Lock-free Transactional Vector[*]

Kenneth Lamar
University of Central Florida
Orlando, Florida
kenneth@knights.ucf.edu

Christina Peterson
University of Central Florida
Orlando, Florida
clp8199@knights.ucf.edu

Damian Dechev
University of Central Florida
Orlando, Florida
dechev@cs.ucf.edu

ABSTRACT

The vector is a fundamental data structure, offering constant-time traversal to elements and a dynamically resizable range of indices. While several concurrent vectors exist, a composition of concurrent vector operations dependent on each other can lead to undefined behavior. Techniques for providing transactional capabilities for data structure operations include Software Transactional Memory (STM) and transactional transformation methodologies. Transactional transformations convert concurrent data structures into their transactional equivalents at an operation level, rather than STM's object or memory level. To the best of our knowledge, existing STMs do not support dynamic read/write sets in a lock-free manner, and transactional transformation methodologies are unsuitable for the vector's contiguous memory layout. In this work, we present the first lock-free transactional vector. It integrates the fast lock-free resizing and instant logical status changes from related works. Our approach pre-processes transactions to reduce shared memory access and simplify access logic. This can be done without locking elements or verifying conflicts between transactions. We compare our design against state-of-the-art transactional designs, GCC STM, Transactional Boosting, and STO. All data structures are tested on four different platforms, including x86_64 and ARM architectures. We find that our lock-free transactional vector generally offers better scalability than STM and STO, and competitive performance with Transactional Boosting, but with additional lock-free guarantees. In scenarios with only reads and writes, our vector is as much as 47% faster than Transactional Boosting.

CCS CONCEPTS

• **Software and its engineering → Concurrent programming structures**; • **Computing methodologies → Concurrent algorithms**; • **Computer systems organization →** *Multicore architectures.*

KEYWORDS

vector, transactional memory, lock-free

[*]This work is supported by the National Science Foundation under NSF award No. 1717515 and NSF award No. 1740095.

ACM Reference Format:
Kenneth Lamar, Christina Peterson, and Damian Dechev. 2020. Lock-free Transactional Vector. In *The 11th International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM'20), February 22, 2020, San Diego, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3380536.3380543

1 INTRODUCTION

The *vector* is a fundamental data structure. Similar to an array, it has constant time traversal and performs read and write operations via index-based access. It features the ability to resize dynamically when elements are added to or removed from the structure. It can also use stack semantics to push and pop values at the end of the list. The versatile functionality of the vector makes it a suitable choice for sparse matrix representation and data storage for in-memory databases. Consequently, there is an imminent need for vectors that are thread-safe in a concurrent environment.

Several concurrent vectors are proposed in literature that suffice for single operation requests [3, 5, 23]. When using a standard multithreaded vector, a composition of concurrent vector operations may be interleaved with the vector operations of other threads. Attempting to execute such a composition can lead to undefined behavior. To overcome this limitation, it is often desirable to execute a composition of operations in the form of a transaction. A *transaction* is a sequence of operations executed by a single thread that appears to execute atomically and in isolation from other transactions. The *isolation* property ensures that transactions cannot see the partial effects of any other transaction; they can only see the effects of completed transactions. We consider a concurrent data structure to be transactional if it supports a composition of operations that execute atomically and in isolation.

Software Transactional Memory (STM) [21] operates using a per-transaction read/write set (RWSet) of objects or memory values to track and manage conflicts. STM low-level memory conflicts may not always correspond to high-level conflicts between operations, and its usage often results in major runtime overhead, making it impractical for the development of transactional data structures due to frequent accesses of data structure hot spots [2].

Transactional transformations convert concurrent data structures into their transactional equivalents at a higher level than STM's low-level memory access. Implementations such as Transactional Boosting (TB) [12] and Software Transactional Objects (STO) [15] consider transaction conflicts at the operation-level and object-level, respectively, to prevent *false conflicts*, which occur when a transaction blocks another transaction due to an overlapping RWSet, even though a real, semantic conflict does not exist. *Lock-freedom* is a progress property that guarantees that some thread makes progress in a finite number of steps, regardless of how many threads are executing. *Logical rollbacks* are used to logically roll back all partial transaction modifications instantly. Lock-free Transactional Transformation (LFTT) [24] furthers TB by enabling both lock-freedom and logical rollbacks. Among the existing transactional transformation designs we are aware of, TB and STO cannot provide lock-freedom due to the use of mutual exclusion while LFTT's node-based approach is prone to fragmentation.

In this paper, we propose the first lock-free transactional vector. It supports a subset of the sequential STL vector's operations: `read`, `write`, `pushBack`, `popBack`, `size`, and `reserve`. This is the same set of operations supported by existing lock-free vectors [3, 5, 23]. We also use a *bucket array*, an underlying structure using buckets of exponentially sized arrays to enable fast concurrent resizing, based on the work of Dechev et al. [3]. We support bounds checking to prevent an out-of-bounds index from being read or written. Our lock-free transactional vector offers several improvements over existing work. Unlike alternative transaction frameworks, our implementation benefits from a bespoke design optimized for the transactional vector. Our design uses a logical status inference system inspired by LFTT, so it does not suffer from the physical rollbacks performed by the alternatives, STM, TB, and STO. Unlike some STMs and STO, our transactional vector maintains lock-free progress guarantees.

Our transactional vector uses *strict serializability* as a correctness condition, guaranteeing that transactions will appear to have occurred in an order equivalent to some sequential history that is consistent with the real-time order of execution. Our vector also supports *opacity* [9], which additionally guarantees that non-committed transactions cannot access inconsistent states. A *static RWSet* is a RWSet where the memory locations are known before the transaction starts. A *dynamic RWSet* is a RWSet where the memory locations are determined during the transaction. Our design supports dynamic RWSets, unlike alternative lock-free STMs. Dynamic RWSets are essential for a vector because operations such as `pushBack` use the vector's size to determine which index to write to, yet size is not known at the start of the transaction. Our array-based vector design avoids false conflicts by using direct-access traversal. For operations dependent on the size of the vector, any size changes will force the entire transaction to restart. To address this, we present a traversal tailored for the vector that uses optimistic traversal when operation indexes are known but switches to a pessimistic approach when an operation dependent on size is identified, preventing further size modifications and disallowing transaction restarts. We support two layouts, the segmented and compact vectors, both optimized to improve cache locality. The compact vector is performant but limits values to 32 bits in size. At the cost of general performance, the segmented vector supports values of arbitrary size and *versioned boxes* [1], containers that maintain a history of values, for improved read-only performance.

