Technical Report

# Sequential Proximity

## Towards Provably Scalable Concurrent Search Algorithms

Karolos Antoniadis, Rachid Guerraoui, Julien Stainer, and Vasileios Trigonakis*

École Polytechnique Fédérale de Lausanne

## 1 Introduction

This report contains complementary definitions on sequential proximity [2]. Furthermore, in this report we motivate sequential proximity by using it to prove that two concurrent search data structures are sequentially proximal and show how sequentially proximity can help a developer create a highly-scalable linked list.

In Section 2 we present related work. In Section 3 we give precise definitions for logical deletion and cleaning-up stores. Then, in Section 4 we prove two relations between sequential proximity properties and classic progress conditions. In Section 5 we prove that a lock-based linked list [17] is sequentially proximal, while in Section 6 we prove that a non-blocking linked list [16] satisfies sequential proximity. We conclude in Section 7 where we present the trend of concurrent search data structures towards sequential proximity, an example on how we can end-up with a highly-scalable concurrent linked list and some instances where sequential proximity is violated.

## 2 Related Work

On the one hand, sequential proximity (SP) can be viewed as the formalization of a vast amount of prior work that directly or indirectly calls for sequential-like concurrent designs (Section 2.1). On the other hand, SP continues a long tradition of formal properties defined with respect to a sequential behavior (Section 2.2).

### 2.1 Common Practices

Several designers of concurrent search data structures (CSDSs) [9, 10, 15, 17, 22] pose as a design goal, one way or another, similarity to sequential algorithms. For example, in the lazy list [17] algorithm "*The wait-free nature of the membership operation means that ongoing changes to the list cannot delay even a single thread from deciding membership.*" Similarly, the skip-list algorithm by Herlihy et al. [22] has "*searches exactly as in a sequential skip list.*"

Read-copy update (RCU) [29] is a technique for designing CSDSs with sequential, wait-free reads. RCU targets read-mostly workloads. Arbel and Attiya [3] extend RCU to better support concurrent updates. PRCU [4] reduces the granularity of waiting in RCU. RLU [28] further provides concurrency of reads with multiple writers. All these RCU variants enforce sequential search operations while improving the scalability of updates.

Flat combining [18] and variants, such as RCL [27], optimize highly-contended critical sections by employing sequential executions of those critical sections. Unlike SP, flat combining targets data structures with single points of contention, such as queues. Still, the idea of sequential execution to minimize synchronization is identical in SP.

Universal constructions [19] can implement non-blocking CSDSs using their respective sequential design. Universal constructions are rarely used in practice due to their inferior performance compared to hand-tuned implementations. Similarly, the transactional memory (TM) abstraction [20, 36] takes as input sequential code. TM executes this code with the necessary synchronization for preserving the sequential semantics in the face of concurrency.

* This work has been supported in part by the European ERC Grant 339539 - AOC.

OPTIK [15] is a design pattern for concurrent data structures. OPTIK implements simple "transactions" using validation with version numbers (i.e., read version number, execute, then lock and validate the version number). OPTIK leads to sequential-like transactions with a "fixed" amount of locking and validation.

Asynchronized concurrency (ASCY) [9] is a paradigm comprising four informal patterns on how to design CSDSs that scale across workloads and platforms. ASCY's approach to scalability is, similarly to SP, to reduce synchronization by designing concurrent algorithms that are close to their sequential counterparts. Compared to SP, ASCY contains only high-level informal hints on designing scalable CSDSs, not precise and formal properties.

All these aforementioned practical approaches either offer hints, or implementation tools on how to design scalable concurrent data structures.

### 2.2 Theoretical Approaches

Classic correctness properties, such as linearizability [21] and serializability [34], are typically defined with respect to a sequential specification. In the same vein, our definition of SP is based on a concrete sequential implementation of the corresponding data structures.

Disjoint-access parallelism (DAP) [25] roughly states that operations that take as parameters different memory addresses should not conflict on the same memory locations. Therefore, such operations progress independently. Similarly, in a CSDS that satisfies SP, only update operations that operate on the same vicinity in the respective sequential structure are allowed to conflict.

The scalable commutativity rule (SCR) [8] states that "*whenever interface operations commute, they can be implemented in a way that scales.*" By definition, insert and delete operations of the CSDS interface do not commute. Still, we can define commutativity between concrete parameters in CSDSs: Any two operations on distinct elements commute. SCR's definition of scalability is unsuitable even for these cases, because it is based on the notion of conflict-free accesses which excludes cases such as a process writing to a memory location that was read by another process. These intricacies of CSDSs are captured by SP, but not by SCR.

The "laws of order" paper [6] proves that atomic operations, such as compare-and-swap, cannot be completely eliminated in many concurrent software constructs, such as queues and locks. Similarly, SP "allows" insertions and deletions in CSDSs to perform up to a fixed number of atomic operations (either for locking, or for lock-freedom).

