Automatic Lock Removal for Scalable Synchronization

Abstract

We present an *automatic* approach for parallelizing sequential data structures in a way that is both safe and scalable. While there exist pessimistic transformations that make code thread-safe by adding (either global or fine-grained) locks, this approach is limited in its performance due to synchronization bottlenecks, for example, locking the root in a tree data structure. In this paper, we improve the performance and scalability of such synthesized code by reducing bottlenecks. Specifically, we present an automatic approach to eliminate many of the locking steps, relying instead on optimistic partial traversals of the data structure. We realize our approach for tree data structures using the domination locking technique. The resulting code scales well, significantly outperforms pessimistic approaches, and its performance is comparable to that achieved by 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.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

General Terms term1, term2

Keywords keyword1, keyword2

1. Introduction

The steady increase in the number of cores in today's computers is driving software developers to allow more parallelism. Indeed, many recent works have developed scalable concurrent data structures [**? ? ? ? ? ? ? ? ? ? ?**]. Such efforts are often very successful, achieving performance that scales well with the number of threads. Nevertheless, each of these project generally focuses on a single data structure (for example, a binary search tree [**?**] or [**Idit:** give another example]) and manually optimizes its implementation. Proving the correctness of such custom-tailored data structures is painstaking (for example, the proofs of [**? ? ?**] are 31,20, and ZZ pages long [**Idit:** add info], respectively). We propose an approach to replace this labor-intensive process by automatic means.

One way to automatically convert a sequential data structure into a correct (thread-safe) concurrent one using locks. The trivial way to do so is to add a single global lock protecting the entire data structure (as in *synchronized methods* in Java^{TM}, for example), but this allows no parallelism whatsoever. A more sophisticated approach can instrument the code (at compile time) and add fine-grained lock and unlock instructions [**? ? ? ?**]. Such methods are applicable to certain data structure families, for example, Domination Locking [**?**] is applicable to all trees or forests (including binary search trees, BTrees, Treaps, etc [**Idit:** more?]), and employs a variant of hand-over-hand locking [**?**], acquiring and releasing locks as it goes down the tree. Unfortunately, to date, solutions of this sort scale poorly. This is due to synchronization bottlenecks such as the root of the tree, which is locked by all operations.

In this paper, we circumvent such synchronization bottlenecks via judicious use of optimism. Specifically, we replace many (but not all) locks with speculative execution and later re-validation. If re-validation fails, the speculative phase is restarted. In striking the balance between optimism and pessimism, we exploit the common nature of data structure operations, which typically begin by traversing the data structure to a designated location, and then perform (mostly local) updates at that location. Our optimistic execution is limited to the initial read-only part of the code (the data structure traversal)[1]. Unlike most software transactional memory approaches [**? ?**], our synthesized code neither speculatively modifies shared memory contents, nor does it defer writes. Hence it never needs to rollback, and saves the overhead for tracking writes in dedicated data structures. Further comparison with related work appears in Section 6.

Our approach works as follows: Given a sequential data structure implementation, it first invokes a given (black-box) mechanism that instruments the code and adds fine-grained locks, e.g., [**? ? ?**] that satisfy some *locking protocol*. Our assumptions on the data structure and locking scheme are detailed in Section 2. We then invoke our algorithm, detailed in Section 3, which (1) adds version numbers to shared memory objects, (2) identifies the read-only prefix of the code, (3) replaces locks in the read-only prefix with mere tracking of both locks and the read objects' versions, and (4) introduces appropriate re-validation mechanisms. Re-validation occurs at the end of the read-only phase, as well as during timeouts and exceptions. The latter addresses exceptions and infinite loops that may arise when an operation sees an inconsistent view of the data structure. Our code transformation is general, in the sense that it applies to any data structure for which an appropriate locking protocol exists, as proven in Section 4. The transformation is trivial to implement at compile time.