We propose a vector with the following contributions:

(1) The first lock-free transactional vector design: Existing lock-free vectors do not offer transaction properties, while general-purpose STM designs are either incompatible with the vector due to the use of a static RWSet or a lack of lock-free progress guarantees. To the best of our knowledge, there are no STM designs that offer both dynamic RWSets and lock-free progress. Furthermore, existing general-purpose transactional transformation approaches are either lock-based, prone to fragmentation, or incompatible with an array-based vector data structure.
(2) Vector-conscious transaction optimizations: We develop new vector-specific transaction optimizations, giving our implementation an edge over alternative, general-purpose transactional frameworks. Other transactional approaches have not explored these optimizations previously because the frameworks are general-purpose and would not typically benefit from them. Our contributions include the combination of optimistic and pessimistic traversal approaches based on the vector operations used and support for a dynamic RWSet while maintaining lock-freedom.
(3) Performance improvements over the state-of-the-art transactional vectors: We find that the compact vector is competitive with the most competitive alternative, an optimized TB vector, while maintaining lock-freedom. In the 66% read 33% write scenario, for instance, the compact vector is as much as 47% faster than TB and averages 18% faster.

2 BACKGROUND AND RELATED WORK

The background and related work discussed in this section covers topics in existing literature relevant to our lock-free transactional vector design, including lock-free vectors and transactional memory.

2.1 Background

Software Transactional Memory (STM) is a common class of transactional methodologies that uses a RWSet to track and handle conflicts. Each RWSet contains a list of reads and writes performed by a transaction. Writes that overlap with another transaction's RWSet causes the conflicting transaction to abort. STM can be used to easily convert a sequential data structure into its transactional equivalent. A *spurious abort* occurs when a transaction write overlaps with another transaction's RWSet even though a semantic conflict does not exist. If a transaction spuriously aborts, or an operation in the transaction fails, the transaction must perform a costly rollback and restart. A *rollback* reverses all operations that have completed in an aborted transaction. For STM, this involves a *physical rollback*, which undoes all writes in the RWSet by individually restoring the old values.

2.2 Lock-free Vectors

Several lock-free vector designs already exist, but none of them are transactional. For this reason, we do not directly compare the lock-free transactional vector to these approaches. Existing work on lock-free vectors establishes several useful approaches that extend to our transactional implementation.

Dechev et al. designed the first lock-free vector [3]. In the interests of performance, this vector supports only a subset of the sequential STL vector's operations:

(1) `read`: reads a value at a specific index in the vector. Out of bounds reads fail.

(2) `write`: writes a value at a specified index in the vector. Just like `read` operations, out of bounds writes fail.
(3) `pushBack`: writes a pushed value to the end of the vector.
(4) `popBack`: removes and returns the value at the end of the vector.
(5) `size`: returns the number of values stored in the vector.
(6) `reserve`: increases the vector's physical capacity.

Our work fully supports this same subset of operations, for similar performance reasons. Their design uses a shared descriptor object to enable `pushBack` and `popBack` to modify multiple memory locations atomically as one operation. The authors relaxed the STL vector's contiguous memory requirement, instead using a bucket array to enable concurrent resizing. This same bucket design is used in our work as well, since fully contiguous arrays make resize operations computationally expensive. Walulya et al. [23] modified the lock-free vector by Dechev et al. [3] to use flat combining. Since the pending operations of all threads are combined, the flat combining technique violates the isolation property required for transactional execution. Operations from a transaction that later aborts may combine with other operations, invalidating them.

Feldman et al. implemented the first wait-free vector [5], based on Dechev et al.'s lock-free vector[3]. *Wait-freedom* guarantees that every operation will complete in a finite number of steps. While still supporting bucket arrays, the authors added support for contiguous arrays. Though contiguous arrays improve cache performance and are more flexible than buckets, they greatly increase the cost of resize operations, so we opted not to use it. The authors also implemented a high-performance bounds checking approach, where a value is reserved to represent an `UNSET` value. If a read or write accesses an element with this `UNSET` value, then the access is out of bounds, beyond the size value. The novelty of this approach is that size does not have to be accessed to perform bounds checking; only the element itself needs to be read. This optimization is key to our own transactional vector implementation and complements our work.

2.3 Transactional Memory

Many methodologies have been proposed that enable the development of transactional equivalents for a variety of data structures. We used these approaches to guide our own transaction implementation. The majority of existing work is lock-based and suffers from physical rollbacks.

STM, proposed by Shavit and Touitou [21], provides transactional memory in software and achieves lock-free progress by requiring a transaction to acquire ownership of the memory location to be accessed and forcing the transactions that are trying to acquire the same memory location to help the owner complete its transaction. *Obstruction-freedom* is a progress condition and guarantees that a thread executing in isolation will finish in a finite number of steps. Support for obstruction-free transactions operating on dynamic-sized data structures is presented by Herlihy et al. [14]. An obstruction-free word-based software transactional memory [7] and a lock-free object-based software transactional memory [7] are proposed that utilize the single-word CompareAndSwap (CAS) to enable designs that work on commodity CPUs. Marathe et al. [18] present an obstruction-free word-based STM that simplifies the contention-free execution path and relies on the complex metadata management only for ensuring forward progress. Fernandes et al. [6] propose a lock-free multi-version STM that enables commits to proceed concurrently during the validation phase.

All STM approaches are generally prone to false conflicts, but array-based data structures, including the vector, use direct-access traversal and thus do not experience false conflicts. This makes STM appear to be a viable choice for a transactional vector, but the variants of STM that are lock-free [6, 7, 21] only support a static RWSet. Though this would conventionally be called a static transaction [21], we call it a static RWSet because modern research defines a *static transaction* to be a transaction where all of its *operations*, rather than memory locations, are known in advance. To support a transactional vector, a static RWSet must support push and pop operations, which modify vector locations based on the size of the vector. This means that the actual access locations cannot be known in advance.

The *data set* is the set of all shared memory locations that are candidates for the RWSet [21]. For a static RWSet to support transactions containing push or pop, the entire data set must be included in the RWSet to accommodate every possible read or write location. This approach fixes the capacity of the vector and conflicts with all other transactions due to overlapping RWSets, effectively locking the entire vector and incurring substantial overhead. These are severe limitations for approaches that support only a static RWSet, particularly for the vector. To overcome these issues, some STMs [13, 14] support a *dynamic RWSet*, where the RWSet can vary at runtime. These designs support a variable capacity and add only modified locations to the RWSet. To the best of our knowledge, all existing STMs that support a dynamic RWSet are obstruction-free. In the presence of livelock, obstruction-freedom must degrade to sequential execution to guarantee progress. Thus, a stronger lock-free approach is desirable, without the limitations of a static RWSet.

Herlihy et al. developed Transactional Boosting (TB) [12] to reduce false conflicts in STM. Instead of detecting conflicts at the memory level, it functions at the operation level. TB is particularly useful in node-based structures such as linked-lists and trees that read many incidental elements during traversal, which results in false conflicts for commutative operations in traditional read/write conflict detection systems. This optimization has no impact on vectors or other direct-access data structures with no traversals, as these memory conflicts lead to equivalent true semantic conflicts.

