Towards Automatic Lock Removal
for Scalable Synchronization

Maya Arbel*†   Guy Golan Gueta*   Eshcar Hillel*   Idit Keidar*†

*Yahoo Labs, Haifa, Israel   †The Technion, Haifa, Israel

Abstract

We present a *code transformation* for concurrent data structures, which increases their scalability without sacrificing correctness. Our transformation takes lock-based code and replaces some of the locking steps therein with optimistic synchronization in order to reduce contention. The main idea is to have each operation perform an optimistic traversal of the data structure as long as no shared memory locations are updated, and then proceed with pessimistic code. The transformed code inherits essential properties of the original one, including linearizability, serializability, and deadlock freedom.

Our work complements existing pessimistic transformations that make sequential code thread-safe by adding locks. In essence, we provide a way to optimize such transformations by reducing synchronization bottlenecks (for example, locking the root of a tree). The resulting code scales well and significantly outperforms pessimistic approaches. We further compare our synthesized code to state-of-the-art data structures implemented by experts. We find that its performance is comparable to that achieved by the custom-tailored implementations. Our work thus shows the promise that automated approaches bear for overcoming the difficulty involved in manually hand-crafting concurrent data structures.

1 Introduction

The steady increase in the number of cores in today's computers is driving software developers to allow more and more parallelism. An important focal point for such efforts is scaling the concurrency of shared data structures, which are often a principal friction point among threads. Many recent works have been dedicated to developing scalable concurrent data structures (e.g., [8, 19, 40, 11, 14, 9, 5, 21, 10, 31, 24, 38]), some of which are widely used in real-world systems [44].

Each of these projects generally focuses on a single data structure (e.g., a binary search tree [10] or a queue [38]) and manually optimizes its implementation. These data structures are developed by concurrency experts, typically PhDs or PhD candidates, and proving their correctness is painstaking; for example, the proofs of [9, 21] are 31 and 20 pages long, respectively. The rationale behind dedicating so much effort to one data structure is that it is generic and can be used by many applications. Nevertheless, systems often use data structures in unique ways that necessitate changing or extending their code (e.g., [2, 3, 43, 46]), in which cases custom-tailored implementations may not meet the requirements. Here, we propose an approach to facilitate this labor-intensive process, making scalable synchronization more readily available.

Specifically, we present in Section 2 an algorithm for a source-to-source code transformation that takes a lock-based concurrent data structure implementation as its input and generates more scalable code for the same data structure via judicious use of optimism. Our approach combines optimism and pessimism in a practical way. Like some previous hand-crafted solutions [30], we exploit the common access pattern in data structure operations, (e.g., tree insertion or deletion), which typically begin by traversing the data structure (to the insertion or deletion point), and then perform local updates at that location. Our solution replaces locks in the initial read-only traversal with optimistic synchronization, and performs updates using the original lock-based code. It may thus be seen as a form of software

lock elision for read-only prefixes of operations (transactions). Combining optimism and pessimism allows us to achieve "the best of both worlds" – while the optimistic traversal increases concurrency and eliminates bottlenecks, the use of pessimistic updates saves the overhead associated with speculative or deferred shared memory updates, (as occurs in *software transactional memory (STM)* [29]).

We show in Section 3 that our transformation preserves the external behavior (e.g., linearizability, serializability, and deadlock-freedom) of the original lock-based code; formal proofs appear in Appendix A. Moreover, our transformation preserves *disjoint access parallelism* [33], (the property that threads that access disjoint data objects do not contend on low level shared memory locations), as it refrains from introducing a shared global clock (as some STM systems do [45]) or other sources of contention.

One important use case for our transformation is to apply it in conjunction with automatic lock-based parallelization mechanisms [25, 36]. The latter instrument sequential code and add fine-grained locks that ensure its safety in concurrent executions. Our evaluation shows that, by themselves, solutions of this sort may scale poorly. This is due to synchronization bottlenecks, e.g., the root of a tree, which is locked by all operations. By subsequently applying our transformation, one can optimize the lock-based code they produce, yielding an end-to-end approach to scalable parallelization of sequential code.

In Section 4 we evaluate our transformation by generating an unbalanced search tree and a treap (randomized balanced search tree). We synthesize these data structures from *sequential implementations* by applying first the algorithm of [25] (*domination locking*) to create lock-based code, and then our transformation. We evaluate the scalability of the resulting code in a range of workload scenarios on a 32-core machine. In all cases, the lock-based implementations do not scale – their throughput remains flat as the number of running threads increases. In contrast, the code generated by our transformation is scalable, and its throughput continues to grow with the number of threads. We further use the Synchrobench framework [26] to compare our synthesized code to data structures that were recently hand-crafted by experts in the field [19, 10, 1, 21, 14], as well as a state-of-the-art STM [45]. Our results show that the implementations we have generated perform comparably to custom-tailored solutions.

The advantage of our approach is in its *generic* nature, which allows us to parallelize existing code without requiring experts to perform manual optimizations. Other generic approaches we are familiar with are domination locking [25] and STM [45], both of which perform worse than our transformed code in our experiments. Further discussion of related work appears in Section 5.

To conclude, this paper demonstrates that generic synchronization, based on a careful combination of optimism and pessimism, is a promising approach for bringing legacy code to emerging computer architectures. While this paper illustrates the method for tree data structures, we believe that the general direction is more broadly applicable, and maybe used with a variety of locking schemes, such as two phase locking. Section 6 concludes the paper and touches on some directions for future work.

2 Transformation

We present an algorithm for a source-to-source transformation, whose goal is to optimize the code of a given data structure implemented using lock-based concurrency control. In Section 2.1, we detail our assumptions about the given code and the locks it uses. Section 2.2 overviews our general approach to combining optimism and pessimism, while Section 2.3 details how the code is instrumented.

2.1 Lock-based Data Structures

A *data structure* defines a set of *operations* that may be invoked by clients of the data structure, potentially concurrently. Operations have parameters and local (private) variables. The operations interact via *shared memory variables*, which are also called *shared objects*. Each shared object supports atomic *read* (load) and *write* (store) instructions. More formal definitions appear in Appendix A.

In addition, each shared object is associated with a lock, which can be unique to the object or common to several (or even all) objects. The object supports atomic *lock* and *unlock* instructions. Locks are exclusive (i.e., a lock can be held by at most one thread at a time), and blocking. We assume

that in the given code every (read or write) access by an operation to a shared object is performed when the executing thread holds the lock associated with that object.