Gibson et al. [13] (as well as [9, 17]) argue that search operations of CSDSs, such as lists, are better off selfish (i.e., if they do not help other operations, e.g., by cleaning up the list). $\mathrm{SP}_1$ captures this exact behavior.

Atalar et al. [5] introduce a way of modeling data structures in order to predict their throughput. This model however only applies to lock-free data structures that have a constant retry loop [1], such as stacks and queues. SP targets both lock-free and lock-based CSDSs.

## 3 Logical Deletion Algorithms

Many CSDSs [16, 17], when deleting a value (node) from their data structure, first mark the node to be deleted and later on do the actual deletion, i.e., physical removal of the node.

### 3.1 Logically Deleting

Here we define what it means for a node to be logically deleted. Roughly speaking, a node is logically deleted if the node is physically accessible, meaning it can be reached from the root pointer but search operations are not able to find the value residing in this node.

Formally we say that a $NodeAlloc(t_{al}, \pi)$ has been *logically deleted* if:

- $NodeAlloc(t_{al}, \pi)$ was allocated during an $\mathsf{insert}(v)$ operation call;
- there is a successful $\mathsf{delete}(v)$ that is linearized at a write that corresponds to a transition $t_{del}$ and this write was issued to a location $l \in NodeAlloc(t_{al}, \pi)$;
- and $NodeAlloc(t_{al}, \pi) \subseteq reachable(root, \infty)_{t_{del}}$, since it can be accessed by the root pointer.

### 3.2 Cleaning-Up Stores

Given a well-formed execution $\pi$ and a contiguous subsequence $\pi_{(a,b)}$ of it, let us consider a transition $t_w \in \pi_{(a,b)}$ that corresponds to a global write or compare-and-swap statement in a location $l$. We say that $t_w$ is a *cleaning-up store* in $\pi_{(a,b)}$ if one of the two following conditions hold.

- If $t_w$ is a global write or a successful compare-and-swap (i.e., one such that a call to compare-and-swap($l, old, new$) returns $old$) transition and there is a logically deleted node $NodeAlloc(t_{al}, \pi)$ such that $NodeAlloc(t_{al}, \pi) \subseteq reachable(root, \infty)_{t_w - 1}$ and $NodeAlloc(t_{al}, \pi) \cap reachable(root, \infty)_{t_w} = \emptyset$. This means that $NodeAlloc(t_{al}, \pi)$ is still physically accessible before the store, but not after it. Transition $t_w - 1$ corresponds to the exact preceding transition of $t_w$ in $\pi$. For the above relations to hold, it should be the case that $t_w$ is not the first transition in our execution, something that is valid to assume if we consider that writes in a well-formed program take place only inside operations. This implies that, before $t_w$, there was an entry transition by the same process in $\pi$.
- If $t_w$ is an unsuccessful compare-and-swap transition. Then there should be a transition $t_r$ in $\pi_{(a,b)}$ by the same process as the one taking $t_w$ that reads location $l$ and a transition $t_{w'}$ between $t_r$ and $t_w$, taken by a different process, that writes to location $l$.

The second condition is used to avoid having CSDSs that satisfy sequential proximity, where compare-and-swap statements fail for no apparent reason.

Note that cleaning-up stores are not confined in parse phases but could as well take place in modify phases.

## 4 Progress of Traversals

In what follows, we say that an operation *op* of a program *Prog* satisfies "$n$ steps non-blocking". This means, that for any execution $\pi \in [\![Prog]\!]$, if for any transition $t_s \in \pi$ that corresponds to an operation-entry of *op* and $t_e \in \pi$ its matching operation-exit, then process $proc(t_s)$ is "$n$ steps non-blocking" in the interval $\pi_{(t_s, t_e)}$. In case transition $t_s$ has no matching operation-exit in $\pi$, then process $proc(t_s)$ is "$n$ steps non-blocking" in the execution interval starting from transition $t_s$ until the end of execution $\pi$.

Also, note that we assume a scheduler that does not prevent any process from taking steps.

**Proposition 1.** *If an operation op satisfies "$n$ steps non-blocking" and allocates a finite amount of memory, then it is obstruction-free.*

*Proof.* Consider such an operation *op* that allocates a finite amount of memory and satisfies "$n$ steps non-blocking". Note, that "$n$ steps non-blocking" requires that at least one global read is issued during every $n$ steps taken by the process executing *op*. So, if *op* is executed solo, eventually it is going to run out of memory locations it can read, and it is going to finish.

