Transactional Data Structure Libraries

Alexander Spiegelman *
Technion, Israel
sasha.spiegelman@gmail.com

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

Idit Keidar
Technion and Yahoo Research, 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 focusing on 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.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming

General Terms Algorithms, Performance

Keywords Concurrency, data structures, semantics, transactions.

1. Introduction

1.1 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)* [1, 2, 5, 9, 11, 13, 21, 25, 27, 28, 36, 37, 44, 48, 49], 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, as shown in [47]. 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 *get* and *insert* operations does not suffice for safety: two concurrent threads processing transfers to the same account may obtain 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* [24] spanning multiple operations, 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, 46]. Numerous academic works have developed *software transactional memory (STM)* toolkits [10, 26, 43]. Moreover, some (limited) hardware support for transactions is already available [33].

Nevertheless, as of today, general-purpose transactions are not practical. STM incurs too high a overhead [6] and hardware transactions are only "best effort" [33]. And in both cases, abort rates can be an issue. Thus, with the exception of eliding locks [45] 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 [1, 2, 36, 37] and are widely adopted [47].

* This work was done when Alexander Spiegelman was an intern with Yahoo. Alexander Spiegelman is grateful to the Azrieli Foundation for the award of an Azrieli Fellowship.

PLDI'16, June 13–17, 2016, Santa Barbara, CA, USA

http://dx.doi.org/10.1145/2908080.2908112

1.2 Contributions

Our goal in this paper is to provide transaction semantics for CDSLs without sacrificing performance. We introduce in Section 2 the concept of a *transactional data structure library (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 [10], 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. A TDSL can combine the two, by using optimistic concurrency control for its maps and a pessimistic approach for its queues.
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.

We exemplify these three ideas in Section 3, where we present example TDSL algorithms for popular data structures – maps and sets (implemented as skiplists), and queues – as well as compositions thereof. In Section 4 we generalize this concept, and discuss a generic approach for composing TDSLs with each other as well as with STM toolkits such as TL2 [10]. 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.

We implement our new algorithms in C++. Our evaluation in Section 5 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), on par with state-of-the-art CDSLs. We further use our library to support Intruder [19] – a multi-threaded algorithm for signature-based network intrusion detection – and it runs up to 17x faster than using a state-of-the-art STM.

Finally, we note that our example TDSL is by no means exhaustive. Our goal here is to put forth TDSL as a new concept for concurrent programming, which can offer programmers the ease-of-use of transactions at the speed of CDSLs. Section 6 compares this paradigm with earlier ideas in the literature. We conclude in Section 7 by expressing hope that the community will adopt this new concept and build additional TDSLs.

2. Bringing Transactions into CDSLs

In this section we consider a single TDSL and explain how it can be used by programmers. Composition of multiple TDSLs is discussed in Section 4 below.

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 *TX-begin* and *TX-commit* calls are global to the library, i.e., span all objects managed by the library, since a thread may have at most one ongoing transaction at a time.

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* [17], 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.

Finally, a TDSL is, in particular, a CDSL, and legacy code may continue to use its operations outside of transactions, in which case they 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,

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

each run has a *linearization* [30] encompassing all of its transactions and singletons.

Use case. We illustrate the usage of a TDSL via a simplified example from the Intruder benchmark [19]. One of the tasks performed by the benchmark is *reassembly*, where many threads process messages from the network concurrently. Each network flow consists of multiple messages, each carrying the flow's key and some value associated with it. The values for each key are tracked in a map until all of them arrive, at which point the flow is removed from the map and inserted to a queue for further processing. Example 1 shows a simplified pseudo-code for one thread executing this task, using a shared queue Q and a shared map M.

Example 1 Intruder reassembly task

```
 1: procedure handle_message(m)
 2:     TX-begin()
 3:     cur ← M.get(m.key)
 4:     if cur = ⊥
 5:         cur ← new flow
 6:         M.insert(m.key, cur)
 7:     append m.val to cur
 8:     compute something on cur
 9:     if cur is complete
10:         Q.enqueue(cur)
11:         M.remove(m.key)
12:     TX-commit()

13: upon abort_exception
14:     restart handle_message(m)
```

It is easy to see that a standard "thread-safe" DSL does not suffice here: If tasks execute concurrently, a race can cause two threads to create and *insert* new flows for the same key, resulting in loss of some of the values. Similarly, two threads may *enqueue* (line 10) the same flow if both detect that it is complete.

3. Transactional Data Structure Algorithms

In this section we present algorithms for a TDSL providing skiplists, (which we use for maps and sets), and queues. In Section 3.1 we describe a skiplist supporting efficient transactions. The implementation of maps using the skiplist is immediate, and so we do not elaborate on it. In Section 3.2 we explain how we compose multiple objects within a single transaction. Section 3.3 presents our queue implementation. All three sections discuss full-fledged transactions. In Section 3.4 we describe our support for fast singletons.

3.1 Transactional Skiplist

Our skiplist supports the standard *insert(key)*, *contains(key)*, and *remove(key)* operations. Our library also supports iterators via *getFirst()*, *getNext()*, and *getValue()* operations. We start with an STM-like technique and optimize it by taking advantage of the skiplist's specific semantics and structure. Our goals are to reduce the overhead and to avoid unnecessary aborts.

For didactic reasons we describe the algorithm in three steps. First, in Section 3.1.1, we take a simple linked list and add to it standard STM-like conflict detection using the TL2 algorithm [10]. Next, in Section 3.1.2, we explain how to limit both overhead and aborts by reducing the validation set to a minimum. Finally, in Section 3.1.3, we explain how we make operations faster and reduce aborts even further using a skiplist.

The complete pseudo-code of the transactional skiplist appears in Algorithms 2 and 3. For clarity, the pseudo-code is presented for a sequentially consistent memory model, i.e., without explicit memory fences. In order for it to run correctly on relaxed memory models, which do not guarantee sequential consistency, a number of explicit fences are required, as in implementations of the TL2 algorithm [18, 20].

3.1.1 STM-Like Validation

We begin with a simple sequential linked list and extend it according to the TL2 STM algorithm [10] in order to support transactions of multiple operations. At this point, we consider only a linked list, which does not allow fast access to searched keys. In Section 3.1.3 below, we will extend the linked list with an external *index* that will shorten the access time. In addition, we only consider for now operations invoked inside transactions, and defer the discussion of singletons to Section 3.4.

The algorithm uses a *global version clock (GVC)* that exposes two operations: a *read*, which returns the current version, and an *add-and-fetch* that atomically increases the GVC's version and returns its new value. The versions are used to detect conflicts among concurrent transactions: Each node in the linked list is tagged with a version, which is updated by transactions that create or update the node. A transaction *validates* its reads by checking that their versions have not been increased (by other transactions) since the transaction had begun. Transactions re-validate all their reads at commit time. In case of conflicts (manifested as newer versions), the transaction aborts. In addition to versions, nodes are associated with locks in order to allow a transaction to change all the nodes it affects atomically.

We now explain the operations in more detail. When a transaction is initiated via *TX-Begin*, it reads and stores the GVC's current value in a local variable *readVersion*. Each thread in the midst of an active a transaction also maintains two local sets – *read-set* and *write-set*.

To start an operation on some key k within a transaction, the executing thread *traverses* the linked list until it reaches a node with key $\geq k$. We refer to this node as *successor(k)* and to its predecessor *predecessor(k)*. Note that, in particular, *successor(k)* may be the node holding key k. For each node n encountered during this traverse, the thread checks whether n is locked or its version is bigger than *readVersion*, in which case, it aborts the transaction. More specifi-

cally, the thread invokes *getValidatedValue*(*n, readVersion*), which first searches the node in *write-set* and returns its new value if it is there; otherwise, it checks the node's lock, then reads its value (e.g., the next pointer required to continue the traversal), and then checks its version and checks the lock again. If all checks succeed, n is added to *read-set*, and otherwise the transaction is aborted.