The given code only uses the *lock* and *unlock* instructions, while the transformed code can apply in addition atomic non-blocking *tryLock* and *isLockedByAnother* instructions: *tryLock* returns *false* if the lock is currently held by another thread, otherwise it acquires the lock and returns *true*; *isLockedByAnother* returns *true* if and only if the lock is currently held by another thread.

2.2 Combining Optimism and Pessimism

Optimistic concurrency control is a form of synchronization, which accesses shared variables without using locks in the hope that they will not be modified by others before the end of the operation (or more generally, the transaction). To verify the latter, optimistic concurrency control relies on *validation*, which is typically implemented using *version numbers*. If validation fails, the operation restarts. Optimistic execution of update operations requires either performing roll-back (reverting variables to their old values) upon validation failure, or deferring writes to commit time; both approaches induce significant overhead [12]. We therefore refrain from speculative shared memory updates.

The main idea behind our approach is to judiciously use optimistic synchronization only as long as an operation does not update shared state; we use a standard approach based on version numbers to allow validation of optimistic reads. Once an operation writes to shared memory, we revert to pessimistic (lock-based) synchronization. In other words, we rely on validation in order to render redundant locks that would have been acquired and freed before the first update. This scheme is particularly suitable for data structures, since the common behavior of their operations is to first traverse the data structure, and then perform modifications. Since the read only prefix has no side effects we can sandbox it by catching exceptions and infinite loops, and defer validation to the end of the traversal.

Conceptually, our approach thus divides an operation into three phases: an optimistic *read-only phase*, a pessimistic *update phase* and a *validation phase* that conjoins them. The read-only phase traverses the data structure without taking any locks, while maintaining in thread-local variables sufficient information to later ensure the correctness of the traversal. The read phase is *invisible* to other threads, as it updates no shared variables. The update phase uses the original pessimistic (lock-based) synchronization, with the addition of updating version numbers. The validation phase bridges between the optimistic and pessimistic ones. It first locks the objects for which a lock would have been held at this point by the original locking code, and then validates the correctness of the read-only phase. This allows the update phase to run as if an execution of the original pessimistic synchronization took place. If the validation fails, the operation restarts. In order to avoid livelock, we set a threshold on the number of restarts. If the threshold is exceeded, the code falls back on pessimistic execution. We show below that it is safe to do so, since our semi-optimistic code is compatible with the fully pessimistic one.

Phase Transition In many cases, the transition from the read-only phase to the update phase occurs at a statically-defined code location. For example, many data structure operations begin with a read-only traversal to locate the key of interest, and when it is found, proceed to execute code that modifies the data structure. This is the case in all the examples we consider in Section 4 below.

More generally, it is possible to switch from the optimistic read-only execution (via the validation phase) to pessimistic execution at any point before the first update. Moreover, the phase transition point can be determined dynamically at run time.

One possible way to dynamically track the execution mode is using a flag **opt**, initialized to true, indicating the optimistic phase. Every shared memory update operation is then instrumented with code that checks **opt**, and if it is true, executes the validation phase followed by setting **opt** to false and continuing the execution from the same location.

```
 1: FUNCTION addThird(List list, Node new)        1: FUNCTION addThird(List list, Node new)

    ----------              ▷ read-only phase        ----------                  ▷ read-only phase
 2:                                                 2:   lockedSet.init()
 3:                                                 3:   readSet.init()
 4:   list.lock()                                   4:   if !track(list) then goto 1
 5:   Node prev = list.head                         5:   Node prev = list.head
 6:   prev.lock()                                   6:   if !track(prev) then goto 1
 7:   list.unlock()                                 7:   lockedSet.remove(list)
 8:   Node succ = prev.next                         8:   Node succ = prev.next
 9:   succ.lock()                                   9:   if !track(succ) then goto 1
10:   prev.unlock()                                10:   lockedSet.remove(prev)
11:   prev = succ                                  11:   prev = succ
12:   succ = succ.next                             12:   succ = succ.next
13:   succ.lock()                                  13:   if !track(succ) then goto 1
    ----------                                        ----------                  ▷ validation phase
14:                                                14:   read fence
15:                                                15:   for all obj in lockedSet do
16:                                                16:     if !obj.tryLock() then
17:                                                17:       unlockAll()
18:                                                18:       goto 1
19:                                                19:   if !validateReadSet() then
20:                                                20:     unlockAll()
21:                                                21:     goto 1
    ----------              ▷ update phase            ----------                  ▷ update phase
22:   prev.next = new                              22:   prev.next = new
23:   new.lock()                                   23:   new.lock()
24:   new.next = succ                              24:   new.next = succ
25:                                                25:   prev.incVersion
26:   prev.unlock()                                26:   prev.unlock()
27:                                                27:   new.incVersion
28:   new.unlock()                                 28:   new.unlock()
29:                                                29:   succ.incVersion
30:   succ.unlock()                                30:   succ.unlock()
```

(a) Code with original locking   (b) The code produced by our transformation

Figure 1: Code transformation example. The synchronization code is in bold.

```
 1: FUNCTION track(obj)
 2:   lockedSet.add(obj)
 3:   long ver = obj.getVersion()
 4:   readSet.add(⟨obj,ver⟩)
 5:   if ⟨obj,v⟩ ∈readSet and v!=ver then return false
 6:   if obj.isLockedByAnother() then return false
 7:   return true
```

Figure 2: In read-only phase, locking is replaced by tracking locks and read objects' versions.

```
 1: FUNCTION validateReadSet()
 2:   for all ⟨obj,ver⟩ in readSet do
 3:     if obj.isLockedByAnother() then
 4:       return false                ▷ validation failed (locked object)
 5:     if obj.getVersion() != ver then
 6:       return false                ▷ validation failed (different version)
 7:   retrun true                                 ▷ validation succeed
```

Figure 3: Read set validation: verify that objects are unlocked and their versions are unchanged.

Original code	Transformed Code
`x.lock()`	`if !track(x) then goto S`
`x.unlock()`	`lockedSet.remove(x)`

Table 1: Transformation for read-only phase: each locking instruction (left column) is replaced with the corresponding code on the right; S denotes the beginning of the operation.

2.3 Transforming the Code Phases

We now describe how we synthesize the code for each of the phases. We first describe the regular three-phase flow, and then continue with describing the exceptional cases.

2.3.1 Normal Flow

We illustrate the transformation for a simple code snippet that adds a new element as the third node in a linked list. Each node is associated with a lock. The original and transformed code are provided in Figure 1. The latter uses the tracking and validation functions in Figures 2 and 3, resp. For clarity of exposition, we present a statically instrumented version, without tracking the phases using **opt**.