**Definition 1 (Totally Ordered Nodes).** *We say that a program Prog has* totally ordered nodes *if there exists a one-to-one function $f$ such that, for every execution $\pi \in [\![Prog]\!]$ and any pair of* allocate *transitions $(t_1, t_2)$ in $\pi$, either $f(NodeAlloc(t_1, \pi)) < f(NodeAlloc(t_2, \pi))$, $f(NodeAlloc(t_1, \pi)) = f(NodeAlloc(t_2, \pi))$, or $f(NodeAlloc(t_1, \pi))) > f(NodeAlloc(t_2, \pi))$.*

For example, the function $f$ could be based on a field of the actual node, such as a value field, etc.

For the following definition we define $read(opTrans(t_{en}, \pi))$ that corresponds to the read transitions of an operation. Formally, $read(opTrans(t_{en}, \pi))$ is the subsequence of transitions of $opTrans(t_{en}, \pi)$ that issue a global read. For a transition $t_r$ that issues a read in $\pi$, we define $al(t_r, \pi)$ to be the transition $t_{al}$ such that $rloc(t_r) \subseteq NodeAlloc(t_{al}, \pi)$ (i.e., $al(t_r, \pi)$ is an allocate transition that allocates memory from where transition $t_r$ reads from).

**Definition 2 (Ordered Traversals).** *Consider a program Prog that has totally ordered nodes with function $f$. Prog is said to have* ordered traversals *if for every execution $\pi \in [\![Prog]\!]$, for every* entry *transition $t_{en}$ in $\pi$, if $read(opTrans(t_{en}, \pi)) = t_1, t_2, \ldots, t_n$, then for any $t_i, t_j$ in $read(opTrans(t_{en}, \pi))$ with $i < j$ it is the case that $f(NodeAlloc(al(t_i, \pi), \pi)) \leq f(NodeAlloc(al(t_j, \pi), \pi))$.*

**Definition 3 (Bounded Number of Nodes).** *We say that a* program *Prog* has a bounded number of nodes *if $\exists k \in \mathbb{N}$ such that there is no execution $\pi \in [\![Prog]\!]$ with at least $k+1$ transitions $t_1, t_2, \ldots, t_{k+1} \in \pi$ and $f(NodeAlloc(t_i, \pi)) \neq f(NodeAlloc(t_j, \pi))$ for all $i \neq j$.*

**Proposition 2.** *If an operation op that has "$n$ steps non-blocking", follows an ordered traversal by a program Prog in which all operations allocate a finite amount memory, and Prog has a bounded number of nodes then op is wait-free.*

*Proof.* Consider such an operation *op* that has "$n$ steps non-blocking" and follows an ordered traversal. Since, *op* has "$n$ steps non-blocking", it must read a location every $n$ steps, since the nodes are totally ordered, it has to read locations from "greater" nodes each time. And since the number of nodes is bounded, eventually *op* is going to finish its execution. Therefore *op* is wait-free.

## 5 Lazy Linked List Proof

Here we prove that the slightly modified lazy linked list [17] whose algorithm is presented[1] in Figure 5.2 is sequentially proximal with respect to the sequential linked list depicted in Figure 5.1. The slight modification corresponds to the parse phase result checks before actually acquiring the locks.

Before we present the proof, we describe some conventions regarding our pseudo-code presented in Figure 5.2.

**Language**. Although the syntax of our language does not contain while, for, or other similar constructs, we include them in our pseudo-code examples that follow since they increase the readability of the examples. They can be easily created using conditional expressions and branching constructs.

We slightly abuse notation when referring to nodes by writing $n$.x to mean the value of the x field of the node $n$ based on its set of allocated memory locations (usually x would correspond to some of the memory locations that belong to the node). For example, $n$.next could correspond to the value of a next pointer of a node. Similarly, for allocate statements, we write allocate($n$) instead of allocate($n$.v, $n$.next, ...).

**Initialization of Globals**. We consider that the init operation is executed by a unique process before any other transition of the program. The $\mathsf{init}_G$ reference pointing to the head node that was allocated by the init operation in Figure 5.2 is available to all processes.

**Proof**. We denote by $Lazy_L$ the lazy linked list program and by $Seq_L$ the underlying SDS. Note that, in the algorithm presented in Figure 5.2, we mark the commands (except the lock command) that issue a global write to a memory location allocated during some other operation with "(GW)".

For example, although the statements at lines 39 and 40 in $Lazy_L$ issue global writes, we do not mark them as such, since the writes are issued to the allocated node (newNode) of the operation. In other words, the transitions that issue those writes do not belong to the $OtherNodeWrites$ set. It is the write at line 42 that makes the node reachable by other processes.

Note that we consider that insert and delete operations can be invoked with any value $v$, besides $-\infty$ and $+\infty$. Moreover, the entry and exit statements are omitted from the pseudo-code: They correspond respectively to the call and the return statement of the function implementing the concerned operation.

We start by proving that $Lazy_L$ produces only well-formed executions.

**Lemma 1.** ***Every execution $\pi$ of $Lazy_L$ is well-formed.***

[1] Our presented algorithm uses only one lock for insertion instead of two, but as the authors mention in [17] one lock is adequate.

