yi, Haibo Chen, and Jinyang Li. Scaling multicore databases via constrained parallel execution. In *SIGMOD*, 2016.

44 Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast in-memory transaction processing using RDMA and HTM. In *SOSP*, 2015.

45 Chao Xie, Chunzhi Su, Cody Littley, Lorenzo Alvisi, Manos Kapritsos, and Yang Wang. High-performance ACID via modular concurrency control. In *SOSP*, 2015.

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