The list is not updated before *TX-commit* is invoked. Rather, every write into a list node during an operation is saved in *write-set*. In case of *insert(k)*, *predecessor(k)* and the new node are added to *write-set*, whereas *remove(k)* adds both *predecessor(k)* and *successor(k)* to *write-set*; the latter is the node being removed. If the enclosing transaction successfully commits, *TX-commit* updates the list and the affected node versions accordingly. The thread-local memory at the end of the traversal is illustrated in Figure 1.

An *iterator* is also implemented by traversing the list. All calls to *getFirst*, *getNext*, and *getValue* validate the current node using *getValidatedValue*, and add it to *read-set*.

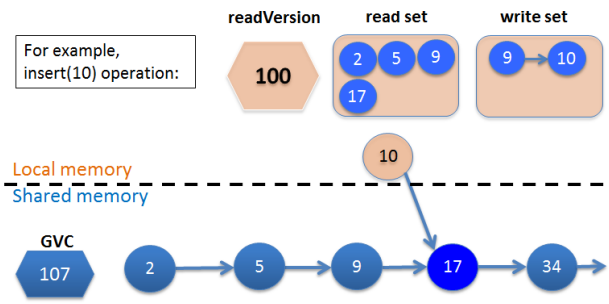

Figure 1: Example of *insert(10)* operation. The executing thread traverses the list from the beginning (node with $k = 2$) until the successor of 10 (key $= 17$). During the traverse it adds nodes with keys 2, 5, and 9 to *read-set*. At the end, it adds the node with key 9 to *write-set* with an indication that its *next* pointer should be set to the node with key 10, and also adds the new node ($key = 10$).

During *TX-commit*, a thread first tries to lock the nodes in the *write-set*, and then to validate the *read-set* (using *getValidatedValue*). If either fails, the transaction aborts. Otherwise, the thread calls the GVC's *add-and-fetch* method, stores the result in a local variable *writeVersion*, and updates the linked list according to the *write-set*. In addition, it updates the versions of all nodes in the *write-set* to be *writeVersion*. Finally it releases the locks and returns successfully.

For example, consider a transaction that consists solely of the *insert(10)* operation illustrated in Figure 1. In this case, *TX-commit* will try to lock the node with key 9, and then validate that the versions of nodes with keys 2, 5, and 9 are still no larger than *readVersion*. If successful, it will *add-and-fetch* the GVC into *writeVersion*, set the version of the node with key 9 to be *writeVersion* and its *next* pointer to be the new node with key 10. It will also set the version of the new node to *writeVersion*.

Algorithm 2 Transactional skiplist: operations

```
Type Node = ⟨key, next, version, lock, deleted⟩
local variables:
    readVersion, writeVersion ∈ ℕ
    read-set: set of nodes
    write-set: set of ⟨node, field, value⟩
    index-todo: set of ⟨node, insert/remove⟩

TX-begin()
    readVersion ← GVC.read()
    read-set, write-set, index-todo ← {}

contains(k)
    ⟨pred, succ⟩ ← traverseTo(k)
    if succ ≠ ⊥ ∧ succ.key = k   return true
    else   return false

insert(k)
    ⟨pred, succ⟩ ← traverseTo(k)
    if succ ≠ ⊥ ∧ succ.key = k   return false
    allocate newNode
    newNode.key = k; newNode.next = succ
    add ⟨pred, next, newNode⟩ to write-set
    add ⟨newNode, ⊥, ⊥⟩ to write-set
    add ⟨newNode, insert⟩ to index-todo

remove(k)
    ⟨pred, succ⟩ ← traverseTo(k)
    if succ = ⊥ ∨ succ.key ≠ k
        return false
    add succ to read-set
    add ⟨pred, next, succ.next⟩ to write-set
    add ⟨succ, deleted, true⟩ to write-set
    add ⟨succ, remove⟩ to index-todo

TX-commit()
    if ¬try-lock(write-set)                          ▷ lock
        release all acquired locks and abort
    if ¬validate(read-set)                           ▷ validate
        release all acquired locks and abort
    writeVersion ← GVC.add-and-fetch()
    do-update(write-set)                             ▷ update the list
    release all acquired locks                       ▷ successful commit
    update-index(index-todo)                         ▷ lazy update
```

3.1.2 Read-Set Reduction

The goal of our next step is to avoid unnecessary aborts and to reduce the overhead for maintaining and validating the *read-set*. To this end, we take advantage of the list's semantics and structure, as custom-tailored CDSLs do.

We observe that operations of concurrent transactions that change the list in different places need not cause an abort, since their operations commute. Figure 2 shows an example where an operation of transaction T_1 that inserts 100 is unaffected by an operation of a concurrent transaction T_2

that removes 10, provided that there are nodes between 10 and 100.

However, in our initial solution, T_1 stores in its *read-set* every node it encounters during the traversal, including the one holding 10. Thus, if T_2 should commit before T_1 performs its validation, then T_1 would detect a conflict and abort.

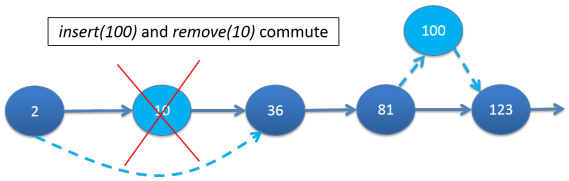

Figure 2: Example of commutative operations.

In order to avoid such unnecessary aborts, we keep the *read-set* as small as possible by adding to it only critical nodes whose update may lead to semantic violations. For example, consider a *contains(100)* operation that traverses the list and sees node n_2 with key 103 immediately after node n_1 with key 93 and returns false. Then *TX-commit* has to validate that no node with key 100 was inserted between the time when the *contains(100)* operation returned and the time when all the nodes in *write-set* were locked for validation during the commit. By the list's structure, it is enough to check that n_1 has not been removed and its *next* pointer has not changed. Thus, n_1 is the only node that needs to be added to the *read-set* during the *contains*.

More generally, every operation (*remove*, *insert*, or *contains*) with key k adds to *read-set* the node *predecessor(k)*, and *remove* also adds the successor in case its key is k. The iterator implementation does not change. In the example in Figure 2, *remove(10)* adds nodes with keys 2 and 10 to the *read-set*, while *insert(100)* adds only the node with key 81, and no conflict is detected in the validation.

In this way, the overhead for maintaining the *read-set* is small, and the number of aborts may drop dramatically.

3.1.3 Non-Transactional Index

So far we have reduced aborts in the validation phase to a minimum, but each operation still traverses the list from the beginning to the successor. This behavior has two drawbacks: First, the traversal is very slow, and so we would like to add "shortcuts" as in a skiplist. Second, the traverse can cause unnecessary aborts since it validates (the lock and version of) each node it passes through.