*Proof.* Consider an execution $\pi$ of $[\![Lazy_L]\!]$. The only calls to functions that occur inside the search, insert, and delete operations are calls to the auxiliary function validate that does not call any function itself. Consequently, for each process $p$, a matching exit statement (return statement of the function) immediately follows any entry statement (call to the function corresponding to the operation) in $hs(\pi)|_p$. It follows that, for any process $p$, $hs(\pi)|_p$ is sequential. Hence, $hs(\pi)$ is a well-formed history.

Consider now an entry insert transition $t_{en}$ in $\pi$. Since there is no branching nor return between beg- and end-parse or beg- and end-modify statements, they appear by pairs in the right order in $pm(opTrans(t_{en}, \pi))$. Moreover, an end-parse (or an end-modify) returning false at line 28 (or line 50) is immediately followed by an exit insert false (return false) statement (lines 30 and 52). A successful end-modify is followed by a return true statement and a successful end-parse by a beg-modify. Finally, an end-modify statement returning restart (line 50) triggers a jump to a beg-parse statement (line 51). Consequently, any insert operation of $\pi$ follows a parse-modify pattern. The same applies to delete operations.

The analysis of the code of Figure 5.2 shows that lock and unlock statements, as well as global reads and writes only occur inside the search insert and delete functions (operations) (and inside the auxiliary function validate which is itself only called from insert and delete functions). Consequently, $\pi$ has no global transitions outside operations.

Moreover, in the case of insert and delete operations, these global instructions only take place between beg- and end-parse statements or between beg- and end-modify statements. It follows that $\pi$ has no global update transition outside parse and modify phases.

We can then conclude that all the executions $\pi$ of $[\![Lazy_L]\!]$ are well-formed.

We continue by presenting lemmas that help us show that $Lazy_L$ satisfies the sequential proximity properties.

**Lemma 2.** ***$Lazy_L$ has search read-only traversals**: For any search entry transition $t_{en}$ in an execution $\pi \in [\![Lazy_L]\!]$, there is no transition executing a write instruction in any sequence of $traversals(t_{en}, \pi)$.*

*Proof.* Checking the code of the search (lines 55-59) we can see that no stores are issued to global memory. Therefore no transition executing a write instruction takes place.

**Lemma 3.** ***The value of a node remains unchanged in $Lazy_L$**: The value field of a node after it is inserted in the list never changes.*

*Proof.* After the value field of a node is initialized at line 39, it is never modified. This can be verified by looking at the algorithm in Figure 5.2, there is no statement changing the value field of a node.

**Lemma 4.** ***No process $p$ can modify the fields of a locked node during an update operation, if the node is locked by some other process $p'$ in $Lazy_L$**: The fields of a node that is locked by some process $p'$ cannot be modified by some other process $p \neq p'$ that is executing an insert or a delete operation.*

*Proof.* Nodes are modified during either insert or delete operations. In an insert operation, a global write occurs at line 42, but the node where previous points to has been locked at line 33. In a delete operation a write occurs at lines 85 and 86, but in both cases the nodes where previous and current point to have been locked at lines 78 and 79 respectively. Thus, in both cases a node is modified only after it has been locked and since a lock cannot be acquired more than once, a node that is being locked by some process $p'$ cannot be modified by some other process $p \neq p'$.

**Lemma 5.** ***Nodes are stored in increasing order of values in $Lazy_L$**: Given two nodes $n_1$ and $n_2$ in $Lazy_L$ such that the next field of $n_1$ points to $n_2$ then $n_1$.value $<$ $n_2$.value at any point during an execution of $Lazy_L$.*

*Proof.* Initially, after the execution of the init operation, it is the case that head.next=tail and head.value $= -\infty < +\infty =$ tail.value, so the inequality holds.

We continue by showing that every time the next field of a node is changed, the inequality still holds. The next field of a node is changed during insert operations (line 42) or during delete operations (line 86).

Consider an insert operation that was invoked with a parameter $v$, and the last two nodes the parse phase traversed correspond to the nodes $n_1$ and $n_2$, that correspond to previous and current respectively in the code. We know that for the value of node $n_2$, it is the case that $v \leq n_2$.value or otherwise the parse phase would have continued (line 22) traversing nodes. Since the node $n_2$ was read, this means that $v > n_1$.value, otherwise the parse phase would have stopped (line 22) when it read the value of node $n_1$. Since the parse phase was successful (line 29) this means that $n_2$.value $\neq v$ and thus $v < n_2$.value. Afterwards, node $n_1$ is locked (line 33). It is then validated that the read nodes have not been modified (line 34) which means that $n_1$ still points to $n_2$ and neither of the two nodes is logically deleted (marked). Since node $n_1$ is locked, no node can be appended between $n_1$ and $n_2$ by some other process because such a process would need to lock $n_1$ as well. Similarly, neither $n_1$ or $n_2$ can be logically deleted or removed from the list. Which means that after $n_1$ is locked, it is still the case that $n_1$.value $< v <$ $n_2$.value. Since a locked node cannot be modified by other processes due to Lemma 4 node $n_1$ cannot be modified to point to some other node. And due to Lemma 3, the value field of a node does not change. The new node is appended between $n_1$ and $n_2$ and contains a value that is in-between $n_1$.value and $n_2$.value so the desired inequality still holds.