Our transformation instruments each lock with an additional field *version*. We assume each object supports *getVersion* and *incVersion* instruction to read and increment the version number of the lock associated with the object. We invoke *incVersion* when holding the lock, and are therefore are not concerned about contention. Note that each lock has its own version, i.e., version numbers of different locks are independent of each other.

Read-only Phase In this phase the executing thread is invisible to other threads, i.e., avoids contention on shared memory both in terms of writing and in terms of locking. During this phase, our synchronization maintains two thread-local multi-sets: *lockedSet* and *readSet*. The *lockedSet* tracks the objects that were supposed to be locked by the original synchronization. The *readSet* tracks versions of all objects read by the operation, in order to allow us to later validate that the operation has observed a consistent view of shared memory.

At the beginning of the read-only phase, we insert code that initializes *lockedSet* and *readSet* to be empty (see lines 2-3 of Figure 1b). Throughout the read-only phase, (i.e., when **opt** is true with dynamic phase transitions), we replace every lock and unlock instruction with the corresponding code in Table 1. A lock instruction on object o is replaced with code that tracks the object and the version of its lock in *lockedSet* and *readSet* (see Figure 2). An unlock instruction on object o is replaced with code that removes o from *lockedSet* (see lines 2-13 of Figure 1b).

In Figure 2 (lines 5-6), we use an eager validation scheme[1]: If the object already exists in *readSet*, we check that the current version of its lock is equal to the version in *readSet*; and if the versions are different the operation restarts (line 5). Similarly, it is checked to be unlocked, and the operation restarts if it is locked (line 6).

Although it only accesses thread-local data structures, lock tracking induces a certain overhead due to the need to search a lock in the *lockedSet* in order to unlock it. (In our experiments presented below, in large data structures, this overhead slows operations down by up to 40%). We suggest some optimizations to mitigate this cost. First, we observe that the *lockedSet* does not need to be tracked in read-only operations, which a compiler can easily detect. We can further avoid this overhead in update operations in certain cases by relying on the structure of the transformed code. For example, if the lock-based code is created from sequential code using domination locking [25], then at any given time in the read phase, it holds locks on a well-defined set of objects – the ones currently pointed by the operation's local variables. When applying our transformation to code generated by this scheme, we can optimize it to remove lock-tracking, and instead populate the *lockedSet* with the appropriate locks immediately before executing the validation phase.

---

[1]Eager validation is not required for correctness.

Validation Phase The code of the validation phase is invoked between the read-only phase and the update phase (lines 14-21 of Figure 1b). It locks the objects that are left in *lockedSet* and validates the objects in *readSet*. To avoid deadlocks, the locks are acquired using a *tryLock* instruction. If any *tryLock* fails, the code unlocks all previously acquired locks and restarts from the beginning (lines 15-18).

The function *validateReadSet* in Figure 3 verifies that the objects in the read set have not been updated. The function checks that each object in the read set is not locked by another thread, and that the current version of the lock associated with the object matches the version saved in the *readSet*. This check guarantees that the object was not locked from the time it was read until the time it was validated. Since operations write only to locked objects, it follows that the object was not changed. This *readSet* validation can be viewed as a double collect [4] of all objects accessed by the read-only phase. The operation is restarted if the validation fails (lines 19-21).

We assume that, following standard practice in lock implementations, the function *isLockedByAnother* imposes a *memory fence* (barrier). This ensures that the lock and version are read during *track* before the object's value is read optimistically during the read-only phase. To ensure that the second read of the lock and version, during the validation phase, succeeds the optimistic read of the object's value, we precede the validation phase with a memory fence as well (line 14). Note that it suffices to impose a *read fence* (sometimes called acquire or load fence) prior to the validation as well as during *isLockedByAnother*, because this part of the code does not include writes to shared memory.

Update Phase In this phase our transformation preserves the original locking while maintaining the versions of the objects, i.e., the version of an object o is incremented every time o is unlocked. Here, (i.e., in case **opt** is false with dynamic phase transitions), before each unlock instruction `x.unlock()` we insert the code `x.incVersion()` . An example is shown in lines 22-30 of Figure 1b.

2.3.2 Exceptions from Regular Flow

The read phase does not validate past reads during its executions (other than when re-reading the same variable). As a result, it may observe an inconsistent state of shared memory, which may lead to infinite loops or spurious exceptions (as explained, e.g., in [29]). We avoid such infinite loops using a timeout. If the timeout expires before the read-only phase completes, read set validation takes place (via the function *validateReadSet*). If the validation fails, the operation is restarted. This is realized by inserting code that examines the timeout in every loop iteration in the original code. Similarly, we avoid spurious exceptions by catching all exceptions and performing validation. Here too, if the validation fails, the operation is restarted. Otherwise, the exception is handled as in the original code.

Our sandboxing relies on properties of managed languages like Java or C#:

1. We can identify all instructions that may update shared memory and end the read phase before they occur.

2. The ability to capture all exits from a block via the try-finally mechanism ensures that we never exit the read phase without performing validation.

3. The code is not self-modifying and hence the tracking and validation code is executed as intended.

4. The speculative execution does not alter the references to the thread-local variables we introduce (readSet, lockedSet) since they are constant references to well-typed objects.

   Hence, our tracking and validation code executes correctly.

While recent work [16] has shown that differing validation to the end of a transaction can be unsafe, this problem does not occur in our solution. The key problem shown there is that access to the object on which the conflict is checked (namely the lock) is deferred until after other unchecked shared accesses, which could potentially be inconsistent and cause the lock not to be accessed. In our case, on the other hand, all accesses to shared data are recorded for validation purposes. If an object that should be

accessed (like the lock in the lock elision case) is not accessed because of earlier conflicts, these earlier conflicts will be detected and the transaction will abort.

Note that, using our transformation, the shared state at the end of the validation phase is identical to the state that would have been reached had the code been executed pessimistically from the outset. Hence, the three-phase version of the code is compatible with the instrumented pessimistic version. This means that if the optimistic phase is unsuccessful for any reason, we can always fall back on the pessimistic version. Moreover, we can switch from optimistic to pessimistic synchronization *at any point* during the read phase. We use this property in two ways. First, we avoid livelocks by limiting the number of restarts due to conflicts: The validation phase tracks the number of restarts in a thread-local variable. If this number exceeds a certain threshold, we perform the entire operation optimistically.

Second, this property offers the optimistic implementation the liberty of failing spuriously, even in the absence of conflicts, because it can always fall back on the safe pessimistic version of the code. One can take advantage of this liberty, and implement the *readSet* using a constant size array. In case the array becomes full, the optimistic version cannot proceed, but there is no need to start the operation anew. Instead, one can immediately perform the validation phase, which, if successful, switches to a pessimistic modus operandi, after having acquired all the needed locks.