Optimistic concurrency control (OCC) is an alternative to the pessimistic approach used by standard STM. Instead of locking and operating on data structure objects as they are reached, a lock-free traversal is performed instead. The traversal tracks memory locations that should not be changed before commit time, then locks and verifies that other transactions have not made conflicting modifications. The transaction writes the required changes if no conflicting modifications are made, then unlocks the memory locations. This type of optimization is common for state-of-the-art transaction implementations, but this implementation only extends blocking approaches. Hassan et al. added support for OCC in TB [10], but uses a semantic read and write set, rather than memory locations.

Our pre-processing approach converts a static list of transaction operations into a RWSet. Unlike OCC, our pre-processing does not perform locking and does not need a verification step, reducing the overall abort rate. Part of the reason for this is that optimistic STM performs reads tentatively before verification, meaning that writes to those locations invalidate the transaction and require a restart. Our design performs reads and writes after the entire RWSet has been created, conflicting only when the size value is modified.

Several libraries have unified various transactional transformations into a framework. Examples include Proust [4] and transactional data structure libraries (TDSL) [22]. These approaches inherit the same shortcomings as the frameworks they unify, including potential deadlock and physical rollback. Proust and TDSL are generalized to be applicable to arbitrary abstract data types, so they do not address design choices specific to the transactional vector, such as pre-processing and contiguous memory access.

Herman et al. created a specialized STM called STO [15]. Instead of considering transactions in terms of memory accesses like STM, they consider object accesses. STO works as a general purpose framework rather than a specific set of design decisions tailored for a vector. STO includes a transactional vector data type, but the commit protocol in STO uses a locking scheme to perform transactions, leading to possible complications including priority inversion, convoying, and deadlock.

Zhang et al. devised Lock-Free Transactional Transformation (LFTT) [24], which uses semantic conflict detection, but with several enhancements. LFTT uses *logical status* inference, an approach that uses transaction status information, rather than reading a single value, to determine a current value of an index, either an old, unmodified value, or a new, just-written value. With this design, transaction status changes logically roll back all partial modifications instantly when necessary, referred to as a *logical rollback*. The work also uses a *helping scheme* to enable transactions to help complete the execution of otherwise blocking transactions to maintain lock-freedom. Our vector uses a similar lock-free design, benefiting from logical rollbacks and thread helping. LFTT only works with node-based structures, making it unsuitable for an array-based vector implementation, but we were able to enhance it to work with index-based access. To accomplish this, we removed node logic associated with physical insertions and removals, replacing it with an underlying resizable bucket vector for direct access. We also use the segmented and compact vector approaches to place elements contiguously.

3 CHALLENGES

Our key challenges to create a high performance lock-free transactional vector are as follows: 1) preserving direct, contiguous access to the vector; 2) a vector-specific traversal to minimize aborts and reduce conflicts; 3) minimizing the cost of operation failure induced aborts; and 4) developing a class of transactions that can read values without the overhead of the helping scheme.

Our first challenge is to place elements contiguously in memory to maximize spatial locality, a key attribute of the vector. An *element* is an object located at some index in the data structure containing the information necessary to interpret a data value for its index. As examples, an element in a sequential vector contains just the value itself, while LFTT and our design instead store logical status information to infer the value on traversal. The *segmented vector* uses segments of elements to improve spatial locality. The *compact vector* reduces elements down to 128-bits in size so that they fit in the underlying vector itself, enabling lock-free atomic `CMPXCHG16B` instructions to update values. Our segmented vector and compact vector designs improve the vector's spatial locality. A detailed overview of these implementations can be found in Section 6. Our testing shows that the segmented vector is more versatile, while the compact vector is more performant.

To overcome the second challenge, minimizing aborts, we consider two existing traversal techniques. *Optimistic traversal* reads all locations that must be accessed in a lock-free manner, verifies that the values are unchanged, then performs changes. If any values are changed, the traversal must start over. *Pessimistic traversal* locks each location as it is accessed to ensure that they cannot change, modifies them, then unlocks. Neither of these general-purpose approaches are a good fit for the transactional vector. The size element is a special kind of element, responsible for keeping track of the current bounds of the vector. Because the size element changes often, the optimistic approach would lead to frequent rollbacks and restarts. Though pessimistic traversal prevents spurious aborts on size, the remaining values in the vector would be unnecessarily accessed during traversal as well, causing potential conflicts. It is beneficial to defer access for as long as possible to minimize transaction conflicts. Our custom design only accesses size pessimistically. The remaining values are only read and written after pre-processing. This novel design stems from knowledge that, outside of size, changing values in the vector should never result in an unresolvable conflict or aborted transaction.

Our third challenge is to minimize the cost of transaction aborts. Physical rollbacks make STM a poor choice for our design. While logical rollbacks are fast, aborted transactions still waste computation, as the operations that later roll back still took time to execute. Our pessimistic size access eliminates spurious aborts, meaning that only operation failures, specifically out-of-bounds accesses, cause aborts. We determine that accessing and modifying vector values from high to low index, an ordering made possible by an effectively static RWSet, ensures that any out-of-bounds access will be the first attempted vector modification. These early aborts rely on knowledge of the vector's bounds checking system and are not relevant in the general cases studied by existing transaction research.

Our fourth challenge is to design a way to read values without conflicting and thus without helping. To support this, we implemented versioned boxes, similar to existing works [1, 19, 20]. This approach keeps old object versions so that *read-only transactions*, transactions that only perform reads [1], can read those objects in a consistent state without conflicting or interfering with other operations. This means that our read-only transactions avoid the helping scheme while maintaining strict serializability and opacity. Only the modifications that occurred before the transaction started are acknowledged while all subsequent changes are ignored.

4 TRANSACTIONS

A transaction is constructed by providing a list of ordered operations. Transactions are represented via a transaction descriptor, which stores this list of operations. Our pre-processing, discussed in Section 5, converts the list of operations into a RWSet. These reads and writes are used to alter the shared memory data structure, and the transaction commits when every read and write in the set has completed.

Each transaction can be active, committed, or aborted. All transactions are marked as active on creation. *Active* transactions are transactions that are in progress and have not finished executing their operations, so whether they will succeed or fail is unknown. If a transaction conflicts with another active transaction, it must help the active transaction to guarantee lock-free progress. If a transaction is *committed*, then it has succeeded. If a transaction is *aborted*, it has failed. This status is critical for determining the logical status of vector indices modified by the transaction.

Each index of the vector has an old value, a new value, and a transaction descriptor pointer associated with it. The transaction descriptor pointer always points to the last transaction to affect the element. To determine whether to read the old value or the new value, logical status inference is used.