We observe, however, that the traversal is not an essential part of the *insert*, *remove*, and *contains* operations. In other words, a transaction is semantically unaffected by an interleaved transaction that changes a node on its path (unless traversed by an iterator). It only performs the check (in the transactional traversal) in order to ensure that (1) it traverses a segment of the list that is still connected to the head of the list and (2) it does not miss its target during the traversal. But

Algorithm 3 Transactional skiplist: procedures

```
procedure traverseTo(k)
    startNode ← index.getPrev(k)
    while startNode.locked ∨ startNode.deleted
        startNode ← index.getPrev(startNode.key)
    for n in list from startNode to successor(k)
        if getValidatedValue(n, readVersion) fails
            abort
    add predecessor(k) to read-set
    return ⟨predecessor(k), successor(k)⟩

procedure try-lock(write-set)
    for each node n ∈ write-set
        if ¬try-lock(n)
            return false
    return true

procedure validate(read-set)
    for each n ∈ read-set
        if getValidatedValue(n, readVersion) fails
            return false
    return true

procedure do-update(write-set)
    for each ⟨node, field, newValue⟩ ∈ write-set
        if (field ≠ ⊥)
            node.field ← newValue
        node.version ← writeVersion

procedure update-index(index-todo)
    for each ⟨node, op⟩ ∈ index-todo
        if op = remove
            index.remove(node)
        else                                    ▷ op = insert
            index.insert(node)
            if node.deleted
                index.remove(node)
```

if an operation with key k could somehow guess *predecessor(k)*, then it could forgo the traversal and start the operation from there. Or if it could guess any node n between the head of the list and *predecessor(k)*, it could start from n.

To facilitate such guessing, we employ an *index* with the following API: *add(node n)*, *remove(node n)*, and *getPrev(key k)*, where *getPrev(k)* returns the node with the biggest key that is smaller than k among all nodes that were previously added to the index and have not been removed.

Such an index can be provided by a standard (thread-safe) skiplist CDSL (e.g., using a textbook algorithm [25]). We update it, in a lazy way, outside transactions.

The index is used as follows: An operation with key k first calls the index's *getPrev(k)* method to obtain a node with a smaller key than k, and then starts the traverse from that node. Ideally, the index returns *predecessor(k)*, but as we later explain, this is not guaranteed. Following a successful *TX-commit*, the thread updates the index by calling *add(n)*

(resp. *remove(n)*) for every node n pertaining to a key k such that *insert(k)* (resp. *remove(k)*) was executed in the transaction.

It is important to notice that the index accesses are not part of the transaction: the index is updated outside the transaction (after commit), and reads from the index are not validated. Therefore, index accesses cannot cause any aborts. On the flip side, the lazy update implies that another thread that invokes *getPrev(k)* after the transaction has committed may receive in return a node that precedes k's predecessor, or even a node that has already been removed from the list. This scenario is illustrated in Figure 3. If the first case occurs, then the executing thread simply traverses a segment of the list until it reaches the appropriate successor, which may slow the operation down, but does not violate correctness. (Notice that *getPrev(k)* never returns a node that follows *predecessor(k)* in the list).

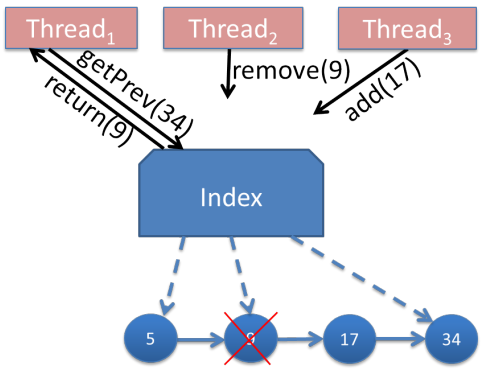

Figure 3: Example scenario where the non-transactional index may return obsolete values. Black arrows represent operations, and hanging arrows represent pending ones (or ones that will be invoked in the future). Blue dashed arrows point to nodes that were added to and not removed from the index, and thus may be returned by *getPrev* operations. In this example $Thread_1$ invokes *getPrev(34)* after $Thread_2$ commits a transaction that removes node 9 and $Thread_3$ commits a transaction that inserts node 17, but before they update the index accordingly.

The latter case is a bit more subtle. To deal with it we store an additional *deleted* bit in every node, indicating whether the node has been removed from the list. A commit therefore does not free the removed node, but instead sets the *deleted* bit (in addition to updating the node's version to be *writeVersion*). Now if the index returns a removed node, it is guaranteed that its *deleted* bit is true. The operations therefore check this bit, and if it is set, invoke *getPrev(k)* again, but this time with k equal to the removed node's key. Deleted nodes are eventually freed using a standard epoch-based memory reclamation approach [13].

We now describe how we update the index. During a transaction, the executing thread maintains a set, *index-todo*, of the *add* and *remove* operations it needs to perform. Once a transaction commits successfully, it updates the index. However, since the index is updated outside the scope of the transaction, a race can lead to a scenario where a transaction T adds to the index a node that has already been removed after T had committed. In this case, we have T remove the deleted node from the index lest it will be "stuck" in the index forever. To this end, after adding a node to the index, a transaction checks its deleted bit, and if it is set, removes the node. Note that the opposite scenario is not possible: since nodes are identified by pointers, and memory is not reclaimed while some node still holds the pointer to it, nodes are unique. Thus, an added node cannot be removed from the index by an older remove. Races may, however, cause the index to hold multiple nodes with the same key. If a new node with key k is inserted, an old remove operation for an earlier node with key k does not remove the new one.

3.2 Composing Multiple Objects in a Library

One of the strengths of our concept is that multiple objects can be accessed in the same transaction. We now explain how we implement this.

To ensure opacity, all objects from our library share a common GVC, which is read at the beginning of each transaction. A shared GVC allows us to support arbitrary dynamic transactions, meaning that we do not have to know the set of accessed objects at the beginning of a transaction. However, if one can give up this kind of dynamism, then each object may have a private GVC, as discussed in Section 4.

During a transaction, a thread maintains an *object-set* of the objects it accesses. Each object in *object-set* has its own *read-set*, *write-set*, and *index-todo*. These sets may be structured differently in different object types. For example, the queue data structure we present below does not employ an index, and hence its *index-todo* is empty. And, unlike the skiplist's *write-set*, the queue's *write-set* includes an ordered set, because the order in which items are enqueued ought to be preserved.

At the end of a transaction, *TX-commit* uses two phase locking, i.e., locks the appropriate items in all objects, then validates all of them, then updates all objects, and finally releases all locks. We note objects may define their own locking policies— our skiplist employs fine-grain locking while our queue relies on a single global lock for the entire queue. These per-object policies are tailored to the contention level and attainable parallelism in each object. While a skiplist can enjoy a high degree of parallelism using fine-grain locks, queues induce inherent contention on the head and tail of the queue and are thus better suited for coarse-grain locking.

The pseudo-code for the generic *TX-commit* appears in Algorithm 4. First we lock the *write-sets* (line 4) and validate the *read-sets* (line 7) of all objects using the appropriate object-specified methods. If the locking and validation is successful for all objects, we can safely commit the transaction. (Otherwise, we abort).

In the second phase, we *add-and-fetch* the GVC into *writeVersion* (line 10), update the objects (line 12) according to the *write-set* and *writeVersion* (again, each object according to its specific implementation), and release the locks (line 13). At this point the transaction is committed. Finally, we apply lazy updates where applicable (line 14). For example in a skiplist, we update its index.