Consider a delete operation that was invoked with a parameter $v$, and the last two nodes the parse phase traversed are the nodes $n_1$ and $n_2$, that correspond to previous and current respectively in the code. Consider that $n_2$.next points to a node $n_3$. It should be the case that $n_1$.value $<$ $n_2$.value, $n_2$.value $= v$ (otherwise the parse phase would have failed at line 74), and $n_2$.value $<$ $n_3$.value. Nodes $n_1$ and $n_2$ are locked at line 78 and line 80 respectively. It is then validated (line 80) that nodes $n_1$ and $n_2$ have not been modified: They are both not marked and $n_1$ still points to $n_2$. Due to lemmas 3 and 4 a node cannot be inserted after $n_1$ or $n_2$ and the value field of a node does not change, so it still holds that $n_1$.value $<$ $n_2$.value $<$ $n_3$.value. After the deletion, when node $n_2$ has been physically removed, $n_1$ points to $n_3$, and it is still the case that $n_1$.value $<$ $n_3$.value.

Therefore nodes are always stored in increasing order of their values, at any point during the execution.

**Lemma 6.** *$Lazy_L$ **has** search **no back-step traversals**: For any execution $\pi \in [\![Prog]\!]$, for any search entry transition $t_{en}$ taken by process $p$ in $\pi$, in every sequence trav in sequence in $traversals(t_{en}, \pi)$, process $p$ has no back-steps in trav.*

*Proof.* Initially a search starts by reading head which contains the value $-\infty$. A search operation continues by moving to subsequent nodes following the next pointers (line 57) of the nodes. Due to Lemma 5, when a search operation reads the value $n$.value of a node $n$ and then moves to the next node $n'$, then $n$.value $<$ $n'$.value. Due to Lemma 3 the value of a node remains unchanged. Therefore, we infer that a node is never revisited and there are consequently no back-steps.

**Lemma 7.** *$Lazy_L$ **has** search **non-blocking traversals**: There exists an $n \in \mathbb{N}$ such that for any execution $\pi \in [\![Prog]\!]$, for any search entry transition $t_{en}$ taken by a process $p$ in $\pi$, $p$ is $n$ steps non-blocking in every sequence of $traversals(t_{en}, \pi)$.*

*Proof.* For the "non-blocking" condition we set the needed $n$ to be equal to 4. This means that at least one global read is issued every 4 steps and furthermore no memory location is read more than 4 times. This can be seen by checking what happens during the search operation, invoked with a parameter $v$. We can see that the head of the list is assigned to current at line 55, then its value is read at line 56 and is compared with the value $v$. Afterwards, if the comparison was successful the next field of the node is read at line 57 and now current points to another node. In this case, in just 3 steps, the search operation moved to a new node. As was stated in Lemma 6, search operations do not revisit nodes. Therefore after moving to a new node, previously read memory locations are never read again. If the *while* condition was unsuccessful then the value and *marked* fields of the node are read in 2 steps (line 59), so a total number of 4 steps before the search operation finishes.

To summarize, in at most 4 steps executed during a search operation a global read is issued. Moreover a location is never read more than 4 times.

**Lemma 8.** *$Lazy_L$ **has no** insert **or** delete **back-steps during a parse phase in** $Lazy_L$: For any execution $\pi \in [\![Prog]\!]$, for any update entry transition $t_{en}$ taken by process $p$ in $\pi$, $p$ has no back-steps in any sequence of $traversals(t_{en}, \pi)$.*

*Proof.* Parse phases are quite similar to search operations. A parse phase of an insert operation starts by reading head which contains the value $-\infty$. The parse phase then continues by moving to subsequent nodes following the next pointers of the nodes. For every node, its value field (line 22) and its next field (line 24) are read. The statement at line 23 does not correspond to a global read, it just assigns previous the same pointer that current contains.

Due to Lemma 5, when a parse phase reads the value $n.value$ of a node $n$ and then moves to the next node $n'$, then $n.value < n'.value$. Therefore, we infer that a node is never revisited.

Still, it could be the case that during the evaluation of the parse phase result (lines 26-27) a back-step is taken (i.e., a location that belongs to some previously visited node is read).

But in the parse phase result, only the value field of current is read. But current points to the last node that was read (when the while loop condition evaluated to false at line 22). So re-reading the value does not constitute a back-step. The marked fields were never read before line 26 during the parse phase, so those reads also do not constitute a back-step.