5 PRE-PROCESSING AND EXECUTION

Our vector design only supports static transactions, but this allows transaction operations to be pre-processed. Our implementation uses pre-processing to avoid cyclic dependencies, reduce vector modification complexity, and minimize time spent modifying shared memory. During pre-processing, a static list of transaction operations is converted into a RWSet. To do this, all `reserve` calls in the transaction are consolidated to the largest reservation size requested. All operations in a transaction are evaluated in-order, converting `read`, `write`, `pushBack`, `popBack`, and `size` operations into simpler reads and writes. The list of reads and writes is maintained by a RWSet. The RWSet stores read/write operations (RWOps), each of which are used to describe read and write attempts on an element.

Each RWOp is associated with a specific index in the vector. If an operation is accessing an index for the first time in the transaction, it is added to the RWSet. If another operation previously accessed that index, the existing RWOp is modified as needed and automatically resolves conflicts that exist between operations in the same transaction. For instance, no matter how many writes occur in a transaction on the same location, we only perform the transaction's last write to that location in shared memory. Likewise, reads only need to get a value from shared memory if they occur before a write to the same location. Otherwise, they return the value written by the earlier write operation. This minimizes the amount of time spent in shared memory. Not that, unlike other operations, a `reserve` cannot be rolled back. We consider this reasonable because changes to the physical capacity of the vector will not affect other transactions; all transactions implicitly reserve up to the capacity they require, regardless of the existing capacity, and `reserve` can never reduce the capacity of the vector.

`pushBack` and `popBack`, as well as `read` and `write` operations that occur relative to size, are specialized, limited cases where a dynamic RWSet is required. To handle these cases properly, size is read and tentatively set up to write. As with any other element, size will be associated with the active transaction. Any other transactions attempting to modify size must first help the active transaction complete. Since the helping scheme is maintained, lock-freedom

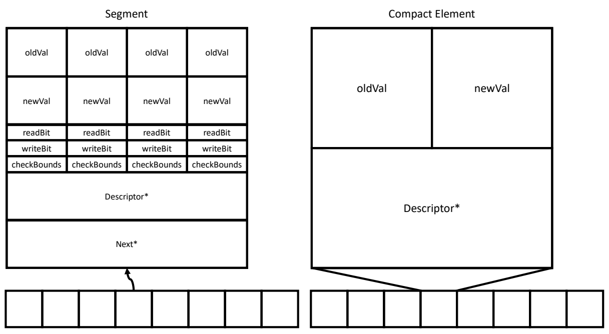

Figure 1: Layout of segments and compact elements. The segment contains data on a fixed number of contiguous elements, four in this example. `readBit`, `writeBit`, and `checkBounds` bits are used to infer the logical status of elements on traversal. The next pointer references previous layers associated with the segment. While the segments use atomic pointers to update, the compact vector's small element size allows placement in the underlying array itself. The compact vector does not use the status bits, instead checking the transaction descriptor's RWSet to perform logical status inference.

is preserved. The difference here is that size must be acquired before pre-processing completes and before we know the new value to write to size. To minimize the impact of this, we only acquire size when the first of these size-dependent operations are called. If the transaction does not contain any size-dependent operations, then the transaction will use a static RWSet. In either case, size is effectively fixed to make the operations deterministic and to prevent rollbacks. This approach, inspired by pessimistic traversal, is essential for determining which operations affect which indexes and thus identifying and handling internal conflicts.

After pre-processing, our approach executes the transactions, performing modifications to the vector in shared-memory. Modifications are always able to occur in a fixed order because an effectively static RWSet was generated during pre-processing. This resolves the cyclic dependency issues present in LFTT [24]. LFTT performs each operation in order, modifying the shared structure for each operation. Operation order is different for each transaction, leading to potential cyclic dependencies in LFTT. Contrarily, we pre-processes each operation in order, but, with the exception of size, we do this without affecting the shared structure. Once pre-processing has identified all reads and writes, we can perform those reads and writes in any order. If the order is consistent across all transactions, then a cyclic dependency can never occur. We choose to adjust the size first when needed, then access elements from high to low index. This order is chosen as an optimization for our bounds checking system to abort as early as possible.

6 VECTOR DESIGN

Because the lock-free transactional vector is inspired in part by LFTT, we can naively implement the vector by using an array of

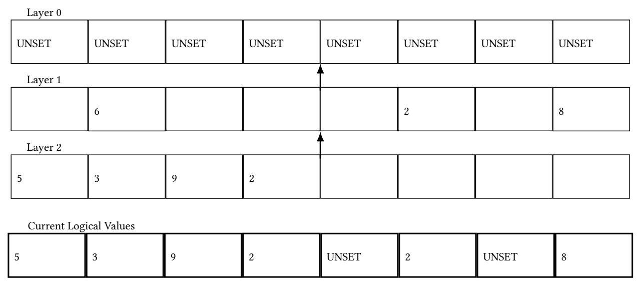

Figure 2: Delta Update Layers - Layer 0 is the initial state of the segment. Layer 1 is prepended to the list, writing new values to indexes 1,5, and 7. Layer 2 further updates the values, notably overwriting the 6 stored by transaction 1. The current logical values can be determined if the layers are stacked on top of each other, with the newest values visible.

atomic pointers, where the pointers reference vector elements. This approach is straightforward because each element can be updated atomically with a CAS instruction, but it also suffers from memory fragmentation. While the pointer array is contiguous, each element is dynamically allocated and has no guarantees of adjacent placement. Discontiguous elements forfeit spatial locality guarantees, reducing performance. In this paper, we have developed the segmented vector and compact vector to overcome these weaknesses. The segmented vector groups together multiple elements per object pointer. The compact vector uses a double-width CAS to atomically replace the entire element in the array itself, rather than a pointer to an object. While the compact vector offers improved spatial locality and a simpler design than the segmented vector, it requires a specialized instruction to support double-width atomics and, because elements are limited to 128-bits, is limited to storing values up to 32-bits in size. Meanwhile, the segmented vector supports elements of arbitrary size. The compact vector should be used in cases where these constraints are acceptable, such as the storage of integers, Booleans, or characters, or where compressed, 32-bit pointers can be used. Otherwise, the segmented vector with a segment size that maximizes cache utilization is suitable, especially in cases with read-only transactions.

Both approaches store logical status information to determine the value to read at an index. The new value is only read if the last transaction to interact with the element wrote to that location and committed. If the transaction is only attempting a read or the transaction aborted, then the old value is read. If the transaction is active, then helping must occur to enable transactional synchronization. Both approaches perform physical resizing by allocating buckets from low to high index and swapping them into shared memory atomically. Disallowing capacity reductions ensures previous buckets are never removed, enabling a lock-free approach.