3 Analysis

We argue that our transformation is *correct*, in the sense that all the external behaviors of the synthesized code are allowed by the original implementation. This implies that the transformed code preserves essential properties of the original, such as serializability, linearizability, and deadlock-freedom. In this section we provide informal correctness arguments; a formal proof is deferred to Appendix A.

Indistinguishability of successful executions Every operation execution of the modified code starts at the beginning of the read-only phase and, (barring exceptions, deadlocks, or infinite loops, which will be discussed later), ends in a return statement in the update phase. To show the correctness of such an execution, we argue that it is indistinguishable from an execution of the entire operation using the update phase. Since the instrumented code maintains version numbers that are not present in the original code, we need to project these out of the state in order to argue that the same states are reached in both cases. For a state s (i.e., an assignment of values to shared and local variables) in an execution of the instrumented code, we denote by $\hat{s}$ the same state without version numbers. The indistinguishability hinges on the following properties:

1. *The read-only phase has no side effects* since it does not modify shared state.

2. *An unsuccessful validation phase has no lasting effect on shared state*, since it releases all locks it acquires. Moreover, it does not cause deadlock thanks to the use of tryLocks and releasing all locks upon failure to acquire one. Since we assume that the update phase only uses blocking lock calls, the only impact an unsuccessful validation phase can have on it is additional waiting on locks, which does not impact its external behavior.

3. *A successful validation phase executed after point ℓ in the read-only phase leads to a memory state s, such that $\hat{s}$ would have been reached by performing the original update phase (from the beginning) until point ℓ.*

While the first two properties can be directly observed from the code, the third requires more careful reasoning. To understand why it is correct, recall that, in the read-only phase, the only difference between the optimistic execution and the original one is that reads are performed without holding a lock. We argue that the execution is equivalent to one that would have acquired and released locks as in the code of the update phase. To this end, we show that the object was unlocked (and hence, was not modified), for the duration of entire the period when the original code would have locked it. We distinguish between two cases:

7

1. In case the lock would have been released by original code before location ℓ, we argue that the lock was free during the read-only phase from the time when it would have been locked and at least until the time when it would have been released by the original code. This is because the successful validation means that (1) the object is currently unlocked, and (2) object's version had not increased from the time it was added to the *readSet* in the read-only phase (i.e., the *lock* instruction in the original code), and until the validation phase, i.e., after it would have been released. Recall that in our instrumented code, the object's version increases every time it is locked.

2. In case the lock would have been held by the original code at location ℓ, it is included in the *lockedSet* during the validation phase, and hence acquired before branching to the update phase. As in the previous case, the successful validation of the *readSet* ensures that the lock had been available since the time when it would have been locked by the original code.

Progress Finally, we argue that our read-only and validation phases do not generate spurious dead-locks, exceptions, or infinite loops that were not present in the original code. In other words, if the original code would have successfully completed with a return step, so does the instrumented code.

First, consider deadlocks. The read and validation phases of our instrumented code do not use blocking locks – the read-phase does not use locks at all, whereas the validation phase uses tryLocks. Therefore, both phases are non-blocking. In principle, the optimistic approach may lead to livelocks, but our algorithm falls back on the pessimistic approach following a bounded number of restarts, and hence cannot livelock. We get that any lack of progress must be due to blocking in the update phase. Since this section of the code is unchanged, and since we ensure that it begins when holding the same locks as in the original protocol, we get that our transformation does not introduce any source of spurious blocking that is not present in the original code.

Exceptions and infinite loops may be introduced in the read-only phase due to reading an inconsistent view of shared memory. Nevertheless, we detect these situations by performing validation both periodically and on exceptions: As we have argued above, the validation phase is successful only if the execution is equivalent to a lock-based one. Thus, every inconsistent view that might lead to a spurious exception or infinite loop causes the validation to fail, and we detect these cases before they have any external impact.

4   Evaluation

We evaluate the performance of our approach on search trees supporting insert, delete, and get operations. We compare the throughput of our approach to fully pessimistic solutions applying fine-grain locking, solutions based on software transactional memory, and hand-crafted state-of-the-art data structure implementations.

Methodology We use the micro-benchmark suite *Synchrobench* [26], configured as follows. Each experiment consists of 5 trials. A trial is a five second run in which each thread continuously executes randomly chosen operations drawn from the workload distribution, with keys selected uniformly at random from the range $[0, 2 \cdot 10^6]$. Each trial is preceded by initiating a new data structure with 10^6 keys and a warm-up of five seconds. Our graphs present the average throughput over all trials. We consider three representative workloads distributions: a *read-only* workload comprised of 100% lookup operations, a *write-dominated* workload consisting of insert and delete operations (50% each), and a *mixed workload* with 50% lookups, 25% inserts, and 25% deletes.

Platform All implementations are in Java. We ran the experiments 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). We used Ubuntu 12.04.4 LTS and Java Runtime Environment (build 1.7.0_51-b13) using the 64-Bit Server VM (build 24.51-b03, mixed mode).

Unbalanced		Balanced	
LO-Tree	Locked-based [19]	**LO-AVL**	Lock-based relaxed AVL [19]
LF-Tree	Lock-free [21]	**Snap-Tree**	Lock-based relaxed AVL [10]
		CF-Tree	Contention-friendly tree [14]
		Skiplist	Java lock-free skiplist

Table 2: Hand-crafted state-of-the-art data structures. The code of LO-Tree was provided by the authors, all other implementations provided by Synchrobench.

Implementations We start from textbook sequential implementations of an unbalanced internal binary tree and a treap [7]. We next synthesize concurrent lock-based code by (manually) applying the domination locking technique [25] to the sequential data structures. The resulting algorithms are denoted Lock-Tree and Lock-Treap. Then, we manually apply our lock-removal transformation to the reference implementations by following the algorithm line-by-line (requiring no understanding of the base code) to get our semi-optimistic versions of the code, which we call LR-Tree and LR-Treap, respectively. Note that this solution does not track the *lockedSet* for read-only operations and does not use eager validation of version numbers. Finally, we apply the optimization described in Section 2.3.1, which eliminates explicit tracking of the *lockedSet* in update operations, and instead locks all objects the thread holds a pointer to in the validation phase; this optimization is applicable since our parallel implementation is synthesized using domination locking. The resulting algorithms are denoted Opt-LR-Tree and Opt-LR-Treap.

For the competition, we parallelize the sequential implementations also using Deuce [23], a Java implementation of TL2 [17]. The resulting algorithms are denoted STM-Tree and STM-Treap. We further compare our implementations to their hand-crafted state-of-the-art counterparts listed in Table 2.