We realize our approach with the Domination Locking scheme [**?**], which is applicable to tree and forest data structures. We apply the appropriate code transformations to balanced and unbalanced tree data structure implementations in Java^{TM}. In Section 5 we evaluate the resulting code on a 32-core machine, and compare it to fully pessimistic as well as state-of-the-art hand-crafted data structure implementations [**? ?**]. Our results show that the optimistic approach successfully overcomes synchronization bottlenecks: our synthesized data structures scale linearly [**Idit:** is linearly true?]

[1] Our solution may be seen as a form of software lock elision for read-only operation prefixes.

with the number of threads, and achieve comparable performance to that of custom-tailored solutions.

To conclude, this paper illustrates that automatic synchronization 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 may be more broadly applicable. Section 7 concludes the paper and touches on some directions for future work.

2. Model and Definitions

Shared Memory Data Structures We consider an asynchronous shared memory model, where independent threads interact via shared memory objects. For the sake of our discussion, we do not distinguish among different types of shared memory (e.g., global or heap-allocated). In addition, each thread has access to *local* (thread-local) memory.

A *data structure* is an abstract data type exporting a set of *operations*. A data structure is implemented from a collection of primitive shared *objects* supporting atomic load (read) and store (write) operations. Below, we extend the allowed primitive variables to also include locks.

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 an initial value.

An *execution* is an alternating sequence of configurations and steps, $C_0, s_1, \dots, s_i, C_i, \dots$, 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}. An execution is *sequential* if steps of different operations are not interleaved. In other words, a sequential execution is a sequence of operation executions.

Locking Protocols A *lock* is a primitive type that supports atomic *lock*, *try_lock*, and *unlock* operations, where try_lock is a non-blocking attempt to acquire a lock that may fail.

We assume in this paper an existing (black box) locking mechanism, which transforms a sequential data-structure to a concurrent one by adding locks: it associates a lock with every primitive shared object used by the data structure, and instruments the sequential code by adding lock and unlock operations. The locks are added so as to abide to some set of rules, called a *locking protocol*. Examples of such protocols are Two-Phase Locking [**?**], Tree Locking [**?**], and Domination Locking [**?**]. We note that there are known code transformations that satisfy certain locking protocols (e.g., Domination Locking [**?**]), but no known transformation for others; the question of developing such transformations is orthogonal to our contribution in this paper. Intuitively, the Domination Locking protocol, which is restricted to tree-like data structures, automatically performs some sort of "hand-over-hand" locking, acquiring locks as it traverses a linked-list or tree, and releasing locks on previously traversed nodes.

We assume that the resulting code (obtained by adding the locks) satisfies the following properties:

- Every (load or store) access by an operation to a shared object is performed when the executing thread holds the lock on that object.
- The protocol is deadlock-free, i.e., locking does not introduce deadlocks.

Correctness The correctness of a data structure is defined in terms of its external behavior, as reflected in values returned by invoked operations. This is captured by the notion of a *history* – the history of an execution σ is the subsequence of σ consisting of invoke and return steps (including the invocation parameters and return values). Correctness of a code transformation is proven by showing that the synthesized code's histories are *equivalent* to ones of the original code, in the sense that one is a permutation of the other (i.e., includes the same invoke and return steps).

The widely-used correctness criterion of serializability relies on equivalence to sequential histories in order to link a data structure's behavior under concurrency to its allowed behavior in sequential executions. 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 transformation also satisfies the real-time order of operations (that is, operations that do not overlap appear in the same order in the histories of the transformed and original code), then it further preserves linearizability (or atomicity) of code it transforms.

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

3. Automatic Transformation

We address the problem of introducing optimism to a pessimistic locking protocol, performing load and store steps on locked objects only, given a sequential algorithm and a "black box" transformation that ensures this protocol. The idea of optimistic synchronization is to read shared variables without locks and start using the locking protocol when the operation reaches a store step. This scheme is applicable to many data structure implementations since the common behaviour of data structure operations is to first traverse the data structure and perform mostly local modifications at the end of the operation. Our algorithm shows that it is redundant to acquire locks if they are freed before any change is made.