6.1 Segmented Vector

We implement the segmented vector, an approach that uses segments of elements. More specifically, the underlying array points to segments rather than containing the elements themselves. This design was inspired by the Bw-tree designed by Levandoski et al. [17]. They used *delta updates*, a loan-term from version control systems to express the ability to update only a subset of elements. An example segment is illustrated in the left half of Figure 1. This layout is used to improve spatial locality by keeping ranges of elements adjacent in memory. Updates applied to a segment are called *layers*. Each layer may only update a subset of the elements in a segment, but every element in a layer is associated with the same transaction. Layers are traversed from newest to oldest until the latest logical value is retrieved. A high-level example of layer traversal can be seen in Figure 2. Delta updates minimize cache invalidation caused by object updates, keeping the existing objects unchanged and prepending new objects instead. Because layers store several values and their auxiliary data, they are large enough that they can only be updated atomically by exchanging pointers.

The segmented vector is optimized around the assumption that most operations in a given transaction will affect a range of elements rather than random accesses. A transaction that modifies multiple elements in a single segment can group them all together in a single layer and insert them using one CAS. In this way, each segment layer is associated with a single transaction, allowing that layer's logical status to be quickly resolved for all of its elements.

We implemented several optimizations to improve overall vector performance. For space optimization, layers can be configured to only contain space for elements that are actually modified. For simplicity, every layer object is a fixed size in our implementation. This approach also reduces compute time, as elements have direct array access instead of needing to compute an offset. Once all old values have been identified in a segment, if the subsequent CAS fails, then other threads have added one or more layers. To reduce redundant work, the re-traversal of layers only needs to consider the newly-added layers, maintaining the unchanged values acquired from previous traversals.

6.2 Compact Vector

To maintain lock-freedom and prevent race conditions, it is critical to ensure that objects representing elements are placed atomically in shared memory. Typical approaches can only modify word-sized objects atomically. Anything larger risks being performed by separate instructions, meaning that thread preemption could leave only part of the object modified. In LFTT [24] and the segmented vector, this is handled by using an atomic pointer.

The compact vector instead overcomes this by limiting elements to 16 bytes (128-bits) in size, enabling the use of the `CMPXCHG16B` instruction. `CMPXCHG16B` is a special hardware instruction supported by 64-bit architectures such as x86_64 to CAS 16 bytes atomically. 16 bytes is enough space to hold a 32-bit new value, a 32-bit old value, and a 64-bit transaction pointer referencing all supplemental information. Because these elements are small enough to be modified atomically, atomic pointers are unnecessary. Instead, the element itself is stored directly in the vector and exchanged atomically. An example element is illustrated in the right half of Figure 1. This layout makes the compact vector contiguous, though the underlying bucket array will still prevent fully contiguous element placement.

Algorithm 1 Type Definitions

```
 1: UNSET = INT_MAX
 2: enum TxStatus
 3:     active
 4:     committed
 5:     aborted
 6: struct CompactElement
 7:     int $oldVal$
 8:     int $newVal$
 9:     Desc *$descriptor$
10: struct Operation
11:     OpType $type$
12:     VAL $val$
13:     size_t $index$
14:     VAL $ret$
15: struct Desc
16:     size_t $size$
17:     Operation $ops[]$
18:     atomic<TxStatus> $status$
19:     atomic<map<size_t,
            RWOperation *>> $set$
20: struct RWOperation
21:     bool $checkBounds$
22:     Operation $lastWriteOp$
23:     vector<Operation *>
            $readList$
```

Algorithm 2 Update Element

```
 1: function UpdateElem($index$, $newElem$)
 2:     do
 3:         // Attempt to read an element.
 4:         if !$array$.read($index$, $oldElem$) then
 5:             // Failure means the vector wasn't allocated to this point.
 6:             $newElem.descriptor.status = aborted$
 7:             return $false$
 8:         // Quit if the transaction is no longer active.
 9:         // Can happen if another thread helped the transaction complete.
10:         if $newElem.descriptor.status \neq active$ then
11:             return $true$
12:         // Quit early if the element is already inserted. May occur during helping.
13:         if $oldDesc == newElem.descriptor$ then
14:             return $true$
15:         // Help the active transaction.
16:         while $oldDesc.status$ == active do
17:             completeTransaction($oldDesc$, $index$)
18:         // Get the new element for the old value.
19:         // We only get the new value if it was write committed.
20:         if $oldDesc.status$ == committed &&
                $oldDesc.set[index].lastWriteOp \neq$ NULL then
21:             $newElem.oldVal = oldElem.newVal$
22:         else// Else grab the old page's old value.
23:             $newElem.oldVal = oldElem.oldVal$
24:         // Abort if the operation fails our bounds check.
25:         $op = newElem.descriptor.set[index]$
26:         if $op.checkBounds == yes$&& $newElem.oldVal$ == UNSET then
27:             $newElem.descriptor.status = aborted$
28:             return $false$
29:     while !$array$.tryWrite($index, oldVal, newVal$)
30:     // Store the old value in the associated operations.
31:     $op = newElem.descriptor.set[index]$
32:     // For each operation attempting to read the element.
33:     for $i = 0$, $i < op.readList$.size(), $i$++ do
34:         // Assign the return values.
35:         $op.readList[i].ret = newElem.oldVal$
36:     return $true$
```

7 ELEMENT UPDATE

Before discussing our update algorithm, an explanation of the structs and enums in Algorithm 1 is required. UNSET on Line 1 is a reserved element value used in bounds checking. In this case, UNSET is the maximum value that can be represented by a 32-bit integer. Line 2 lists the three different transaction statuses: active, committed, or aborted.

The compact element on Line 6 contains an old value, a new value, and a transaction descriptor pointer. Whether the old or new value is read depends on the logical status of the element's associated transaction. The transaction pointer points to the last transaction that affected the element. Note that, because the compact element is size-constrained, we have used an int to represent the old and new values. If int is 32-bit, pointers are 64-bit, and our double-width CAS supports 128-bits, then the compact element can be exchanged atomically.

Each transaction consists of ordered operations, shown on Line 10. Each operation has a type, which determines which of the other fields are used. The *val* field stores the new element used in write and pushBack operations. popBack implicitly stores UNSET in *val* to preserve bounds-checking. The *index* field is used for absolute operations like read and write to determine which index to modify. It is also used in reserve calls to indicate the desired capacity. The *ret* field stores the return value for the operation, which is essential for read and popBack operations. It is used locally to access return values after a transaction commits.

Desc on Line 15 is a transaction descriptor. *size* stores the number of operations. *ops* is the array of the operations themselves, created and passed in by the user. *status* stores the current logical status of the transaction and is always active on transaction start. *set* is used during pre-processing to keep track of all relevant RWSet data. RWOperation keeps track of the latest operation that performed a write and all operations attempting to read at a target index.