We also measured the performance of global lock-based implementations. In all workloads, the results were identical or inferior to those achieved by pessimistic fine-grain locking. We hence omitted these results to avoid obscuring the presentation.

Results Figures 4 and 5 show the throughput of unbalanced and balanced data structures, resp. We see that our semi-optimistic solution, both optimized and unoptimized, is far superior to the fully-pessimistic automated approach; it successfully overcomes the bottlenecks associated with lock contention in Lock-Tree and Lock-Treap.

Our approach also outperforms STM by 1.5x to 2.5x. The additional overhead of STM most likely stems from two reasons: deferring writes to commit time, and using a global clock to ensure a consistent view of the read set. The latter is done in order to satisfy opacity [28], which we avoid by "sandboxing". In our experiments, the code *never* incurred a spurious exception or timeout due to inconsistent reads, and so the sandboxing was not associated with a performance penalty.

Our solution comes close to custom-tailored implementations, and the optimized version is even superior to some of them. The throughput of our read-only operations is up to 1.5x lower than that achieved by the best-in-class. By profiling the code, we learned that the bulk of this overhead stems from the need to track all read objects, which is inherent to our transformation. This is in contrast with the hand-crafted implementations, which have small overhead on reads that complete without any retries. In workloads that include update operations, our solution is up to 2.2x slower. This stems from tracking read and locked sets and not from retries as the percentage of retries is less than 1%.

5   Related Work

Concurrent Data Structures Many sophisticated concurrent data structures (e.g., [8, 19, 40, 11, 14, 9, 5, 21, 10, 31, 24, 38]) were developed and used in concurrent software systems [44]. Implementing efficient synchronization for such data structures is considered a challenging and error-prone task [44,
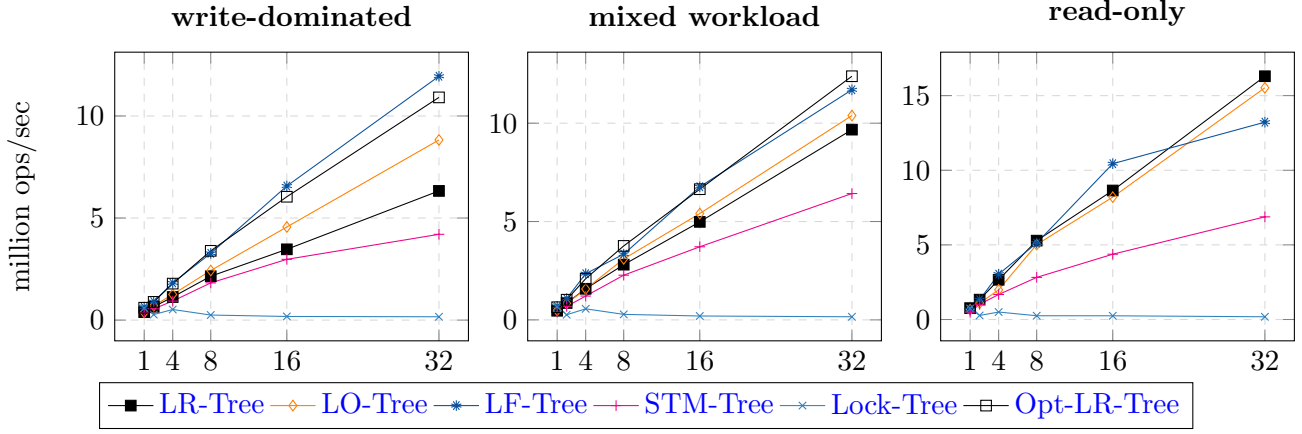
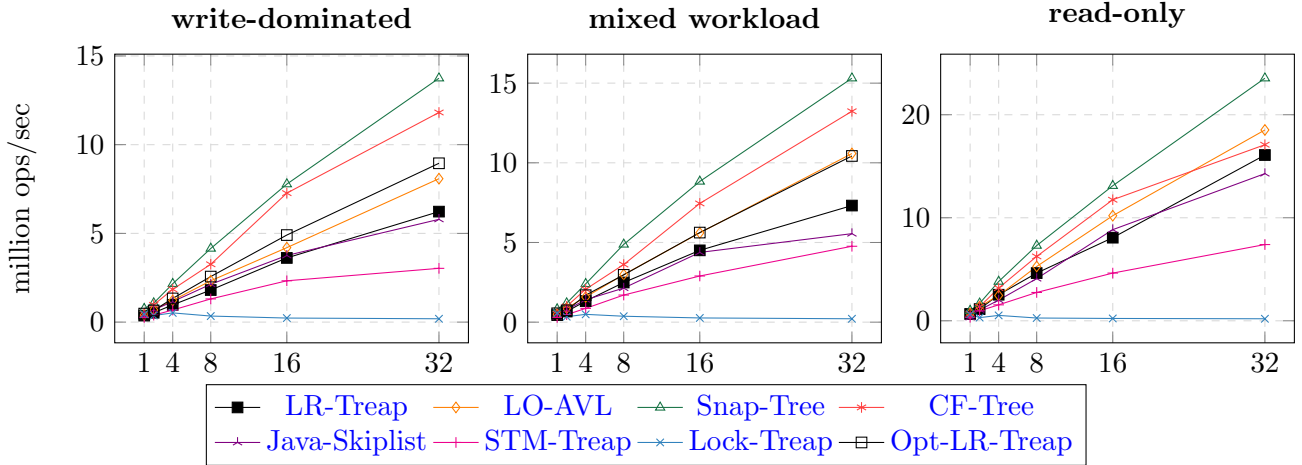Figure 4: Throughput of unbalanced data structures.

Figure 5: Throughput of balanced data structures.

18, 34]. As a result, concurrent data structures are manually implemented by concurrency experts. This paper shows that (in some cases) an automatic algorithm can produce synchronization that is comparable to synchronization implemented by experts.

Lock Inference Algorithms   There has been a lot of work on automatically inferring locks for transactions. Most algorithms in the literature infer locks that follow the two-phase locking protocol [36, 22, 27, 13, 32, 15]. Our approach can potentially be used to optimize the synchronization produced by these algorithms. For example, for algorithms that employ a two-phase variant in which all locks are acquired at the beginning of a transaction (e.g., [27, 13]), our approach may be used to defer the locking (e.g., to just before the first write operation) and even to eliminate some of the locking steps. We demonstrate the benefit of combining our transformation with such algorithms by using the domination locking protocol [25] to produce efficient concurrency control for dynamic data structures.