Algorithm 4 Transactional library: generic commit

```
 1: local variables:
 2:     object-set: set of objects

 3: TX-commit()
 4:     for each obj ∈ object-set                          ▷ lock
 5:         if ¬obj.try-lock(obj.write-set)
 6:             release all acquired locks and abort
 7:     for each obj ∈ object-set                          ▷ validate
 8:         if ¬obj.validate(obj.read-set)
 9:             release all acquired locks and abort
10:     writeVersion ← GVC.add-and-fetch()
11:     for each obj ∈ object-set                          ▷ update objects
12:         obj.do-update(obj.write-set)
13:     release all acquired locks                         ▷ transaction is over
14:     for each obj ∈ object-set                          ▷ lazy update
15:         obj.update-index(obj.index-todo)
```

3.3 Queue

Notice that a queue has two points of contention (its head and tail), so accessing it optimistically and deferring the validation to commit time as we did with the skiplist is likely to lead to many aborts. Instead, we use a pessimistic (lock-based) approach with eager *dequeues* (taking place during the transaction) and commit-time *enqueues*. In order to satisfy opacity of transactions spanning queues and additional objects, each queue has a version and we use the GVC to ensure that all transactions see consistent states.

enqueue*:* During a transaction, the thread maintains a local queue, (essentially an ordered list), in each queue object's *write-set*. In every *enqueue(value)* operation, it simply enqueues *value* to the appropriate *write-set*. In contrast to queues that run on top of optimistic STMs, we do not track the current head of the queue in a read-set. At the end of the transaction, if the first phase of *TX-commit* is successful, the *do-update* function (Algorithm 4, line 12) simply enqueues the entire local queue from the *write-set* at the end of the (locked) queue.

dequeue*:* Since *dequeue* operations return values, we do not defer their execution to the commit phase as we do for *enqueue*. Note that if two interleaved transactions perform *dequeue* operations on the same queue, then at least one of these transactions will eventually abort. It is best to absorb this conflict as soon as possible, and so we implement *dequeue* in a pessimistic way.

We lock the queue the first time *dequeue* is invoked on a given queue in a given transaction, and release the lock only at the end of the transaction (commit or abort). In addition, to ensure opacity, after acquiring a lock we check the queue's version in order to validate that nothing was enqueued to or dequeued from the queue since the transaction began. If the check fails or we fail to acquire the lock, the transaction is immediately aborted. Otherwise, we return the node at the head of the queue, but since the transaction may still potentially abort, we do not remove the node from the queue just yet. Instead, we advance a transaction-local pointer and store in the *write-set* an indication that this node ought to be dequeued. Ensuing *dequeue* operations on the same queue in the same transaction also advance this pointer, update the *write-set*, and return the next available node. If the queue is empty, we check our local queue in the *write-set* to see if a previous enqueue was performed during the transaction. The dequeued nodes are removed from the queue by the *do-update* function during the second phase of *TX-commit*.

TX-commit*:* Here we refer to the procedures of the generic commit from the previous section (Algorithm 4). The procedure *try-lock* simply tries to lock the queue if the transaction hadn't locked it yet. The queue has an empty *read-set*, since it does not perform optimistic reads. It also has an empty *index-todo* as it has no index. The *validate* and *update-index* functions hence simply do nothing. The procedure *do-update(write-set)* removes the nodes returned by *dequeue* operations during the transaction, enqueues all the nodes in local queue included in the *write-set*, and updates the queue's version to be *writeVersion*.

3.4 Fast Abort-Free Singletons

An important aspect of our idea is to extend CDSLs with transactional support *without hampering legacy CDSL operations*. The algorithms described above are efficient for transaction processing (as shown in Section 5), but may incur a non-negligible overhead relative to stand-alone CDSL operations. In particular, having all mutating operations increment the GVC creates a contention point. This contention point is essential for achieving opacity, where operations pertaining to the same transaction see a consistent state of the data structure, but is redundant for stand-alone CDSL operations. Another problem introduced by transactions is the possibility of abort; legacy code using the CDSL non-transactionally will not have mechanisms for handling abort exceptions.

In this section we explain how we support efficient CDSL operations without the synchronization cost of transactions and without aborts. At the same time, from a semantic perspective, these CDSL operations are seen as singleton transactions preserving the system opacity, and whose atomicity relative to other transactions is preserved.

Detecting singleton updates. To ensure opacity of transactions in the presence of singletons, we augment each

skiplist node and every queue with an additional bit, *singleton*, which indicates whether the last update to this node (or queue) was by a singleton.

Skiplist singletons. Like their transactional counterparts, the skiplist's stand-alone operations use the index to find the successor and predecessor of their argument. But unlike operations called within a transaction, they do not defer any work to commit time. In particular, they don't maintain read- and write-sets. Their traversals validate that accessed nodes are neither deleted nor locked; otherwise, the operation restarts. The read-only operation *contains* simply returns once it finds the successor.

An *insert(k)* operation first reads GVC and creates a new node. Then it finds *predecessor(k)* and *successor(k)*, and if the successor's key is not k, it tries to lock the predecessor. In case it succeeds, it validates that the *next* pointer is still *successor(k)* and the *deleted* bit is off (so the node was not removed), and if so it (1) sets the predecessor's *next* pointer to the address of the new node, (2) updates its version according to the read GVC and sets its *singleton* bit to true, and (3) releases the lock. In case the locking or validation fails, it searches for *predecessor(k)* again and retries.

A *remove(k)* operation first reads GVC, finds a successor with key k, locks the successor and predecessor of k, and validates that they are still the successor and predecessor of k. Then, it (1) updates the predecessor's *next* pointer to bypass the successor, (2) sets the successor's *deleted* bit to true, (3) updates the predecessor's and successor's versions according to the read GVC and sets their *singleton* bits to true. Then, it releases the locks. If the locking or validation fails, it releases the acquired locks and retries.

Upon completion, singletons update the index in the same manner as tranasctional operations.

Queue singletons. For simplicity and compatibility with the transactional version of the queue, we also use a pessimistic approach for stand-alone queue operations. We first lock the queue and read GVC, then perform the operation, update the queue's version according to the read GVC and set its *singleton* bit to true, and then release the lock.

Transaction adjustment. Since singletons do not increment the GVC, transactions cannot rely on versions alone to detect conflicts with singletons. To this end, we use the *singleton* bits. For atomicity, we enhance our transaction's conflict detection mechanism as follows: the validation phase of *TX-commit* now not only validates that there is no node in *read-set* with a bigger version than *readVersion*, but also checks that there is no node with a version equal to *readVersion* and a true *singleton* bit (indicating that there is a conflicting singleton). For opacity, every skiplist node and queue accessed during a transaction is checked in the same way. In both cases, if a node or queue with a version equal to *readVersion* and *singleton* bit set to true is checked, the transaction increments GVC, (to avoid future false aborts), and aborts.

Note that some of these aborts may be spurious, because the conflicting singleton had occurred before the transaction began. In order to minimize such aborts, it is possible to have the transaction try to serialize itself after the conflicting singleton instead of aborting. To this end, the transaction must re-validate that all the values in its read-set still reflect the current memory's state. However, relying on versions is insufficient in this case, because if an object in the read-set has a version equal to *readVersion* and a set *singleton* bit, it might have been over-written by an additional singleton since it was previously read. Instead, the transaction must validate that the object's *value* has not changed. Specifically, it can do the following: (1) validate that the *pointers* stored in the next fields of the nodes in *read-set* have not changed (note that an ABA problem cannot occur thanks to our epoch-based memory reclamation); and (2) verify that their *deleted* bits are false.

Finally, when a transaction commits successfully, it sets the *singleton* bit to false in every node (or queue) it updates.

4. Support for Library Composition

In this section 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 [10]. Our composition framework is based on the theory of Ziv et al. [51]. Such composition can provide support for fully general memory transactions, including ones that access scalars.

In Section 4.1 we describe an extended API and respective semantics required from composable libraries. Section 4.2 shows how we can use this API to compose a number of TDSLs. To make the discussion concrete, we then explain in Section 4.3 how the extended API is supported by our TDSL algorithm of the previous section, and how it can be supported by a generic TL2 implementation

4.1 Composable Libraries API and Semantics

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* [12], 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).