Similar arguments are applied to parse phases of delete operations. Therefore, we conclude that no process takes back-steps in a parse phase.

**Lemma 9.** *$Lazy_L$ **has insert and delete non-blocking traversals**: There exists an $n \in \mathbb{N}$ such that for any execution $\pi \in [\![Prog]\!]$, for any update entry transition $t_{en}$ taken by a process $p$ in $\pi$, $p$ is $n$ steps non-blocking in every sequence of $traversals(t_{en}, \pi)$.*

*Proof.* The proof is similar to Lemma 7. The existing $n$ can be any value greater or equal to 8. This means that at least one global read is issued every 8 steps and furthermore no memory location is read more than 8 times. This can be seen by checking what happens during a parse phase. Since the parse phases of insert and delete operations are almost the same (except the evaluation of the parse phase result) let us consider the parse phase of an insert operation.

If the check in the while loop (line 22) fails, then in at most 5 steps the parse phase will be finished. If not, the parse phase updates its previous and current pointers. As was stated in Lemma 8, a parse phase keeps traversing the list while reading the value field of the node current, points to, without revisiting already traversed nodes. Therefore memory locations, such as the value or next fields of a node are never read again, except during the evaluation of the parse phase result. Still, if the while condition was unsuccessful, then the value and marked fields of the node are read in 3 steps, so a total number of 5 steps before the update operation finishes.

To summarize, in at most every 8 steps executed during a parse phase a global read is issued. Moreover a memory location is never read more than 8 times. Same arguments apply for the parse phase of a delete operation.

**Lemma 10.** *$Lazy_L$ **has no allocation traversals and no allocation modifications**: For any search transition $t_{en}$ taken in $\pi$, there is no transition executing an allocate instruction in any sequence of $traversals(t_{en}, \pi)$. Furthermore, for any transition $t_{en}$ taken in $\pi$ that executes entry delete, there is no transition executing an allocate instruction in any sequence of $modifications(t_{en}, \pi)$.*

*Proof.* This directly follows from the code shown in Figure 5.2. The search and delete operations never allocate memory. Only delete operations call another operation, namely the validate operation (line 80), which does not allocate any memory either.

**Lemma 11.** ***No stores are issued during a parse phase in** $Lazy_L$: For any update entry transition $t_{en}$ in $\pi$, there are no stores in any sequence of $traversals(t_{en}, \pi)$.*

*Proof.* No stores are issued during a parse phase as can been seen in Figure 5.2. Specifically for an insert operation a parse phase consists of the statements from line 19 to 28 and none of these statements issue a global write. For a delete operation, a parse phase consists of the statements from line 64 to 73 and none of these issue a global write.

In the following lemmas, when saying that a node is inserted in the list of $Lazy_L$, we consider it happens when the statement at line 42 (Figure 5.2) is executed.

**Lemma 12.** *$Lazy_L$* ***has*** **insert** ***and*** **delete** ***read-only unsuccessful modifications****: For any complete sequential history $S \in Spec_{SDS}$ and any sequence of processes $P$, the solo execution $\pi = se(S, Prog, P)$ verifies that: For every* entry *op transition $t_{en}$ in $\pi$ that has a matching* exit *op* false *statement in $hs(\pi)$, it is the case that $modifications(t_{en}, \pi) = \emptyset$.*

*Proof.* For proving this lemma, we first argue that an update operation enters a modify phase at most once in a solo execution. An update operation can execute a modify phase more than once only because of restarts. Due to Lemma 19, restarts only occur due to concurrency (i.e., a concurrent process writing to a node). Therefore in a solo execution of an update operation of $Lazy_L$, a modify phase can be executed at most once.

Regarding unsuccessful modifications, there are two possible cases: unsuccessful insert and delete operations. Consider an unsuccessful insert operation that entered the modify phase. Since it is unsuccessful, this means that the operation returned false at line 36. This occurs only if current.value $= v$ (at line 35). But at the end of the parse phase it was checked that current.value $\neq v$ (line 29). Since, the operations are being executed solo, current.value could not have been modified by some other process in the meantime. Since there is at most one modify phase that can take place in an update operation, this contradicts the statement that an unsuccessful insert has entered the modify phase. Similar argument can be applied to show that $Lazy_L$ has delete read only unsuccessful modifications.