7.1 Update

Algorithm 2 shows the approach used to replace elements in the compact vector. The segmented vector uses the same core approach, but with additional considerations taken to accommodate updating multiple elements in one CAS. First, the existing element in the index is read on Line 4. If the element cannot be read, it means that the vector's capacity is lower than the target index, so the transaction aborts. If helping occurs, another thread could have completed the transaction, so we check the new element's status on Line 10 so the update can end early. To maintain lock-freedom, the update may have to help another conflicting transaction complete by calling completeTransaction on Line 17. We infer the logical status of the old element on Line 20. If the operation being performed requires bounds checking, we check to ensure the old value was not UNSET; if it is, then the transaction aborts, since it attempted an out-of-bounds read. At the end of the loop, we perform as CAS to replace the old element on Line 29. On a successful CAS, all operations attempting to read the old element's value are assigned their return values on Line 35.

8 VERSIONED BOXES

If a transaction only performs read operations, then a designer can use versioned boxes [1] to perform read-only transactions that do not conflict with other transactions. Versioned boxes preserve strict serializability and opacity while avoiding the helping scheme for performance gains. The segmented vector is capable of doing this because, without consolidation, delta updates inherently function as versioned boxes. Versioned boxes work by preserving the history of values in a linked list of updates, associating each update with a version counter to maintain proper ordering. This allows reads to serialize at the beginning of the transaction, reading the proper values for that point in time by traversing through the versioned

	Before	After
active	Ignore*	Ignore
committed	Read new/old	Ignore
aborted	Read old	Ignore

Figure 3: Read-only transaction behavior. Rows represent the target transaction's status and columns represent whether the transaction has a version number before or after the version number of the box's history. If a transaction is ignored, then the algorithm traverses to the next layer and evaluates the next transaction.

history and enforcing a proper fixed ordering. Read-only transactions do not write to shared memory, meaning they do not interfere with the execution of any other transactions. Note that the compact vector does not support versioned boxes because it does not use delta updates.

When a read in a read-only transaction occurs, there are several possible cases, illustrated by the cross table in Figure 3. To interpret this table, consider a layer being read. All modifications in a layer are associated with one transaction. The transaction's state is either active, committed, or aborted. The layer's transaction will come before or after the actively running read. A read-only transaction is only concerned with the status of the data structure before the time it accessed the version counter, so layers that began insertion after the transaction started will be ignored and the previous layer at the target location will be evaluated instead. If a layer was aborted and versioned before the read, then the update's old value is returned. If a layer was committed and versioned before the read and the layer performed a write, then the new value is returned. If a layer was committed and versioned before the read and the layer only performed a read, then the old value is returned. If the layer is active, it is ignored and the algorithm traverses to the next layer.

There is a rare edge case, denoted by the asterisk in the table, where an update is still active but has already been assigned a version number smaller than the associated read transaction. In this situation, we ignore the update and traverse to the next layer. The layer may later commit and become visible to subsequent reads in the same transaction. This would violate transaction isolation. To ensure transaction correctness, we implemented a local ignore set for each read-only transaction. All active transactions visited by the read-only transaction are added to its ignore set. Before reading any subsequent transactions, the read-only transaction will check to see if that transaction is in its ignore set. If the transaction is found in the set, it is ignored to maintain isolation.

9 PROGRESS CONDITIONS

Our transactional vector achieves lock-freedom. As originally defined by Herlihy [11], a concurrent object is *lock-free* if it guarantees that some thread will make progress in a finite number of steps. To reason about this property, consider i, the number of threads and j, the maximum number of index locations modified by a transaction. j is less than or equal to the number of operations in the transaction plus one to include potential size modifications. In the case of the segmented vector, additionally consider k, the length of the longest linked-list of layers. k can be reduced by performing periodic consolidation, but, in practice, up-to-date elements in a segment remain close to the head of the linked-list. There can be no more than i active transactions at a time, bounded by the number of threads. Execution of operations in a transaction are bounded by j, but a transaction may have to help any conflicting active transactions. This helping is bounded by the number of active threads, i. Segmented vector traversals are grouped by segments, but, in the worst case, each element lookup is in its own segment and requires its own traversal. Thus, each modified location may have to perform as many as k dereferences to read and copy the old value during segmented vector traversal. In the worst case, there are i active threads, only 1 thread remains live, and $i - 1$ of them have stalled and conflict with the live thread. This places an upper bound of $i * j$ steps for the compact vector and $i * j * k$ for the segmented vector, since each memory location access could require as many as k linked-list dereferences to resolve.

10 PERFORMANCE TESTING AND RESULTS

We evaluated our approach against several transactional vectors. We implemented a sequential vector and converted it into a thread-safe transactional equivalent using the GCC STM library [8]. This library is built-in to GCC 7 and is implemented by marking off sections of code as transactions. It uses the conventional RWSet of STM to handle conflicts. We implemented our own sequential vector because the STL vector uses atomic operations on our tested architectures, which are considered transaction-unsafe by GCC STM.

We created a TB version of our vector. Note that the TB vector is functionally identical to a pessimistic object-based STM vector because the vector essentially uses direct access traversal. Thus, both approaches share an identical RWSet. Our implementation of the TB vector benefits from our assorted vector optimizations and stores elements directly in a bucket array, so the only functional difference between it and the compact vector is that the TB vector locks element locations while the compact vector is lock-free and uses helping to guarantee progress.

We also integrated STO into our testing framework. STO operates at object-level granularity. It is the only transactional vector we are aware of that was implemented in related work. Unlike our design, it does not offer lock-free progress guarantees.

10.1 Testing Methodology

In our testing, memory was pre-allocated to ensure that performance of the structure itself was accurately tested, without allocation overhead. Likewise, our tests do not perform garbage collection to minimize its impact on performance. All tests were performed across a testbed of four different systems: a 24 core 48 thread Intel Xeon Platinum 8160 @ 2.1 GHz (INTEL), a 32 core 64 thread AMD EPYC 7501 @ 2 GHz (AMD), a NUMA 64 core 4 CPU AMD Opteron 6272 @ 2.1 GHz (NUMA), and a 48 core Cavium ARMv8 ThunderX CP @ 2.0 GHz (ARM). The INTEL and AMD systems were running Ubuntu 18.04 LTS while the ARM and NUMA systems were running Ubuntu 16.04.6 LTS. All systems ran the GCC STM vector (STMVE), compact vector (COMPA), segmented vector (SEGME), STO vector (STOVE), and TB vector (BOOST), with all vectors built using GCC 7.4.0. Our tests consisted of 10,000 transactions per case, with the number of transactions being split evenly between threads. Excess transactions that were not evenly divisible into the number of threads were discarded to ensure an even workload for each thread. Before measuring timing, every vector tested was pre-filled with 10,000 random push operations, split evenly between threads, to ensure reads and writes could succeed. Just like the tested transactions, excess pre-fill transactions were discarded. Each transaction in our testing contains 5 operations per transaction. Varying the transaction size resulted in similar performance differences between data structures, so these results are omitted for space. Values for all data structures were random unsigned 32-bit integers to allow for direct comparison with the compact vector. The segmented vector, compact vector, and TB vector all use the same lock-free reserve operation. STO was pre-reserved since it does not have a transactionally-safe reserve operation. The STM vector resizes sequentially by allocating an array of double the previous size and copying over values. Reads and writes were performed over an index range of 10,000. We performed each test 5 times and averaged the results to minimize the effects of run-to-run variance.