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

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

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 we give here a semi-formal description of the required properties, and refer the reader to a more formal treatment by Ziv et al. [51].

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 $op_1, op_2, \ldots op_k$. Using this notation, the *history* of a committed transaction T accessing two TDSLs is the following sequence of steps:

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

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:

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

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 subseqence 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. [51], 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 $op_1, \ldots, op_m$ 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 $op_1, \ldots, op_m$ 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).

$C3$: **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.

4.2 Composing TDSLs

Given a collection of n TDSLs satisfying conditions $C1$-$C4$, we compose them to create a single TDSL with the API defined in Section 2 as follows:

- *TX-begin*() calls B^i for all the composed TDSLs.
- *TX-commit*() calls $L^1, \ldots, L^n, V^1, \ldots, V^n, F^1 \ldots F^n$ unless an abort exception occurs.
- The handler for abort exceptions of the individual libraries calls $A^1, \ldots A^n$, and throws an abort exception in the composed library.

For a committed transaction T that uses TDSLs $1, \ldots, n$, any point p between L^n and V^1 is a linearization point with respect to all n TDSLs; hence p is also a linearization point for the composed library. Similarly, it can be shown that $C2$ is also satisfied by the composition.

An abort exception from any of the libraries causes all of them to abort and throws an abort exception for the composed one (this exception should be handled by the client of the composed library). This way we ensure that an aborted transaction leaves the library's state unaffected (as required by condition $C3$). Condition $C4$ is trivially satisfied since the composition does not create new blocking operations.

If a transaction uses a subset of the libraries, it need only call the transactional API in the ones it accesses. Note that this requires a transaction to know all the libraries it composes at the beginning of the transaction (in order to call B in all of them). One way to avoid this restriction in libraries implemented in a TL2-like manner is to share a GVC among all libraries and use one common *TX-begin* for all of them, as we did with our skiplist and queue in Section 3.

4.3 Examples of Composable TDSL Implementations

We now explain how our TDSL algorithm of Section 3 can support the above B, L, V, F, and A operations. The B operation is simply *TX-begin*.

The *TX-commit* operation of Algorithm 4 is divided into the following three operations:

- L (lines 4–5): try to lock the write-sets of all participating objects.
- V (lines 7–8): validate read-sets of all participating objects.
- F (lines 10–15): increment and fetch GVC, update all objects, release all acquired locks, and update all indexes.

The *TX-abort* operation is implemented by clearing all transaction-local data and unlocking all locks owned by the transaction.

The TL2 STM [10] can also be modified to support the above API and semantics. Similarly to our TDSL, it is required to divide the commit phase into three steps and expose an API to each of the steps; the first steps locks the write-set, the second step validates the read-set, and the last step increases the GVC, updates the data according to the write-set, and releases the acquired locks.

Furthermore, using the above API and the serialization windows guaranteed by properties $C1$ and $C2$, the approach of Ziv et al. [51] provides ways to compose TDSLs with other concurrency control schemes, e.g., with pessimistic two-phase locking on standard read-write variables [12], as well as more elaborate locking schemes like tree locking.

5. Evaluation

We implement our transactional library and evaluate its performance. Section 5.1 describes the experiment setup and methodology. In Section 5.2, we compare the singletons of our transactional skiplist to custom-tailored concurrent skiplist operations. Section 5.3 evaluates transactions of skiplist operations using our TDSL relative to a state-of-the-art STM and a transaction-friendly skiplist. In both cases, we run synthetic workloads using the Synchrobench microbenchmark suite [16]. Finally, in Section 5.4, we evaluate our library with Intruder [19, 41], 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.

5.1 Experiment Setup

All implementations are in C/C++. For our library's index, we use a standard textbook concurrent skiplist [25], which is based on [29] and [13], with epoch-based memory reclamation [13]. 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 the experiments in Sections 5.2 and 5.3 we use the Synchrobench framework [16] 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 consider three representative workload distributions: (1) a *read-only* workload with only *contains(key)* operations, (2) an *update-only* workload comprised of 50% *insert(key)* and 50% *remove(key)* operations, and (3) a *mixed* workload consisting of 50% *contains(key)*, 25% *insert(key)*, and 25% *remove(key)* operations.

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).

5.2 Singletons

An important goal for a TDSL is, in particular, to be a good concurrent data structure, i.e., avoid penalizing stand-alone operations for the library's transaction support. We therefore compare our library's singletons to our baseline non-transactional skiplist implementation (our index) as well as to the leading CDSL solutions provided with Synchrobench. The solutions we compare to are listed in Table 2; we note that Gramoli [16] has shown that these particular skiplists outperform other published solutions.

The throughput results for the three workloads described in Section 5.1 above appear in Figure 4. The experiments show that our support for transactions induces virtually no overhead on stand-alone operations, which execute as fast as in the baseline library. Moreover, our library's skiplist

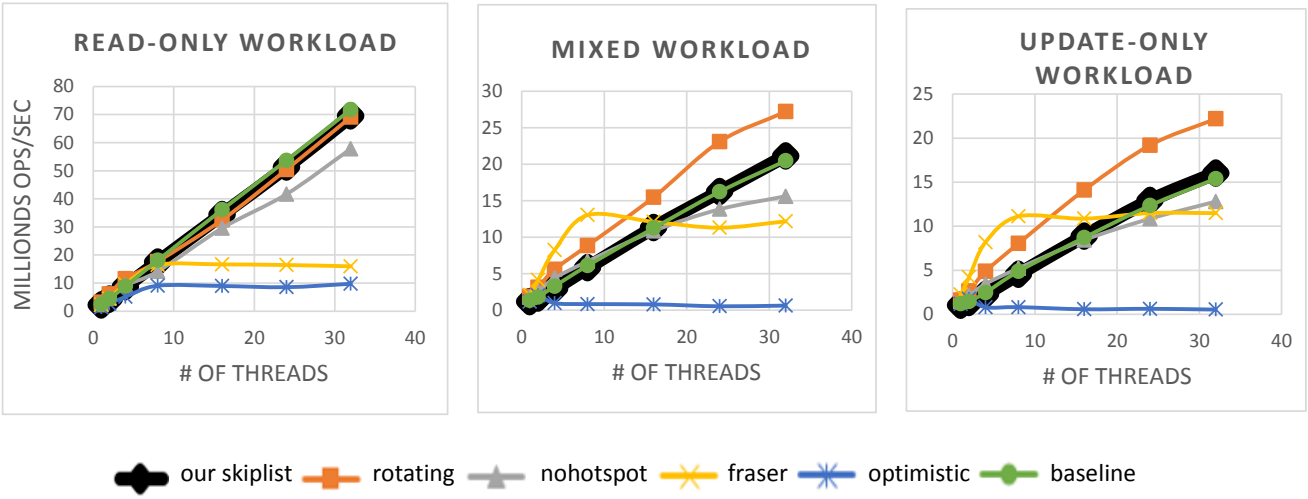

Figure 4: Throughput of our skiplist's singletons versus best-in-class custom-tailored skiplists.

Algorithm	Source
Baseline	Herlihy and Shavit [25], based on [29],[13]
Optimistic	Herlihy et al. [28]
Fraser	Fraser [13]
No hot spot	Crain et al. [9]
Rotating	Dick et al. [11]

Table 2: Evaluated CDSL skiplists.

operations are comparable with the best-in-class custom-tailored skiplists.