Transactional Memory   Transactional memory approaches (TMs) dynamically resolve inconsistencies and deadlocks by rolling back partially completed transactions. Unfortunately, in spite of a lot of effort and many TM implementations (see [29]), existing TMs have not been widely adopted due to various concerns [20, 12, 37], including high runtime overhead, poor performance and limited ability to handle irreversible operations. Modern concurrent programs and data structures are typically based on hand-crafted synchronization, rather than on a TM approach [44].

Lock Elision   Our transformation is inspired by the idea of *sequential locks* [37] and the approach presented in [39], which replace locks with optimistic concurrency control in read-only transactions. But in contrast to these works, we handle read-only prefixes of transactions (operations) that do update the shared memory. In fact, as shown in Section 4, our approach is best suited for update-dominated workloads. Moreover, using these approaches for a highly-contended data structure (as in Section 4) is likely to provide limited performance, because each update transaction causes many read-only transactions to abort.

Other works have proposed using transactional memory in order to elide locks from arbitrary critical sections, and fall back on lock-based execution in cases of aborts (e.g., [41, 42, 6]). In contrast to our approach, however, lock elision does not combine speculative and non-speculative execution within the same transaction.

6   Discussion

The development of scalable concurrent programs today heavily relies on custom-tailored implementations, which require painstaking correctness proofs. In this paper, we have shown a relatively simple transformation that can facilitate this labor-intensive process, and thus make scalable synchronization more readily available. The input for our transformation is a conventional lock-based concurrent program, which may be either constructed manually or synthesized from sequential code. Our source-to-source transformation then makes judicious use of optimism in order to eliminate principal concurrency bottlenecks in the given program and improve its scalability.

We have illustrated our method for unbalanced and balanced search trees. The transformed code performed significantly better than the original lock-based one, and scaled comparably to hand-crafted implementations that had taken considerably more effort to produce. In these examples, we have applied our transformation manually. An interesting direction for future work would be to create a tool that automatically applies our transformation at compile time.

Our approach makes use of a common pattern in data structures, where an operation typically begins with a long read-only traversal, followed by a handful of (usually local) modifications. A promising direction for future work is to try and exploit similar patterns in order to parallelize or remove locks in other types of code (not data structures), for example, programs that rely on two-phase locking.

Furthermore, for programs that follow different patterns, other combinations of optimism and pessimism may prove effective.

Finally, there still remains a gap between the performance achievable by manually optimized solutions and what we could achieve automatically. Our algorithm induces inherent overhead for tracking all operations in the read-only phase for later verification. In specific data structures, these checks might be redundant, but it is difficult to detect this automatically. We believe that it may well be possible to bridge the remaining performance gap using computer-assisted optimizations. For example, a programmer may provide hints regarding certain invariants that are always preserved in the code, in order to eliminate the need for tracking some values for later validation.

References

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

[2] A fast and lightweight key/value database library by google. http://code.google.com/p/leveldb.

[3] jmonkeyengine: a 3d game engine for java developers. http://jmonkeyengine.org/.

[4] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, Sept. 1993.

[5] Y. Afek, H. Kaplan, B. Korenfeld, A. Morrison, and R. E. Tarjan. CBTree: A practical concurrent self-adjusting search tree. In *DISC*, pages 1–15, 2012.

[6] Y. Afek, A. Levy, and A. Morrison. Software-improved hardware lock elision. In *PODC*, 2014.

[7] C. R. Aragon and R. Seidel. Randomized search trees. In *FOCS*, pages 540–545, 1989.

[8] M. Arbel and H. Attiya. Concurrent updates with RCU: search tree as an example. In *PODC*, pages 196–205, 2014.

[9] A. Braginsky and E. Petrank. A lock-free B+tree. In *SPAA*, pages 58–67, 2012.

[10] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. A practical concurrent binary search tree. In *PPOPP*, pages 257–268, 2010.

[11] T. Brown, F. Ellen, and E. Ruppert. A general technique for non-blocking trees. In *PPoPP*, pages 329–342, 2014.

[12] 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):46–58, Sept. 2008.

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

[14] T. Crain, V. Gramoli, and M. Raynal. A contention-friendly binary search tree. In *19th International Conference on Parallel Processing (Euro-Par)*, pages 229–240, 2013.

[15] D. Cunningham, K. Gudka, and S. Eisenbach. Keep off the grass: Locking the right path for atomicity. In *CC*, pages 276–290. 2008.

[16] D. Dice, T. L. Harris, A. Kogan, Y. Lev, and M. Moir. Hardware extensions to make lazy subscription safe. *CoRR*, abs/1407.6968, 2014.

[17] D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. In *DISC*, pages 194–208, 2006.

[18] S. Doherty, D. L. Detlefs, L. Groves, C. H. Flood, V. Luchangco, P. A. Martin, M. Moir, N. Shavit, and G. L. Steele, Jr. Dcas is not a silver bullet for nonblocking algorithm design. In *SPAA*, 2004.

[19] D. Drachsler, M. T. Vechev, and E. Yahav. Practical concurrent binary search trees via logical ordering. In *PPoPP*, pages 343–356, 2014.

[20] J. Duffy. A (brief) retrospective on transactional memory. 2010. http://joeduffyblog.com/2010/01/03/\a-brief-retrospective-on-transactional-memory/.

[21] F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel. Non-blocking binary search trees. In *PODC*, pages 131–140, 2010.

[22] M. Emmi, J. S. Fischer, R. Jhala, and R. Majumdar. Lock allocation. In *POPL*, pages 291–296, 2007.

[23] G. K. Felber, P. and N. Shavit. Deuce: Noninvasive concurrency with a Java STM. In *MULTI-PROG*, 2010.

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

[25] G. Golan-Gueta, N. G. Bronson, A. Aiken, G. Ramalingam, M. Sagiv, and E. Yahav. Automatic fine-grain locking using shape properties. In *OOPSLA*, pages 225–242, 2011.

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

[27] K. Gudka, T. Harris, and S. Eisenbach. Lock inference in the presence of large libraries. In *ECOOP*. 2012.

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

[29] T. Harris, J. Larus, and R. Rajwar. Transactional memory, 2nd edition. *Synthesis Lectures on Computer Architecture*, 5(1), 2010.

[30] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. N. Scherer III, and N. Shavit. A lazy concurrent list-based set algorithm. In *9th International Conference on Principles of Distributed Systems (OPODIS)*, pages 3–16, 2006.

[31] M. Herlihy, Y. Lev, V. Luchangco, and N. Shavit. A simple optimistic skiplist algorithm. In *14th International Conference on Structural Information and Communication Complexity (SIROCCO)*, pages 124–138, 2007.