The segmented vector was designed to improve cache utilization while still using pointers to reference objects. When determining an appropriate segment size, we considered the behavior of cache coherency systems, which operate at the granularity of cache lines. Because of this, multiple threads may compete for the same cache line even if they access unique data. This issue is called *false sharing*, where two or more threads access different data that exists on the same cache line. To avoid false sharing, our segmented vector uses 64 bytes to store values in our testing scenarios. This is the cache line size on all four architectures tested and, with 32-bit values, is enough to store 8 elements per segment.

Versioned boxes are a unique benefit for the segmented vector, enabling reads to execute without conflicting. To measure the overhead associated with the associated version counting, we ran all of our tests both with version counting enabled and disabled. In our testing, we found that the overhead required to support read-only transactions is negligible, being indistinguishable from run-to-run variance. Because runtime overhead is minimal and strict serializability and opacity are still maintained, there is no reason to use traditional reads if a transaction could use read-only transactions instead. Thus, we decided to automatically identify transactions that exclusively contain reads and automatically run them using versioned box histories.

Our implementation of the TB vector uses the same optimizations as the segmented and compact vectors. Both approaches store elements contiguously in memory and use the same underlying bucket array. While the compact vector maintains lock-freedom, the TB vector supports elements of arbitrary size. Our fixed, high to low access order makes aborts occur nearly instantly, minimizing the cost of the TB vector's major weakness, expensive physical rollbacks. The lock-based nature of TB suggests that it would perform worse in thread-constrained scenarios, where the operating system is prone to preempting threads that hold locks, or high-contention situations, where RWSet conflicts are commonplace.

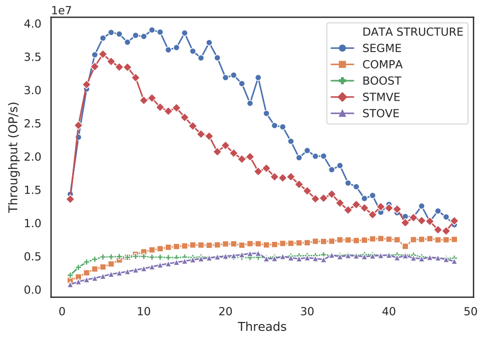

Figure 4: INTEL, 100% Read, Random Indexes

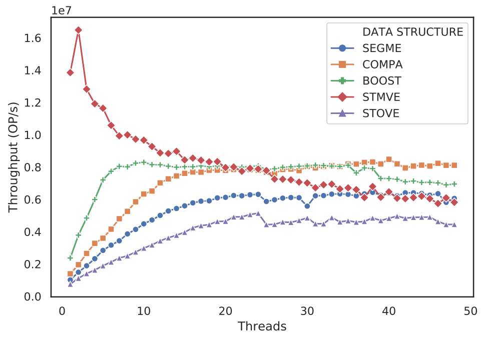

Figure 5: INTEL, 100% Write, Random Indexes

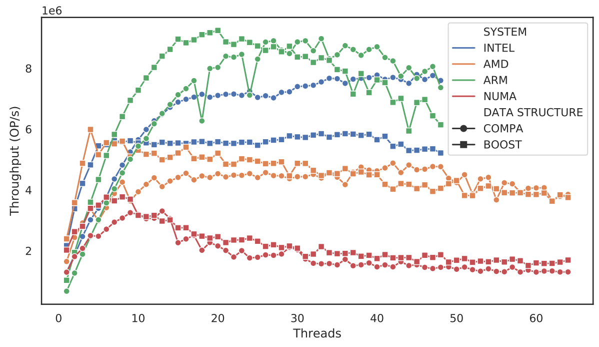

Figure 6: Compact and TB Vector, Architectural Comparison, 66% Read - 33% Write

10.2 Results and Discussion

Read-heavy scenarios are an ideal case for both GCC STM and the segmented vector, with the segmented vector offering the greatest throughput in the 100% read case, shown in Figure 4. This occurs because both GCC STM and the segmented vector traverse optimistically and never generate conflicts on read-only transactions. GCC STM never encounters RWSet conflicts because no writes are performed, while the segmented vector takes full advantage of multi-versioning to quickly read values without conflicts. Note that both GCC STM and the segmented vector's performance starts to degrade with more than 5 threads while the alternatives maintain or improve their throughput with increasing thread counts. Outside of GCC STM and the segmented vector, the compact vector outperforms all other tested data structures in the 100% read case when over 10 threads are used. In the read-only case on INTEL, our segmented vector design is as much as 80% faster than the state-of-the-art GCC STM design and averages 28% faster.

We evaluated 100% write performance in Figure 5. When only writes are performed, GCC STM offers superior throughput compared to the alternatives up to around 10 threads, but write conflicts in the RWSet become increasingly likely at higher thread counts, causing a degradation in performance. STO and the segmented vector offer increasing throughput at scale, but are uncompetitive with TB and the compact vector. TB generally offers the best write performance when compared to the state-of-the-art alternatives, and the compact vector offers competitive write performance with TB while additionally maintaining lock-free guarantees. In the write-only case on INTEL, our compact vector is as much as 17% faster than the state-of-the-art TB design.

Figure 6 offers an architectural comparison of the existing state-of-the-art TB approach and our most performant new design, the compact vector. In this example, a mix of 66% reads and 33% writes are performed. In the figure, each color is associated with an architecture and each shape is associated with a data structure. On INTEL, we find that our compact vector outperforms TB above 10 threads, being as much as 47% faster than TB at 45 threads. Our AMD machine shows TB outperforming the compact vector up to the 32 physical threads of the machine. Above 32 threads, where simultaneous multithreading is used, the compact vector is competitive with TB. Our NUMA machine does not scale well beyond 16 threads because each physical CPU die has 16 cores each. All of our machines running x86_64 support 16 byte atomic operations. The ARM machine only supports up to 8 byte atomic objects, forcing the compact vector, which relies on 16 byte atomic operations, to fall back to lock-based atomic CPU instructions. We verified this behavior using the IS_LOCK_FREE function from the C++ std::atomic library. Though lock-based atomic CPU instructions are being used by the compact vector on ARM, it remains competitive with the TB design. Our design is no longer lock-free on ARM, but it illustrates the portability of our approach.

