

Scalable Automatic Synchronization for Concurrent Data Structures

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 achieves performance close to those of custom-tailored concurrent data structures. 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 [*sub-category*]: third-level

General Terms term1, term2

Keywords keyword1, keyword2

1. Introduction

2. Model and Definitions

2.1 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. In Section 2.2, we extend the allowed primitive variables to also include locks.

Each 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.

2.2 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 a *locking protocol*, which transforms a sequential data-structure to a correct concurrent one by adding locks; correctness is defined in Section ?? below. Examples of such protocols are Tree Locking [8], and Domination Locking [4]. The locking protocol associates a lock with every primitive shared object used by the data structure, and instruments the sequential code by adding lock and unlock operations. Intuitively, such protocols automatically perform some sort of “hand-over-hand” locking, acquiring locks as they traverse a linked-list or tree, and releasing locks on previously traversed nodes. They are typically restricted to tree-like data structures.

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.
- The protocol allows early lock release in the sense that it never needs to hold a lock on an object that it no longer has a pointer to.

The last condition means that even if the protocol holds a lock on an object it no longer holds a pointer to, it is safe to unlock it at this point (and re-acquire the lock later if it is accessed again), in the sense that correctness, as defined below, is not breached.

2.3 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. The widely-used correctness criteria of linearizability and serializability link the data structure’s behavior under concurrency to its allowed behavior in sequential executions. The latter is defined by a *sequential specification*, which is a set of its allowed sequential histories.

A history H is *linearizable* [6] if there exists H' that can be created by adding zero or more return

steps to H , and there is a sequential permutation π of $\text{complete}(H')$, such that: (1) π belongs to the sequential specification; and (2) every pair of operations that are not interleaved in σ , appear in the same order in σ and in π . A data structure is *linearizable*, also called *atomic*, if the histories of all of its executions are linearizable.

A history is *serializable* if it satisfies property (1) above. That is, the real-time order of operations is not required. A data structure is *serializable* the histories of all of its executions are serializable.

In this paper, we are not concerned with internal consistency (as required e.g., by opacity [5] or the validity notion of [7]), 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. ...

We address the problem of introducing optimism to a pessimistic locking protocol. Given an implementation using a pessimistic locking protocol, performing load and store steps on locked objects only, we add optimistic synchronization. The idea of optimistic synchronization is to load shared variables without locks and start using the locking protocol only when the operation reaches a store step. Our algorithm shows that it is redundant to acquire locks if they are freed before any change is made.

This optimistic scheme separates the operation to three *phases*, an optimistic *read phase*, a pessimistic *read-write phase* and a *validation phase* that connect them.

3.1 Detailed Algorithm

Each object maintains a counter, incremented every time the object is locked. This counter is used to validate the correctness of the optimistic read phase.

Read Phase Starts at the beginning of the operation and ends at any point before the first store operation. During this phase the operation maintains a *read_set*, containing references to objects loaded and the local version when it was read. The local versions are incremented during the read-write phase, when the object is locked. Incrementing the version is not atomic with the lock, thus, the object is also checked to be unlocked. The read phase does

not validate reads, in order to avoid infinite loops, a timeout is set. If the operation reaches the timeout, a `read_set` validation takes place, if it fails the operation restarts from the beginning. The use of timeout does not guarantee that the operation reads a consistent snapshot of the memory, thus, ...

Validation Phase This phase connects the read phase with the read-write phase. It has two requirements: (i) lock local variables of the operation and (ii) ensure that the values read during the read phase are consistent, i.e. as if the values were read while executing the locking protocol. To avoid deadlocks, the locks are acquired using a `try_lock` operation, if the `try_lock` fails, the operation restarts from the beginning. Next, the `read_set` is validated, if the validation fails the operation restarts from the beginning. During the `read_set` validation, each reference saved in the `read_set` is checked to be unlocked and that the current version matches the version saved in the `read_set`.

Read-Write 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 read-write phase begins, the operation is guaranteed 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, \dots, 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 $\pi_{LP} = \pi_{lp1}, \pi_1, \dots, \pi_{lpi}, \pi_i$ 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 p_i until the first step of the successful `read_set`

validation of p_{i+1} that includes only the operations by $\{p_1, \dots, p_i\}$. 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, \dots, p_n . Base case is immediate.

Let $\pi' = \pi_{lp1}, \pi_1, \dots, \pi_{lpk-1}, \pi_{k-1}$ 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 $p_j \in \{p_1, p_2, \dots, p_{k-1}\}$ 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 $\pi' = \pi_{lp1}, \pi_1, \dots, \pi_{lpk-1}, \pi_{k-1}$ 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 [4], 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) [1].

```

1: function FOO(Node new)
2:   prev = head
3:   succ = prev.next
4:   prev = succ
5:   succ = succ.next
6:   new.next = succ
7:   prev.next = new

1: function FOO(Node new)
2:   Node prev = head
3:   prve.lock()
4:   Node succ = prev.next
5:   succ.lock()
6:   prev.unlock()
7:   prev = succ
8:   succ = succ.next
9:   succ.lock()
10:  new.next = succ
11:  prev.next = new
12:  prev.unlock()
13:  succ.unlock()

```

```

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

```

Figure 1

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 et al.[3].
- LO-AVL- The locked-based relaxed balanced AVL tree of Drachsler et al.[3].
- OPT-Tree- The locked based relaxed balanced AVL tree of Bronson et al.[2].
- 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 a . . .