5.3 Transactions

We next experiment with transactions spanning multiple skiplist operations. We compare our transactional skiplist to a sequential skiplist running over TL2 [3], which is provided in Synchrobench [16] — we refer to this skiplist as *SeqTL2*. In addition, we evaluate a transaction-friendly skiplist [8] running over TL2, which we refer to as *FriendlyTL2*. Its main idea is to reduce conflicts by deferring work to a dedicated background thread that maintains the skiplist. In our measurements, we do not count transactions (and aborts) executed by this dedicated thread. We implemented *FriendlyTL2* by manually translating the transaction-friendly skiplist in the Java version of Synchrobench to C.

In this experiment, each transaction chooses a number of operations between 1 and 7 uniformly at random, and then selects the operations according to the designated workload, as explained in Section 5.1 above.

Throughput. Figure 5 shows the throughput of successful transactions with our TDSL, *SeqTL2*, and *FriendlyTL2*. We see that our transactions are much faster than *SeqTL2*'s, and in scenarios that involve reads, also much faster than *FriendlyTL2*. In case of read-only transactions, our throughput is twice that of *SeqTL2* and four times that of *FriendlyTL2*. Our advantage over *SeqTL2* grows as we introduce updates to the mix, and is most pronounced when there are only update transactions, in which case our TDSL is an order of magnitude faster than *SeqTL2*. Here, the abort rate is in play in addition to the overhead, as we explain shortly, and *FriendlyTL2* mitigates this gap by also reducing abort rates.

Another point to notice is that in all workloads, our performance and *FriendlyTL2*'s performance continue to scale with the number of threads, while *SeqTL2* saturates at 16 threads in workloads that include updates, presumably due to increasing abort rates. For example, when running a hybrid workload, our library reaches 3.65 million transactions per second with 32 threads whereas *SeqTL2*'s throughput peaks at 0.76 million transactions per second with 16 threads.

FriendlyTL2 is mostly geared towards update-only workloads, where it can avoid most conflicts by deferring the data structure's maintenance to a single background thread, and indeed its performance is close to our TDSL's in this scenario. However, in read-only scenarios, *FriendlyTL2* is even slower than the sequential implementation. This may be due to the overhead induced by the unneeded background thread as well as due to changes made in the data structure to facilitate batch updates. Interestingly, *FriendlyTL2*'s throughput is better in the update-only workload than in the mixed one. We believe that this is due to the non-trivial interaction between transactions and the maintenance thread.

Abort rates. Figure 6 shows the abort rates of all solutions in the update-only workload. We can see that while the abort rate of *SeqTL2* grows linearly in the number of threads, the abort rates of our TDSL and *FriendlyTL2* remain small for any number of threads. The fact that our read-set is restricted to a small subset of *SeqTL2*'s, together with our non-transactional index, dramatically reduces the abort rate, while *FriendlyTL2* reduces the abort rate by deferring most updates to a single thread. In Figure 7 we zoom in and

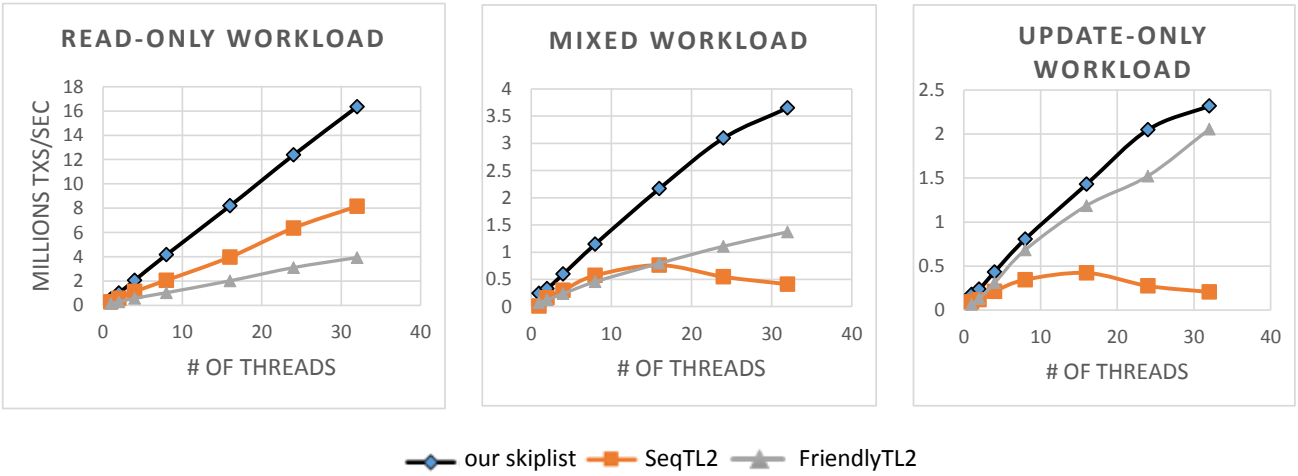

Figure 5: Throughput of our transactional skiplist versus a sequential skiplist on top of TL2 and a transaction-friendly skiplist on top of TL2.

compare the abort rates of our TDSL and *FriendlyTL2* in the update-only workload. While both are small compared to *SeqTL2*, we still see that our abort rate is lower and scales better in the number of threads than the abort rate of *FriendlyTL2*. The abort rates of all solutions in the mixed workload are similar and so we omit them.

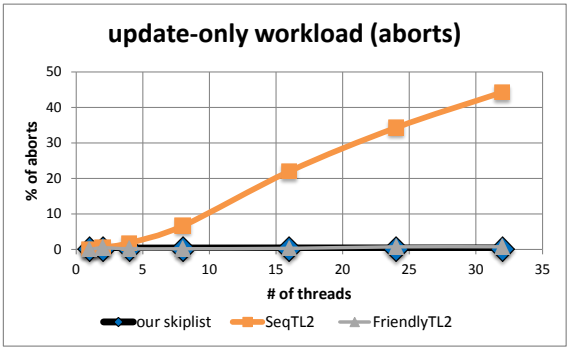

Figure 6: Abort rate of our transactional skiplist versus a sequential skiplist on top of TL2 and a transaction-friendly skiplist.

5.4 The Intruder Benchmark

Having tested our singletons and transactions with synthetic workloads, we turn to evaluate the entire library with a realistic application. We use our library to implement the data structures required by Intruder, a signature-based network intrusion detection system. Intruder scans messages pertaining to network flows and matches them against a known set of intrusion signatures.

The benchmark [19, 41] is multi-threaded. Incoming messages are first inserted into a shared queue. The messages are then dequeued and handed over to threads executing the reassembly task outlined in Section 2 above. This phase uses a map that holds sets, and enqueues the reassembled flows into another queue. Thus, the benchmark exercises the different data structures in our library. It also uses singletons and iterators.

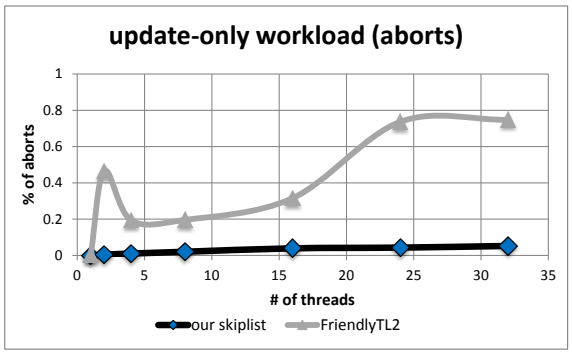

Figure 7: Abort rate of our transactional skiplist versus the transaction-friendly skiplist, zoomed in.

We run the benchmark with 131,072 traffic flows consisting of either 256 or 512 messages each (parameter settings of -n131072 -l256 or -l512 in STAMP). We run the intruder code from STAMP modified to use our library as well as the TL2 implementation taken from [3]. Figures 8(a) and 8(b) show the speedup of both algorithms over a sequential run (single-threaded, no synchronization mechanisms used) for the two flow sizes. Figure 8(c) shows the percentage of aborts in the experiment with larger flows; the results with smaller flows were similar. We see that with the smaller flow, our library is 9x faster than TL2, and with the larger flows, it runs up to 17x times faster.