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

[33] A. Israeli and L. Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *PODC*, pages 151–160, 1994.

[34] G. Jin, W. Zhang, D. Deng, B. Liblit, and S. Lu. Automated concurrency-bug fixing. In *OSDI*, 2012.

[35] K. Lev-Ari, G. Chockler, and I. Keidar. On correctness of data structures under reads-write concurrency. In *DISC*, October 2014.

[36] B. McCloskey, F. Zhou, D. Gay, and E. Brewer. Autolocker: synchronization inference for atomic sections. In *POPL*, pages 346–358, 2006.

[37] P. E. McKenney. Is parallel programming hard, and, if so, what can you do about it? *Linux Technology Center, IBM Beaverton*, Aug. 2012.

[38] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC*, 1996.

[39] T. Nakaike and M. M. Michael. Lock elision for read-only critical sections in java. In *PLDI*, pages 269–278, 2010.

[40] A. Natarajan and N. Mittal. Fast concurrent lock-free binary search trees. In *PPoPP*, pages 317–328, 2014.

[41] R. Rajwar and J. R. Goodman. Transactional lock-free execution of lock-based programs. *SIGOPS Oper. Syst. Rev.*, 36(5):5–17, Oct. 2002.

[42] A. Roy, S. Hand, and T. Harris. A runtime system for software lock elision. In *EuroSys*, 2009.

[43] O. Shacham. *Verifying Atomicity of Composed Concurrent Operations*. PhD thesis, Tel Aviv University, 2012.

[44] O. Shacham, N. Bronson, A. Aiken, M. Sagiv, M. Vechev, and E. Yahav. Testing atomicity of composed concurrent operations. In *OOPSLA*, 2011.

[45] O. Shalev and N. Shavit. Predictive log-synchronization. In Y. Berbers and W. Zwaenepoel, editors, *EuroSys*, pages 305–315. ACM, 2006.

[46] F. Zyulkyarov, V. Gajinov, O. S. Unsal, A. Cristal, E. Ayguadé, T. Harris, and M. Valero. Atomic quake: using transactional memory in an interactive multiplayer game server. In *ACM Sigplan Notices*, volume 44, pages 25–34. ACM, 2009.

A Formal Correctness Proof

We now formalize the correctness arguments made in Section 3. First we define our model and the correctness properties of the algorithm for which we provide the proof.

Model We consider an asynchronous shared memory model, where independent threads interact via shared memory objects. Every thread executes a sequence of operations, each of which is invoked with certain parameters and returns a response. An operation's execution consists of a sequence of primitive *steps*, beginning with an *invoke* step, followed by atomic accesses to shared objects, and ending with a *return* step. Steps also modify the executing thread's local variables.

A *configuration* is an assignment of values to all shared and local variables. Thus, each step takes the system from one configuration to another. Steps are deterministically defined by the data structure's protocol and the current configuration. In the *initial configuration*, each variable holds its initial value.

An *execution* is an alternating sequence of configurations and steps, $C_0, s_1, C_1, \ldots, s_i, C_i, \ldots$, where C_0 is an initial configuration, and each configuration C_i is the result of executing step s_i on configuration C_{i-1}. We only consider finite executions in this paper. An execution is *sequential* if steps of different operations are not interleaved. In other words, a sequential execution is a sequence of operation executions.

Two executions are *indistinguishable* to a set of operations if each operation in the set executes the same steps on shared objects, and gets the same value from those objects, in both executions. A step τ by operation *op* is *invisible* to all other operations if the executions with and without τ are indistinguishable to OP $\setminus \{op\}$. For example, read steps are invisible.

Correctness The correctness of a data structure is defined in terms of its external behavior, as reflected in values returned by invoked operations. Correctness of a code transformation is proven by showing that the synthesized code's executions are equivalent to ones of the original code, where two executions are *equivalent* if when considering operations that have completed every thread invokes the same operations in the same order in both executions, and gets the same result for each operation. More formally, we say in this paper that a code transformation is *correct* if every execution of the transformed code is equivalent to some execution of the original code.

The widely-used correctness criterion of serializability relies on equivalence to sequential executions in order to link a data structure's behavior under concurrency to its sequentially specified behavior. Since equivalence is transitive, we get that any code transformation satisfying our correctness notion, when applied to serializable code, yields code that is also serializable. If the code transformation further ensures the real-time order of operations (i.e., operations that do not overlap appear in the same order in executions of the transformed and original code), then linearizability (atomicity) is also invariant under the transformation. Another important aspect of correctness is preserving the progress conditions of the original code, for example, deadlock-freedom.

In this paper, we are not concerned with internal consistency (as required e.g., by opacity [28] or the validity notion of [35]), which restricts the configurations an operation might see during its execution. This is because our code transformation uses timeouts and exception handlers to overcome unexpected behavior that may arise when a thread sees an inconsistent view of global variables (similar to [39]).

Formal Proof We consider a finite execution π of the transformed code, and find an equivalent execution of the original lock-based code. Each operation in π is an interleaved sequence of read-only and validation phases followed by a (single) update phase, or a prefix of such pattern. For each operation in π we consider its *successful validation*, i.e., the last (successful) execution of a validation phase before switching to the update phase. Each operation executes at most one successful validation. The read-only phase preceding the successful validation phase is called *successful read-only phase* . Each operation executes at most one successful read-only phase. Towards proving equivalence to the original code execution, for each operation op, we remove the prefix of op that precedes the successful read-only phase. This includes completely removing operations that have no successful read-only phase. We call the resulting execution $\hat{\pi}$. The removed prefixes include read steps as well as tryLock and unlock steps. Removing read steps is invisible to other processes. Since we only remove steps that acquire locks all remaining locking steps in $\hat{\pi}$ have the same affect and get the same response (success or failure) as in π. Finally, since the operation discards all local (private) state when restarting a read-only phase, π and $\hat{\pi}$ are indistiguishable to all operations that have completed in π.

Claim A.1 π and $\hat{\pi}$ are equivalent.

Denote by $e_1, e_2, \ldots, e_k$ the sequence of the first steps of the read set validation in the execution of successful validation phases, by their order in $\hat{\pi}$, where e_i is a step of the operation op_i executed by process p_i. (Possibly $p_i = p_j$ for $j \neq i$).

Let OP be the set of operations in $\hat{\pi}$. For every operation op_i in OP, consider the partition of $\hat{\pi}$ to the following intervals $\hat{\pi} = \alpha_i \beta_i \gamma_i$, such that α_i includes the execution interval of op_i's (successful) read-only phase (denote op_i's read set rs_i); $\beta_i = \beta_{i_1} \beta_{i_2}$, is the minimal execution interval of op_i's successful validation phase; in β_{i_1}, op_i acquires locks on its lock set, denoted ls_i; e_i is the first step of β_{i_2}, namely the read set validation interval.