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%,70% GET operations), medium read-only workload (50% GET operations) and update only workload (0% GET operations).

We used two key ranges $[0, 2 \cdot 10^5]$ 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.

6. Related Work

A. Appendix Title

This is the text of the appendix, if you need one.

Acknowledgments

Acknowledgments, if needed.

References

- [1] C. R. Aragon and R. Seidel. Randomized search trees. In *30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, USA, 30 October - 1 November 1989*, pages 540–545, 1989. . URL <http://doi.ieeecomputersociety.org/10.1109/SFCS.1989.63531>.
- [2] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. A practical concurrent binary search tree. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2010, Bangalore, India, January 9-14, 2010*, pages 257–268, 2010. . URL <http://doi.acm.org/10.1145/1693453.1693488>.
- [3] D. Drachsler, M. T. Vechev, and E. Yahav. Practical concurrent binary search trees via logical ordering. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, Orlando, FL, USA, February 15-19, 2014*, pages 343–356, 2014. . URL <http://doi.acm.org/10.1145/2555243.2555269>.
- [4] 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.
- [5] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *PPOPP*, pages 175–184, 2008.
- [6] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990. . URL <http://doi.acm.org/10.1145/78969.78972>.
- [7] K. Lev-Ari, G. Chockler, and I. Keidar. On correctness of data structures under reads-write concurrency. In *DISC*, October 2014.
- [8] A. Silberschatz and Z. Kedem. Consistency in hierarchical database systems. *J. ACM*, 27(1):72–80, Jan. 1980. ISSN 0004-5411. . URL <http://doi.acm.org/10.1145/322169.322176>.