Conceptually, the optimistic scheme separates the operation to three *phases*, an optimistic *read-only phase*, a pessimistic *update phase* and a *validation phase* that conjoins them. The optimistic read-only phase traverses the data structure without taking any locks while maintaining minimal sufficient information to later ensure the correctness of the traversal. The update phase maintains the original locking protocol. In order to bridge between the optimistic read phase and the pessimistic update phase we use the validation phase. The validation phase first locks the objects required by the locking protocol and then validates the correctness of the of the read-only phase, allowing the update phase to run as if the an execution of the locking protocol took place.

3.1 Transformation Details

We start our transformation by applying three preparation steps to the given sequential algorithm. First, we use the "black box" transformation to produce an algorithm that ensures the locking protocol. Second, we add version numbers to shared memory objects, incremented every time the object is locked. These version numbers are used to validate the correctness of the read-only phase during the validation phase as well as at timeouts and exceptions. Lastly, we identify the read only prefix of each operation.

Next we explain what the transformation does in each phase.

Read-only Phase During this phase the operation maintains a *locked set* and a *read set*. The locked set is used to track the lock and unlock steps of the locking protocol, thus, we replace each lock step with a step adding the object to the locked set and each unlock step with a step removing the object from the locked set. (Since the locking protocol might allow re-entrant locks, our locked set allows duplications and the remove operation removes one of the duplications of the object).

The read set is used to track all objects accessed by the operation in order to later validate that this read set belongs to a consistent view of shared memory. It contains a mapping between references to objects loaded to the local version at the time it was read. After objects are added to the read set they are checked to be unlocked, if the object is locked the operation restarts from the beginning.

The read phase does not validate reads during its traversal, in order to avoid infinite loops, a timeout is set. If the operation reaches the timeout, a read set validation takes place (described in the validation phase), if it fails the operation restarts from the beginning.

Validation Phase The validation phase is inserted to the code after the read-only phase. It locks the objects in the locked set and validates the read set.

To avoid deadlocks, the locks are acquired using a try_lock operation, if the try_lock fails, the operation unlocks previously acquired locks and restarts from the beginning. The calls to try_lock also increment the versions of the object, these objects are also part of the operations read set. To avoid the operation invalidating itself, a successful try_lock operation is followed by incrementing the version saved in the read set.

The read set validation is performed as follows, each object saved in the read set is checked to be unlocked and that the current version matches the version saved in the read set. If this check fails, the operation releases all its locks and restarts from the beginning. This guarantees that the object was not locked from the time it was read until the time it was validated, since all operations write only to locked nodes it follows that the object was not changed. The read set validation can be viewed as a double collect to all objects accessed by the read-only phase.

Update Phase This phase enforces the locking protocol while maintaining the local versions, i.e., the local version of an object is incremented every time it is locked. Once the update phase begins, the operation is guaranteed to to finish without restarts.

4. Algorithm's Correctness

We will prove that if the original locking protocol is conflict-serializable then our algorithm is conflict-serializable.

Let π be an execution of our optimistic automation on a sequential algorithm. We will construct an execution π_{LP} which is an execution following the original locking protocol. We will prove that both executions are conflict-equivalent. Since any execution of the original locking protocol is conflict-serializable, then π is conflict-serializable.

Let $p_1, p_2, \ldots, p_n$ be the operations $\in \pi$ ordered by the order of execution of the first step of a successful read_set validation. (If some operation does not have such point we omit it). Let $\pi_{LP} = \pi_{lp1}, \pi_1, \ldots, \pi_{lpi}, \pi_i$ where π_{lpi} is a p_i-only execution of original locking protocol until p_i holds locks only on the local variable locked in the validation phase of $p_i \in \pi$, and π_i is the interval of π starting from the return from the validation of pi until the first step of the successful read_set validation of p_{i+1} that includes only the operations by $\{p_1, \ldots, p_i\}$. In other words, we replace the read-phase and validation phase with an execution of the original locking protocol, taking place at the point just before the read_set validation starts.

LEMMA 4.1. *The construction of π_{LP} is feasible.*

Proof Proof by induction on $p_1, p_2, \ldots, p_n$. Base case is immediate.