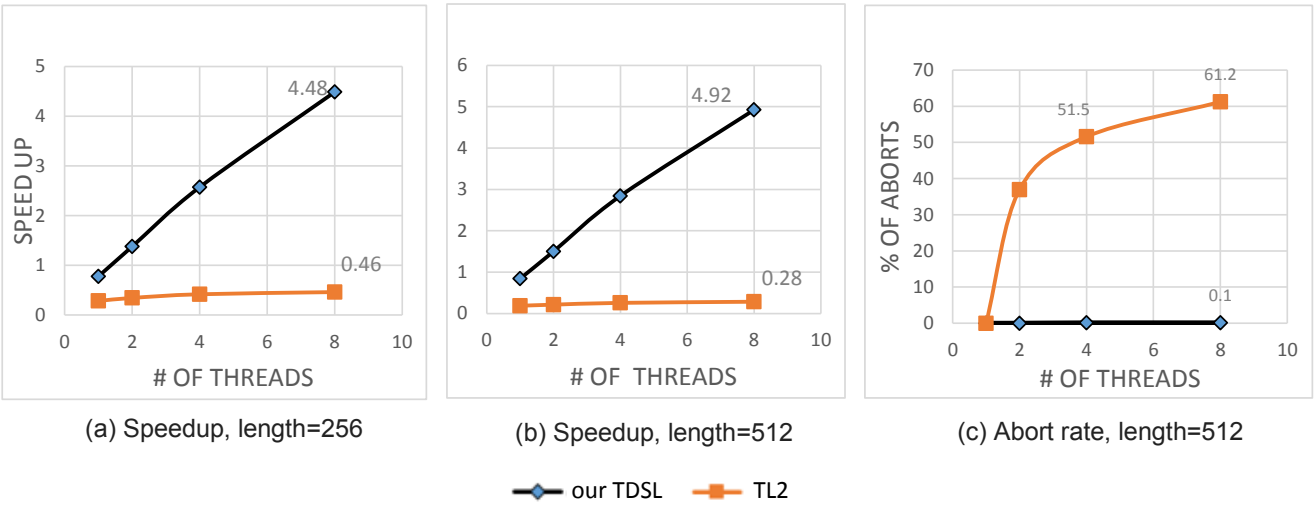

Figure 8: Intruder with our library and TL2: speedup over sequential execution and abort rate.

Furthermore, our library achieves linear speedup up to 8 threads, whereas TL2 is actually slower than a sequential implementation for any number of threads. This is in large part due to TL2's high abort rate (Figure 8(c)), which causes much of the work it does to go to waste. While significant abort rates were observed in TL2 also in skiplist-only transactions (see Section 5.3 above), they increased moderately to a bit over 40% with 32 threads. Here, on the other hand, TL2's abort rate exceeds 50% even with four threads. We believe that this is in large part due to the use of optimistic synchronization for queues, where contention is likely. With our approach, which uses locks for queue operations, the abort rates are negligible. This underscores the importance of selecting individual concurrency-control policies accommodating the unique characteristics of each data structure.

With more than 8 threads, the speedup decreases. However, this is most likely due to the fact that in our platform, each 8-core port has its own DRAM bank, and Intruder induces a substantial amount of data sharing among threads. For example, it uses a shared dispatch queue from which all threads retrieve their work items, and a shared egress queue where all reassembled flows are inserted for further processing. Such data sharing stresses the machine's cross-chip bandwidth and downgrades the performance of the application.

6. Related Work

Transactional memory. Many works have been dedicated to realize atomic transactions via *software transactional memory* approaches [10, 26, 43, 43]. Typically, STMs dynamically resolve inconsistencies and deadlocks by rolling back transactions. However, in spite of a lot of effort, existing STM approaches have not been widely adopted for real-world concurrent programming due to various concerns, including high run-time overhead, poor performance and limited applicability [6].

In a sense, our transactional skiplist algorithm can be seen as a combination of the STM TL2 algorithm [10] with techniques for specialized concurrent data structures [13, 25], while our queues are managed by a different approach that is better suited for them. Such optimizations are facilitated by forgoing the generality of STM toolkits, and restricting transactions to a well-defined set of data structure operations.

Automatic synchronization via static analysis. Several approaches have been proposed to automatically realize transactions by inferring synchronization via static analysis algorithms (e.g., [7, 32, 40]); some of them deal with high-level data structure operations [15]. In contrast, in this paper we do not rely on static analysis.

Semantic synchronization. Some synchronization approaches are designed to utilize semantic properties of data structure operations (e.g., [14, 15, 23, 34, 35, 42]). The main idea is to improve parallelism by detecting conflicts on data structure operations rather than on low level memory accesses. In these approaches two commutative data structure operations can be seen as not conflicting even if their implementations access the same memory location. Our approach also employs semantic synchronization. But in contrast to these previous works, which only consider the data structure's API, our work also utilizes the internal structure and implementation details of the data structure. This allows us to use custom-tailored optimizations such as the non-transactional index of our skiplist. Moreover, in contrast to our approach, it is not clear how semantic approaches can be used to support efficient singleton operations.

One example of a semantic approach is boosting [23], which presents a methodology for bundling linearizable ob-

ject operations into transactions. Boosting is based on a semantic variant of two phase locking, in which the data structure operations are protected by a set of abstract locks. Boosting has a number of limitations, including restricted generality (for example, it is not clear how to efficiently implement a thread-safe iterator), the need to support compensating actions via bookkeeping, and sensitivity to parameters (like the timeouts which are used to detect deadlocks).

Another example of semantic synchronization is open nesting [42]. However, this approach is intended for experienced programmers, as it requires programmers to explicitly use abstract locks and also to provide an abstract compensating action for each of its open nested transactions, to be used in case of abort.

Our approach is inspired by the idea of foresight [14], which also supports atomicity of a number of CDSL operations. But in contrast to our library, it requires external information about the transactions' code, which is computed via static analysis. Furthermore, our solution utilizes optimistic synchronization (transactions may be aborted), whereas synchronization in foresight [14] is fully pessimistic (transactions are never aborted).

Composing operations. Several works have suggested approaches for composing multiple CDSL operations into larger atomic operations. One example is reagents [50], which provides a set of building blocks for writing concurrent data structures and synchronizers that express concurrency algorithms at a higher level of abstraction. However, with reagents, concurrency control is not transparent to the programmer. In particular, reagents programmers need to use non-trivial fine-grained data structure operations and put them together in a way that implements the desired functionality.

Another recent example is RLU [39], which is a tool to compose multiple data structure updates in a way that does not hamper the efficiency of data structure read-only operations. Similarly to our work, their work is also inspired by STM mechanisms for synchronizing between readers and writers that wish to update multiple locations atomically. But in contrast to us, they provide a tool for building efficient complex CDSLs (e.g., doubly linked lists) rather than TDSLs.