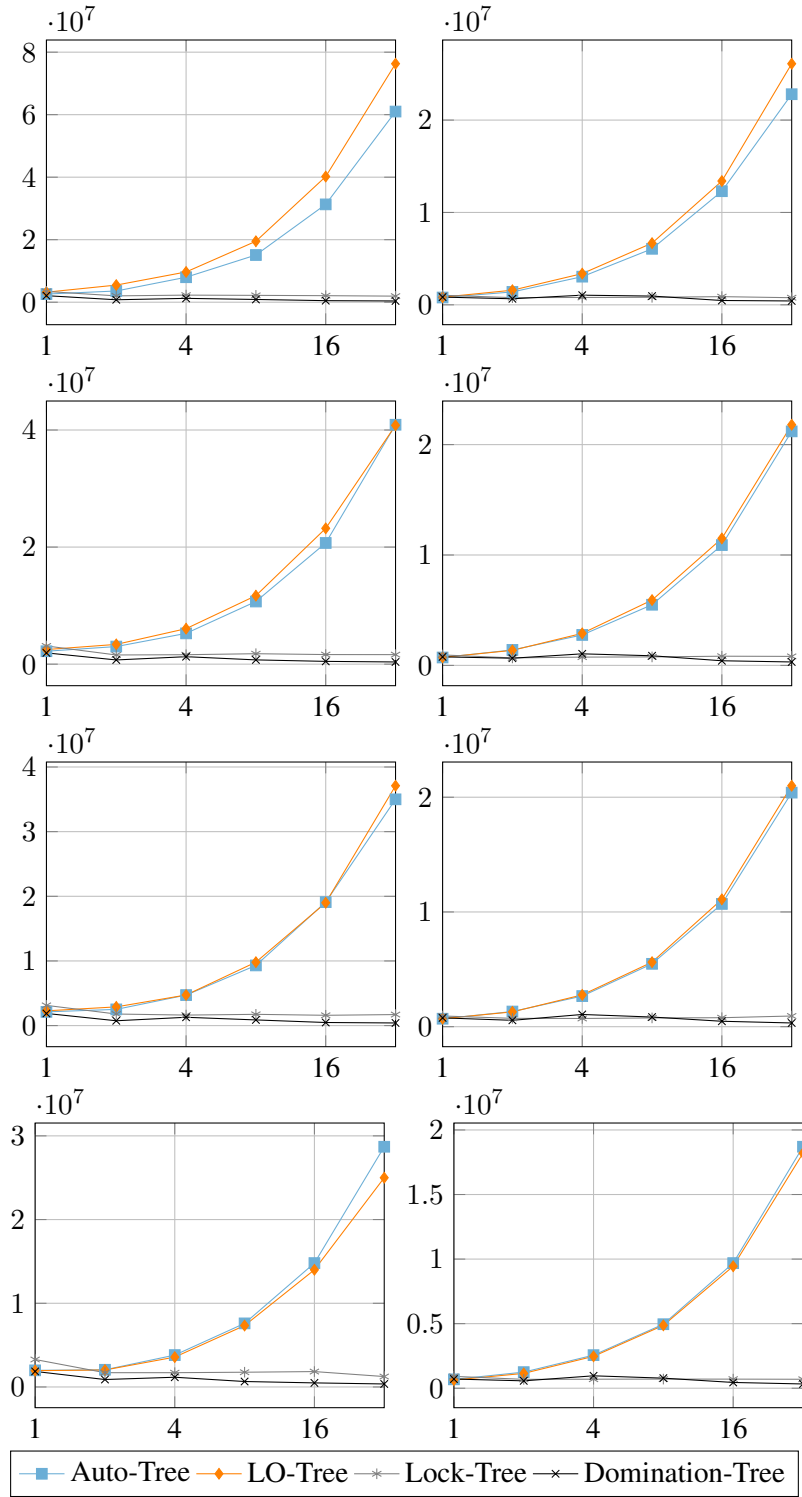


Figure 2: Throughput of unbalanced data structures.

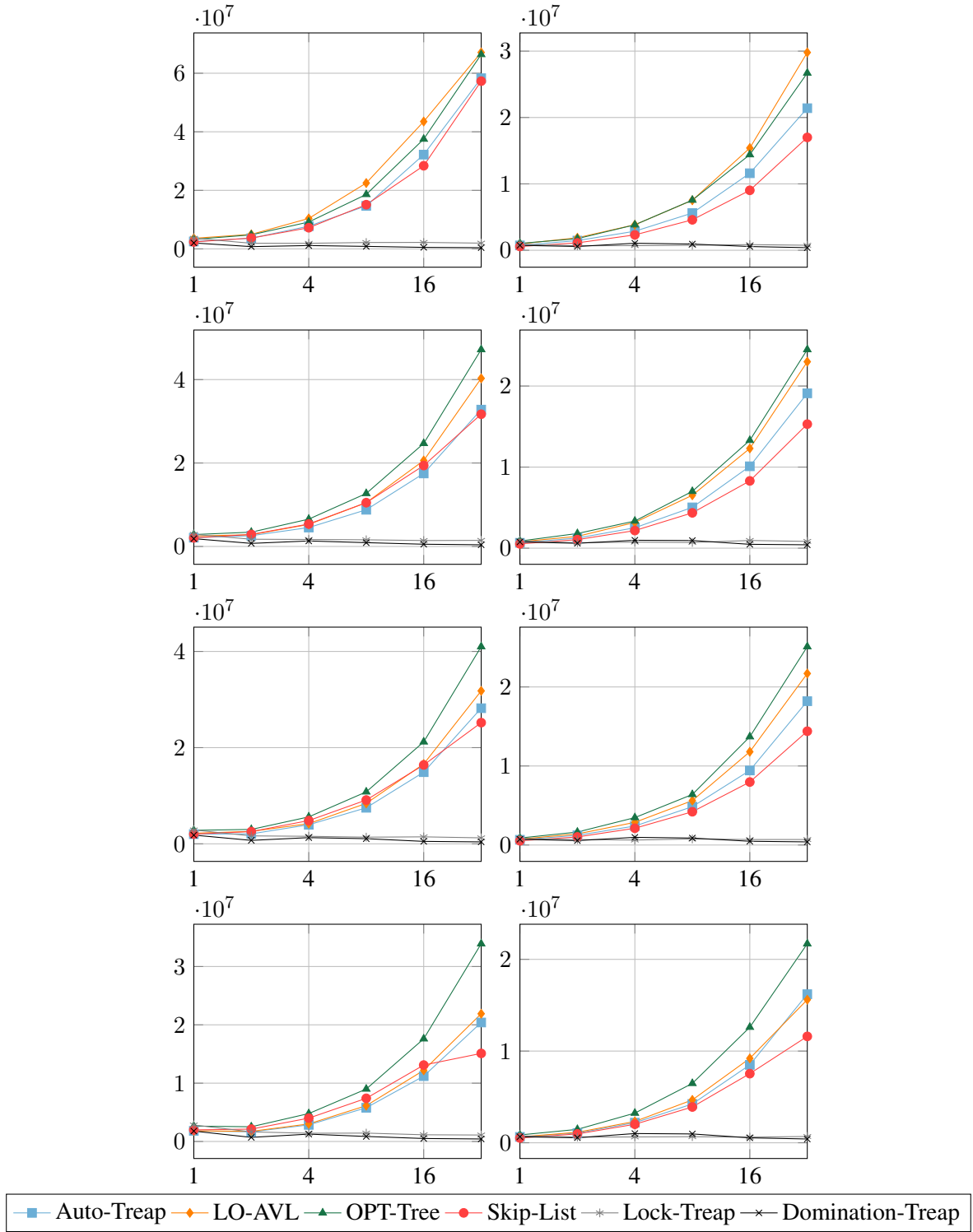


Figure 3: Throughput of balanced data structures.