Let $\pi' = \pi_{lp1}, \pi_1, \ldots, \pi_{lpk-1}, \pi_{k-1}$ be the feasible construction so far, and let p_k be the next operation to be added.

Assume by contradiction that $\pi' \cdot \pi_{lpk} \cdot \pi_k$ cannot be constructed, thus, some object v that p_k locks in π_{lpk} is already locked by $p_j \in \{p_1, p_2, \ldots, p_{k-1}\}$ in the last configuration of π'. If p_j locked v before p_k read v for the first time, then v was locked during the read phase of p_k, in contradiction to p_k reaching its validation. Otherwise, p_j locked v after p_k read v. If v is still locked during the validation of p_k then the validation will fail, contradiction. Alternatively, v was unlocked by p_j before p_k validated v, its version incremented to a version bigger than the local version read by p_k, contradicting the successful validation of p_k. ■

LEMMA 4.2. *π_{LP} is conflict-equivalent to π*

Proof Each operation performs a double collect on all the values it reads. The first collect is the read phase and the second is the read_set validation of the validation phase. Since validation was successful, both collect are identical, meaning that the values of the read_set do not change from the return of the last read of the read phase, until the first read of the read_set validation. Therefore, executing the original locking of p_k after $\pi' = \pi_{lp1}, \pi_1, \ldots, \pi_{lpk-1}, \pi_{k-1}$ is conflict-equivalent to the original read phase. The read-write phase remains unchanged, maintaining conflict-equivalence to π. ■

5. Evaluation

In order to apply our approach to a data structure, a "black-box" pessimistic locking is required. One such approach was presented in [**?**], automatically applying domination locking protocol to forest based data structures. We applied domination locking followed by our optimistic transformation to two tree based data structures, a simple unbalanced binary search tree and a treap (randomized binary search tree) [**?**].

Setup We compared the performance of our automatic implementations, the automatic binary search tree (Auto-Tree) and the automatic treap (Auto-Treap), to the following custom tailored approaches:

- LO-Tree- The locked-based unbalanced tree of Drachsler at al.[**?**].
- LO-AVL- The locked-based relaxed balanced AVL tree of Drachsler et al.[**?**].
- OPT-Tree- The locked based relaxed balanced AVL tree of Bronson et al.[**?**].
- Skip-List- The non-blocking skip-list by Doug Lea included in the the Java standard library.

We also compared our algorithms to previous automatic approaches, mainly global locking (Lock-Tree, Lock-Treap) and domination locking (Domination-Tree, Domination-Treap).

We ran our experiments on four Inter Xeon E-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^{TM} Runtime Environment (build 1.7.0_51-b13) using the 64-Bit Server VM (build 24.51-b03, mixed mode).

We evaluated the performance on a variety of workloads, each workload is defined by the percentage of read-only operations (GET queries) and the remaining operations are divided equally between insert and delete operations. Our workloads include heavy read-only workloads (100% GET operations), medium read-only

```
 1: function FOO(Node new)
 2:    Node prev = head
 3:    prve.lock()
 4:
 5:
 6:
 7:
 8:    Node succ = prev.next
 9:    succ.lock()
10:
11:
12:
13:
14:     prev.unlock()
15:     prev = succ
16:     succ = succ.next
17:     succ.lock()
18:
19:
20:
21:
   ----------------------------
22:
23:
24:
25:
26:
27:
   ----------------------------
28:     prev.next = new
29:     new.next = succ
30:     prev.unlock()
31:     succ.unlock()
```

```
 1: function FOO(Node new)
 2:    Node prev = head
 3:    lockedSet.add(prev)
 4:    long version = prev.getVersion()
 5:    readSet.add(prev,version)
 6:    if prev.isLocked() then
 7:       goto 1
 8:    Node succ = prev.next
 9:    lockedSet.add(succ)
10:     version = succ.getVersion()
11:     readSet.add(succ,version)
12:     if succ.isLocked() then
13:        goto 1
14:     lockedSet.remove(prev)
15:     prev = succ
16:     succ = succ.next
17:     lockedSet.add(succ)
18:     version = succ.getVersion()
19:     readSet.add(succ,version)
20:     if succ.isLocked() then
21:        goto 1
   ----------------------------
22:     if !lockedSet.tryLockAll() then
23:        release all locks
24:        goto 1
25:     if !readSet.validate() then
26:        release all locks
27:        goto 1
   ----------------------------
28:     prev.next = new
29:     new.next = succ
30:     prev.unlock()
31:     succ.unlock()
```

Figure 1: Code option 1

workload (50% GET operations) and update only workload (0% GET operations).