11 CONCLUSIONS AND FUTURE WORK

In this paper, we presented designs for the first lock-free transactional vector. We explored the various design components used to ensure correctness and optimize performance. We found that our lock-free transactional vector was competitive with the leading designs, the TB vector and the GCC STM vector. In the read-only case, our segmented vector design is as much as 80% faster than the state-of-the-art GCC STM design. In the write-only case, our compact vector is as much as 17% faster than the state-of-the-art TB design.

In future work, we intend to focus on further minimizing the bottlenecks associated with vector size, perhaps relaxing some of our constraints to accomplish this. We are also interested in exploring transactional vector designs supporting modern dynamic transactions [16], where the list of operations is not known in advance and varies at runtime.

REFERENCES

[1] João Cachopo and António Rito-Silva. 2006. Versioned boxes as the basis for memory transactions. *Science of Computer Programming* 63, 2 (2006), 172 – 185. https://doi.org/10.1016/j.scico.2006.05.009 Special issue on synchronization and concurrency in object-oriented languages.
[2] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. 2008. Software Transactional Memory: Why Is It Only a Research Toy? *Queue* 6, 5, Article 40 (Sept. 2008), 13 pages. https://doi.org/10.1145/1454456.1454466
[3] Damian Dechev, Peter Pirkelbauer, and Bjarne Stroustrup. 2006. Lock-free Dynamically Resizable Arrays. In *Proceedings of the 10th International Conference on Principles of Distributed Systems (OPODIS'06)*. Springer-Verlag, Berlin, Heidelberg, 142–156. https://doi.org/10.1007/11945529_11
[4] Thomas D. Dickerson, Paul Gazzillo, Eric Koskinen, and Maurice Herlihy. 2017. Proust: A Design Space for Highly-Concurrent Transactional Data Structures. *CoRR* abs/1702.04866 (2017). arXiv:1702.04866 http://arxiv.org/abs/1702.04866
[5] S. Feldman, C. Valera-Leon, and D. Dechev. 2016. An Efficient Wait-Free Vector. *IEEE Transactions on Parallel and Distributed Systems* 27, 3 (March 2016), 654–667. https://doi.org/10.1109/TPDS.2015.2417887
[6] Sérgio Miguel Fernandes and João Cachopo. 2011. Lock-free and Scalable Multi-version Software Transactional Memory. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP '11)*. ACM, New York, NY, USA, 179–188. https://doi.org/10.1145/1941553.1941579
[7] Keir Fraser and Tim Harris. 2007. Concurrent Programming Without Locks. *ACM Trans. Comput. Syst.* 25, 2, Article 5 (May 2007). https://doi.org/10.1145/1233307.1233309
[8] GCC Wiki contributors. 2012. TransactionalMemory - GCC Wiki. https://gcc.gnu.org/wiki/TransactionalMemory. [Online; accessed 17-June-2019].
[9] Rachid Guerraoui and Michal Kapalka. 2008. On the Correctness of Transactional Memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '08)*. ACM, New York, NY, USA, 175–184. https://doi.org/10.1145/1345206.1345233
[10] Ahmed Hassan, Roberto Palmieri, Sebastiano Peluso, and Binoy Ravindran. 2017. Optimistic Transactional Boosting. *IEEE Trans. Parallel Distrib. Syst.* 28 (2017), 3600–3614.
[11] Maurice Herlihy. 1993. A Methodology for Implementing Highly Concurrent Data Objects. *ACM Trans. Program. Lang. Syst.* 15, 5 (Nov. 1993), 745–770. https://doi.org/10.1145/161468.161469
[12] Maurice Herlihy and Eric Koskinen. 2008. Transactional Boosting: A Methodology for Highly-concurrent Transactional Objects. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '08)*. ACM, New York, NY, USA, 207–216. https://doi.org/10.1145/1345206.1345237
[13] Maurice Herlihy, Victor Luchangco, and Mark Moir. 2006. A Flexible Framework for Implementing Software Transactional Memory. *SIGPLAN Not.* 41, 10 (Oct. 2006), 253–262. https://doi.org/10.1145/1167515.1167495
[14] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. 2003. Software Transactional Memory for Dynamic-sized Data Structures. In *Proceedings of the Twenty-second Annual Symposium on Principles of Distributed Computing (PODC '03)*. ACM, New York, NY, USA, 92–101. https://doi.org/10.1145/872035.872048
[15] Nathaniel Herman, Jeevana Priya Inala, Yihe Huang, Lillian Tsai, Eddie Kohler, Barbara Liskov, and Liuba Shrira. 2016. Type-aware Transactions for Faster Concurrent Code. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*. ACM, New York, NY, USA, Article 31, 16 pages. https://doi.org/10.1145/2901318.2901348
[16] Pierre LaBorde, Lance Lebanoff, Christina Peterson, Deli Zhang, and Damian Dechev. 2019. Wait-free Dynamic Transactions for Linked Data Structures. In *Proceedings of the 10th International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM'19)*. ACM, New York, NY, USA, 41–50. https://doi.org/10.1145/3303084.3309491
[17] J. J. Levandoski, D. B. Lomet, and S. Sengupta. 2013. The Bw-Tree: A B-tree for new hardware platforms. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 302–313. https://doi.org/10.1109/ICDE.2013.6544834
[18] Virendra Jayant Marathe and Mark Moir. 2008. Toward High Performance Nonblocking Software Transactional Memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '08)*. ACM, New York, NY, USA, 227–236. https://doi.org/10.1145/1345206.1345240
[19] Dmitri Perelman, Anton Byshevsky, Oleg Litmanovich, and Idit Keidar. 2011. SMV: Selective Multi-Versioning STM. In *Distributed Computing*, David Peleg (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 125–140.
[20] Torvald Riegel, Pascal Felber, and Christof Fetzer. 2006. A Lazy Snapshot Algorithm with Eager Validation. In *Distributed Computing*, Shlomi Dolev (Ed.).

Springer Berlin Heidelberg, Berlin, Heidelberg, 284–298.

[21] Nir Shavit and Dan Touitou. 1997. Software transactional memory. *Distributed Computing* 10, 2 (01 Feb 1997), 99–116. https://doi.org/10.1007/s004460050028

[22] Alexander Spiegelman, Guy Golan-Gueta, and Idit Keidar. 2016. Transactional Data Structure Libraries. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 682–696. https://doi.org/10.1145/2908080.2908112

[23] I. Walulya and P. Tsigas. 2017. Scalable Lock-Free Vector with Combining. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 917–926. https://doi.org/10.1109/IPDPS.2017.73

[24] Deli Zhang, Pierre Laborde, Lance Lebanoff, and Damian Dechev. 2018. Lock-Free Transactional Transformation for Linked Data Structures. *ACM Trans. Parallel Comput.* 5, 1, Article 6 (June 2018), 37 pages. https://doi.org/10.1145/3209690