Claim A.2 No operation in OP $\setminus \{op_i\}$ holds a lock in $\alpha_i \beta_{i_1}$ that is associated with an object obj in rs_i after op_i's first read of obj in α_i.

Proof Let lck be the lock associated with obj. Before reading obj the first time in α_i op_i records the version number of lck (line 3 in function *track*) and checks that lck is not held by any other thread (line 6 in function *track*).

We assume the function *isLockedByAnother* imposes a memory fence and that at the beginning of the validation phase there is a read fence. If another thread acquired *lck* after the first read and did not release it, this is discovered during validation (line 4 in function *validateReadSet*). If another thread acquired *lck* after the first read–and therefore after reading the version the first time—and did release the lock, then this thread increased the version number of *lck* before releasing it. The fencing guarantees that op_i observes the version number has changed (line 6 in function *validateReadSet*). Since the validation phase of op_i is successful no operation other than op_i holds *lck* in $\alpha_i \beta_{i_1}$. ∎

We next project object versions out of $\hat{\pi}$'s configurations, and remove all accesses (reads and writes) to object versions. That is, we replace steps that access versions with local steps that modify the operation's local memory only. Note that we get an execution with exactly the same invocations, responses, local states, and shared object states, but without versions. We call the resulting execution π'.

Claim A.3 π' *and* $\hat{\pi}$ *are equivalent.*

Our main lemma constructs the execution of a fully-pessimistic locking code. The core idea is to replace the optimistic read-only phase and validation phase of each operation with a solo execution of the pessimistic lock-based read phase taking place at the point where all objects in the lock set are locked.

Lemma A.4 *There is an execution of lock-based algorithm that is equivalent to* π'.

Proof We start with the execution $\pi_0 = \pi'$. For every $i \geq 0$, we show how to perturb π_i to obtain an execution π_{i+1}. We consider the operations $\text{OP} = op_1, \ldots, \text{OP}_k$ as defined above by the steps $e_1, e_2, \ldots, e_k$. For an operation op_j such that $j \geq i + 1$ in π_i, let $\beta'_j = \beta'_{j_1} \beta'_{j_2}$ be the minimal interval containing op_j's validation phase, where β'_{j_1} is the minimal interval containing op_j's tryLock phase. Denote the configuration between β'_{j_1} and β'_{j_2} C_j. In π_i the following conditions are satisfied:

1. The operations $op_1, \ldots, op_i$ execute the fully-pessimistic locking algorithm, while the rest of the operations $\text{OPT}_i = \text{OP} \setminus \{op_1, \ldots, op_{i-1}\}$ execute our semi-optimistic algorithm.

2. For $j \geq i + 1$, no operation in $\text{OP} \setminus \{op_j\}$ holds a lock in C_j that is associated with an object *obj* in rs_j.

3. For $j \geq i + 1$, no operation in $\text{OP} \setminus \{op_j\}$ writes to an object *obj* in rs_j after op_j first read *obj* before C_j.

4. For $j \geq i + 1$, all try-lock steps by op_j are invisible to $\text{OP} \setminus \{op_j\}$.

5. π' and π_i are equivalent.

For $\text{OPT}_k = \emptyset$, we get an execution where all operations execute the pessimistic locking algorithm, and by Condition 5 π_{k+1} is equivalent to π' and we are done.

The proof is by induction on i. For the base case we consider the execution π_0. Condition 1 holds since none of the operations in this execution execute the full locking algorithm. Conditions 2 and 3 hold by Claim A.2, and since accesses to objects (other than versions) are similar in $\hat{\pi}$ and π_0. Condition 4 holds since by construction, in π_0 every step accessing an object, either for locking it or for validating it is not locked, finds the object not locked. Condition 5 vacuously holds since π' and π_0 are the same execution.

For the induction step, assume $\text{OPT}_i \neq \emptyset$ and the execution π_i satisfies the above conditions. We consider $op_i \in \text{OPT}_i$ which partitions π_i to $\alpha'_i \beta'_{i_1} \beta'_{i_2} \gamma'_i$. We replace π_i with $\pi_{i+1} = \alpha''_i \beta''_{i_1} \delta_i \beta''_{i_2} \gamma'_i$, such that α''_i, β''_{i_1}, and β''_{i_2} are the projection of α'_i, β'_{i_1} and β'_{i_2}, excluding the steps by op_i, while δ_i is a p_i-only execution interval in which p_i follows the locking algorithm while reading rs_i; after δ_i, p_i holds the locks on all objects in ls_i, and holds no lock on other objects. In other words, we replace the optimistic

read-only phase and validation phase of op_i with an execution of the original locking algorithm, taking place at C_j.

By Condition 2 of the induction hypothesis no operation holds locks associated with objects in the read set of op_i in C_i, therefore, p_i can acquire the locks on these objects while executing δ_i. By Condition 3 of the induction hypothesis no operation writes to an object obj in rs_i after op_i first read obj before C_i, hence op_i reads the same values in its read set in π_i and π_{i+1}. After δ_i, op_i holds the locks on all objects in ls_i, hence it can continue with the execution of the locking algorithm. This implies that the projection of the execution π_{i+1} on op_i follows the full pessimistic locking algorithm satisfying Condition 1.

In $\alpha_i'' \beta_{i_1}''$ we only removed read steps and tryLock steps by op_i that are invisible to all other operations, by Condition 4. Therefore, the executions $\alpha_i' \beta_{i_1}'$, ending with configuration C', and $\alpha_i'' \beta_{i_1}'' \delta_i$, ending with configuration C'', are indistinguishable to all operations in $\text{OP} \setminus \{op_i\}$. In addition, in β_{i_2}'' we only removed invisible read steps. The values of all shared objects and locks are the same in C' and C'', hence the executions $\alpha_i' \beta_{i_1}' \beta_{i_2}' \gamma_i'$ and $\alpha_i'' \beta_{i_1}'' \delta_i \beta_{i_2}'' \gamma_i'$ are indistinguishable to all operations in $\text{OP} \setminus \{op_i\}$.

The indistinguishability and the induction hypothesis imply that Conditions 2, 3, 4 hold. In addition, this implies that all completed operations return the same value in π_{i+1} and π', which means Condition 5 holds.

∎

By Lemma A.4, Claim A.1 and Claim A.3 we conclude the following theorem:

Theorem A.5 *Every execution of the transformed code is equivalent to an execution of the original locking code.*