Data structures for STMs. Some works describe data structures that are compatible with STMs [4, 8, 31, 38]. Similarly to our skiplist, they use operation semantics to reduce the read-set size, leading to fewer aborts and smaller validation overhead. The data structures in [31, 38] and [4] translate their high-level operations into low level ones, and the transaction-friendly skiplist [8] delays some non-critical work that may lead to aborts to a dedicated thread that runs in the background. These data structures are designed to correctly and efficiently work within STM transactions, and do not include support for fast abort-free singletons as we provide. Furthermore, whereas in our approach the synchronization algorithm is part of the library implementation, in these works, the actual synchronization algorithm (or at least part of it) is implemented outside the data structure library, i.e., handled by the external STM. Therefore, their data structures must expose some abstract conflict information for the STM to use. This may limit data structure implementations, for example, our non-transactional index, our fast singletons, (which do not update the GVC), and the ordered write-set we use in the queue are not readily amenable to such abstract representations. *Software transactional objects* (STO) [31], which were developed independently and concurrently with our work, advocate co-design of the transactional data structures and the STM.

7. Conclusions

We introduced the concept of transactional data structure libraries, which adds transaction support to CDSLs. Such transactions offer composability and may span arbitrary sequences of operations on any number of data structures. We illustrated this concept by presenting algorithms for transactional sets, maps, and queues, and offering them as a C++ library. Previous studies have shown that many concurrent programs employ such data structures and require atomic transactions spanning multiple operations [47]. Moreover, we have outlined a methodology for composing such transactional data structure libraries with arbitrary synchronization mechanisms like STM, allowing for fully general transactions alongside high performance specialized ones comprised of only data structure operations.

Our TDSL caters stand-alone operations at the speed of a custom-tailored CDSLs, and at the same time, provides the programmability of transactional synchronization. Our TDSL transactions are also much faster than those offered by a state-of-the-art STM toolkit, (tenfold faster in synthetic update-only benchmarks, up to 17x faster in a non-trivial concurrent application). This is thanks to our ability to reduce overheads and abort rates via specialization to particular data structure organizations and semantics. In particular, we can use the most appropriate concurrency-control policy and for each data structure regardless of the approaches used for other data structures.

While we illustrated the concept here for one particular library supporting queues, maps, and sets, we believe that the idea is broad reaching, and many existing CDSLs can be extended in a similar manner to offer transactions on top of the CDSL operations they provide.

Acknowledgments

We thank Edward Bortnikov for many helpful discussions and Tomer Morad for assistance with running the experiments. We are grateful to the referees for their many detailed comments, and in particular, Tony Hosking for shepherding the paper.

References

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

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

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

[4] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. Transactional predication: High-performance concurrent sets and maps for stm. In *PODC*, 2010.

[5] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. A practical concurrent binary search tree. In *ACM Sigplan Notices*, 2010.

[6] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software transactional memory: Why is it only a research toy? *Queue*, 6(5):40:46–40:58, Sept. 2008.

[7] S. Cherem, T. Chilimbi, and S. Gulwani. Inferring locks for atomic sections. In *PLDI*, 2008.

[8] T. Crain, V. Gramoli, and M. Raynal. A contention-friendly methodology for search structures. Technical report, RR-1989, INRIA, 2012.

[9] T. Crain, V. Gramoli, and M. Raynal. No hot spot non-blocking skip list. In *ICDCS*, pages 196–205, 2013.

[10] D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. In *Distributed Computing*, pages 194–208. Springer, 2006.

[11] I. Dick, A. Fekete, and V. Gramoli. Logarithmic data structures for multicores. Technical report, 697, University of Sydney, 2014.

[12] 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.

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

[14] G. Golan-Gueta, G. Ramalingam, M. Sagiv, and E. Yahav. Concurrent libraries with foresight. In *PLDI*, 2013.

[15] G. Golan-Gueta, G. Ramalingam, M. Sagiv, and E. Yahav. Automatic scalable atomicity via semantic locking. In *PPOPP*, 2015.

[16] V. Gramoli. More than you ever wanted to know about synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorithms. In *PPoPP*, pages 1–10, 2015.

[17] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *PPoPP*, pages 175–184, 2008.

[18] R. Guerraoui, T. A. Henzinger, and V. Singh. Software transactional memory on relaxed memory models. In *Computer Aided Verification*, 2009.

[19] B. Haagdorens, T. Vermeiren, and M. Goossens. Improving the performance of signature-based network intrusion detection sensors by multi-threading. In *WISA'04*, pages 188–203, 2005.

[20] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory, 2nd Edition*. Morgan and Claypool Publishers, 2010.

[21] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. N. Scherer III, and N. Shavit. A lazy concurrent list-based set algorithm. In *Principles of Distributed Systems*, pages 3–16. 2006.

[22] M. Herlihy. The transactional manifesto: Software engineering and non-blocking synchronization. In *PLDI*, 2005.

[23] M. Herlihy and E. Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *PPoPP*, 2008.

[24] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, pages 289–300, 1993.

[25] M. Herlihy and N. Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008.

[26] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *PODC*, 2003.

[27] M. Herlihy, Y. Lev, V. Luchangco, and N. Shavit. A provably correct scalable concurrent skip list. In *OPODIS*, 2006.

[28] M. Herlihy, Y. Lev, V. Luchangco, and N. Shavit. A simple optimistic skiplist algorithm. In *SIROCCO*, pages 124–138. 2007.

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

[30] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, July 1990.

[31] N. Herman, J. Priya Inala, Y. Huang, L. Tsai, E. Kohler, B. Liskov, and L. Shrira. Type-aware transactions for faster concurrent code. In *EuroSys*, 2016. To appear.

[32] M. Hicks, J. S. Foster, and P. Prattikakis. Lock inference for atomic sections. In *ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, June 2006.

[33] Intel. Intel transactional synchronization extensions. In *Intel architecture instruction set extensions programming reference*. 2012.

[34] E. Koskinen, M. Parkinson, and M. Herlihy. Coarse-grained transactions. In *POPL*, pages 19–30, 2010.

[35] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *PLDI*, 2007.

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

[37] D. 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.

[38] V. J. Marathe, M. F. Spear, and M. L. Scott. Transaction safe non-blocking data structures. In *DISC*, 2007.

[39] A. Matveev, N. Shavit, P. Felber, and P. Marlier. Read-log-update: a lightweight synchronization mechanism for concurrent programming. In *SOSP*, pages 168–183, 2015.

[40] B. McCloskey, F. Zhou, D. Gay, and E. Brewer. Autolocker: synchronization inference for atomic sections. In *POPL*, 2006.

[41] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. Stamp: Stanford transactional applications for multi-processing. In *Workload Characterization, IISWC 2008*, pages 35–46, 2008.

[42] Y. Ni, V. S. Menon, A.-R. Adl-Tabatabai, A. L. Hosking, R. L. Hudson, J. E. B. Moss, B. Saha, and T. Shpeisman. Open nesting in software transactional memory. In *PPoPP*, pages 68–78, 2007.

[43] D. Perelman, A. Byshevsky, O. Litmanovich, and I. Keidar. Smv: Selective multi-versioning stm. In *DISC*, pages 125–140, 2011.

[44] W. Pugh. Concurrent maintenance of skip lists. 1998.

[45] R. Rajwar and J. R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *MICRO*, 2001.

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

[47] O. Shacham, N. G. Bronson, A. Aiken, M. Sagiv, M. T. Vechev, and E. Yahav. Testing atomicity of composed concurrent operations. In *OOPSLA*, pages 51–64, 2011.

[48] N. Shavit and I. Lotan. Skiplist-based concurrent priority queues. In *IPDPS*, pages 263–268, 2000.

[49] H. Sundell and P. Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. In *IPDPS*, 2003.

[50] A. Turon. Reagents: expressing and composing fine-grained concurrency. In *ACM SIGPLAN Notices*, volume 47, pages 157–168, 2012.

[51] O. Ziv, A. Aiken, G. Golan-Gueta, G. Ramalingam, and M. Sagiv. Composing concurrency control. *SIGPLAN Not.*, 50(6), June 2015.