**Lemma 13.** ***Number of stores****: $Lazy_L$ has a sequential number of stores per modification, with respect to $Seq_L$.*

*Proof.* This can be easily seen in tables 1 and 2. A modify phase in $Lazy_L$ only acquires one lock during an insert (line 33) and two locks during a delete (lines 78 and 79). Also, an insert only issues one global write (line 42) to a node that was not allocated by it. A delete issues only two global writes (line 85 and 86).

| $Seq_L$ **Number of Stores and Freed Nodes** |
|---|
| $MaxOtherNodeWrites$(insert) $= 1$ |
| $MaxOtherNodeWrites$(delete) $= 1$<br>$MaxFreedNodes$(delete) $= 1$ |

**Table 1.** Number of stores and freed nodes by the sequential linked list.

| $Lazy_L$ **Number of Stores and Acquired Locks** | |
|---|---|
| **insert** | $\lvert AcquiredLocks(modi)\rvert = 1$ |
| | $\lvert OtherNodeWrites(modi, \pi)\rvert = 1$ |
| **delete** | $\lvert AcquiredLocks(modi)\rvert = 2$ |
| | $\lvert OtherNodeWrites(modi, \pi)\rvert = 2$ |

**Table 2.** Number of stores and acquired locks by the lazy linked list.

**Lemma 14.** ***The value of a node remains unchanged in*** *$Seq_L$: The* value *field of a node never changes after it is inserted in the list.*

*Proof.* After the value field of a node is initialized at line 27, it is never modified. This can be verified by looking at the algorithm in Figure 5.1, there is no statement changing the value field of a node.

For the following lemma, when we say that a node is locked by some process, we mean that this process executed a lock statement on the lockf field of this node and has not yet issued an unlock statement on the same lockf field.

**Lemma 15.** ***Nodes are stored in increasing order of values in*** $Seq_L$*: Given two nodes* $n_1$ *and* $n_2$ *in* $Seq_L$ *such that the* next *field of* $n_1$ *points to* $n_2$ *then* $n_1$.value $<$ $n_2$.value *at any point during an execution of* $Seq_L$*.*

*Proof.* Since there is no concurrency in $Seq_L$, similar arguments as in Lemma 5 can be used to prove this lemma, although simpler ones.

**Lemma 16.** ***No marked or locked nodes exist in a steady state in*** $Lazy_L$*: In any steady state* $\pi \in [\![Lazy_L]\!]$ *that is produced by a solo execution there is no reachable marked or locked node.*

*Proof.* A node can only be marked during a delete operation. After a node is marked (line 85) then it is physically removed (line 86) by the same process that is executing the delete operation. When a node is physically removed it is not reachable anymore from the head of the list. Thus, an operation that starts traversing the list after the delete operation finished will not be able to reach ("see") this node.

Concerning locked nodes, operations always unlock their acquired locks before returning. An insert operation first locks the node pointed to by previous at line 33 and then unlocks this same node at line 49. A delete operation first locks the nodes pointed to by previous and current at lines 78 and 79 respectively, it then unlocks the nodes pointed to by current and previous at lines 93 and 94 respectively. Note that nodes are unlocked even if an update operation restarts due to a failed validation, at line 51 of insert operations or at line 96 of delete operations.

Thus, we conclude that there exists no marked or locked nodes in a steady state.

**Lemma 17.** ***Each node is associated with exactly one value in*** $Lazy_L$*: There is a one to one correspondence between a node and its value.*

*Proof.* Nodes are created when memory is allocated which occurs only during an insert (line 38) operation or during the init operation (line 4 or 6). In the case of an insert, that was invoked with a parameter $v$, the allocated node is associated with the value $v$. While the first node allocated by init corresponds to value $-\infty$, and the second to the value $+\infty$.

**Lemma 18.** ***Each node is associated with exactly one value in*** $Seq_L$*: There is a one to one correspondence between a node and a value.*

*Proof.* Similar to Lemma 17.

**Lemma 19.** $Lazy_L$ ***has valid conflict restart modifications, with respect to*** $Seq_L$*: For any complete sequential history* $S'$ *with at least four tuples,* $(S', Seq_L, Lazy_L)$ *is a valid restart triple.*

*Proof.* Consider a complete sequential history $S' = S, en_0, ex_0, en_1, ex_1$. We are going to prove that $t = (S', Seq_L, Lazy_L)$ is a valid restart triple.

Assume by the way of contradiction that $t$ is not a valid restart triple. This means that a solo execution $\pi_S = se(S', Seq_L, P_S)$ exists such that transitions $t_{en_0}$ and $t_{en_1}$, that correspond to entry statements $en_0$ and $en_1$ in $\pi_S$, are conflict-free, but there exists an extension $\pi_{C'}$ of $se(S, Lazy_L, P_C)$ such that the transitions that correspond to the entry statements $en_0$ and $en_1$ in $\pi_{C'}$ are not restart-free.