We used two key ranges $[0, 2 \cdot 10^4]$ and $[0, 2 \cdot 10^6]$, for each range, the tree was pre-filled until the tree size was within 5% of half the key range.

We ran five seconds trials measuring the total throughput (number of operations per second) of all threads. During the trial, each thread continuously executed randomly chosen operations according to the workload distribution using uniformly random keys from the key range. We ran every trial 7 times, we report the average throughput while eliminating outliers.

Results Figure 2 reports the throughput of unbalanced data structures and Figure 3 reports the throughput of the balanced data structures. The results for the read-only workload show the main overhead of our automatic approach. Unlike the hand crafted implementations that have no overhead on reads in this scenario, either by wait free reads in LO-Tree and LO-AVL or the optimistic validation of OPT-Tree, our implementation requires full read_set validation. This overhead cannot be avoided without prior knowledge on the data structure.

As the update workload increases . . .

6. Related Work

7. Discussion

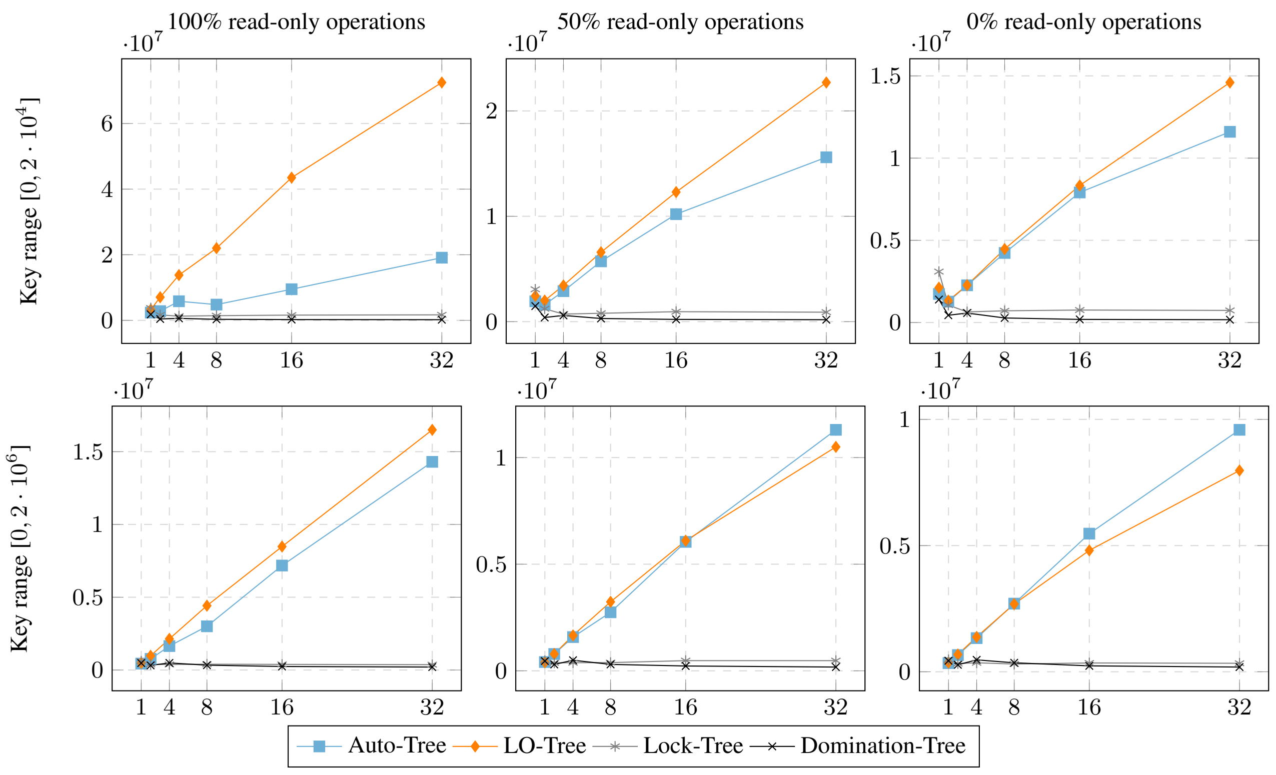

Figure 2: Throughput of unbalanced data structures.

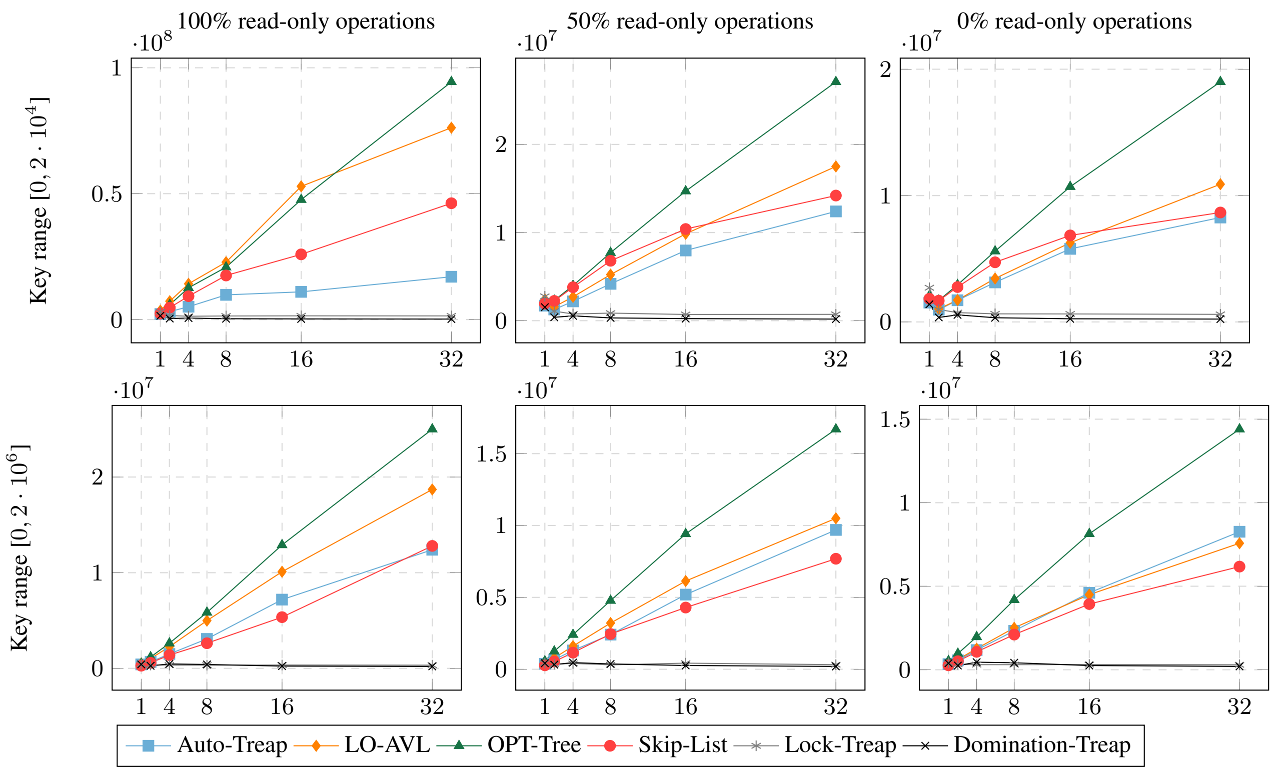

Figure 3: Throughput of balanced data structures.