An operation in $Seq_L$ can write to at most one node (e.g., for an insert operation, one global write takes place at line 29) and obviously at most one node is freed. Therefore, we have $WrittenNodes(t_{en_0}, \pi_S) \cup FreedNodes(t_{en_0}, \pi_S) \subseteq \{rn_0, rn_1\}$ and $WrittenNodes(t_{en_1}, \pi_S) \cup FreedNodes(t_{en_1}, \pi_S) \subseteq \{rn'_0, rn'_1\}$. $Lazy_L$ writes to at most one node during an insert operation and at most to two nodes in case of a delete operation. But in the case of a delete operation, the second written node corresponds to the node to be freed (due to marking at line 85). So, in the concurrent execution of the operations $en_0$ and $en_1$ in $\pi_{C'}$, $en_0$ is going to write to at most two nodes with values $rn_0$.value and $rn_1$.value, and $en_1$ is going to write to at most nodes with values $rn'_0$.value and $rn'_1$.value. Since the transitions $t_{en_0}$ and $t_{en_1}$ are conflict-free in $\pi_S$, this means that $(WrittenNodes(t_{en_0}, \pi_S) \cup FreedNodes(t_{en_0}, \pi_S)) \cap (WrittenNodes(t_{en_1}, \pi_S) \cup FreedNodes(t_{en_1}, \pi_S)) = \emptyset$, which means (due to Lemma 17) that the sets of values are disjoint: $\{rn_0.\mathsf{value}, rn_1.\mathsf{value}\} \cap \{rn'_0.\mathsf{value}, rn'_1.\mathsf{value}\} = \emptyset$. Subsequently, this means that the concurrent execution of operations $en_0$ and $en_1$ in $\pi_C$ are going to write to different values and hence different nodes. Restarts in both insert and delete operations occur when the call to validate, lines 34 and 80 respectively, returns false. This happens if the nodes pointed to by previous or current are modified. Since, both operations write to different nodes, it is the case that one operation does not invalidate the other. Meaning a restart cannot occur, a contradiction.

**Lemma 20.** ***Nodes created by solo executions of $Lazy_L$ and $Seq_L$ that contain the same value, are associated with the same relative node:*** *Consider a sequential history $S$ that is executed solo by $Lazy_L$ and $Seq_L$. At the end of the execution, $Lazy_L$ contains a node $n_{Lazy}$ that is reachable from the head of the list generated by $Lazy_L$. While $Seq_L$ contains a node $n_{Seq}$ that is reachable from the head of the list generated by $Seq_L$. Consider that the node $n_{Lazy}$ is associated with the relative node $rn_{Lazy}$, while the node $n_{Seq}$ is associated with the relative node $rn_{Seq}$. If $n_{Lazy}.\mathsf{value} = n_{Seq}.\mathsf{value}$ then $rn_{Lazy} = rn_{Seq}$.*

*Proof.* Due to Lemma 16 all reachable nodes after a solo execution are not marked or locked in $Lazy_L$. The same applies to $Seq_L$, since the sequential linked list does not use any locks or employ logical deletions.

Assume by the way of contradiction that nodes $n_{Lazy}$ and $n_{Seq}$ exist such that $n_{Lazy}.\mathsf{value} = n_{Seq}.\mathsf{value}$ but $rn_{Lazy} \neq rn_{Seq}$. Since $n_{Lazy}.\mathsf{value} = n_{Seq}.\mathsf{value}$, both nodes were allocated (line 38) during an insert operation that was invoked with a parameter of value $v$, where $v = n_{Lazy}.\mathsf{value} = n_{Seq}.\mathsf{value}$. Since the node was created (allocated), this means that this insert invocation was successful (i.e., it returned true). Assume that $rn_{Lazy}{=}(l, 1)$ while $rn_{Seq}{=}(s, 1)$ where the second part of both nodes is one since insert operations call the allocate instruction only once. Since $rn_{Lazy} \neq rn_{Seq}$, this implies that $l \neq s$. Assume that $l < s$, this means that when the subsequent insert operation was invoked with parameter $v$, the insert operation that exists in position $s$ of the sequential history $S$[2], could not have returned true, since a unmarked node with value $v$ was still reachable in the list. Similar arguments apply if $s < l$. A contradiction. Therefore $rn_{Lazy} = rn_{Seq}$.

**Lemma 21.** ***Region of stores per modification****: $Lazy_L$ has a valid region of stores per modification with respect to $Seq_L$.*

*Proof.* Assume by the way of contradiction that there exists a sequential history $S$, two sequences of processes $P_C$ and $P_S$ and an operation *op* such that $Lazy_L$ does not satisfy $\mathrm{SP}_9$. If there are many operations in $S$, then we consider as *op* the first operation such that this holds. We examine the cases based on what *op* can be: insert or delete.

Since search operations do not issue any stores, neither in $Lazy_L$, nor in $Seq_L$ we do not have to take them into account. Nevertheless, we start by showing that for a search operation in $S$, $Lazy_L$ and $Seq_L$ read the same nodes. This helps us show that $Lazy_L$ and $Seq_L$ write to similar nodes.

Assume *op* is a search operation that was invoked with a parameter of value $v$. search operations read the same nodes while looking for value $v$. Let us assume they do not. We prove that this is impossible, by showing that both search operations in $Seq_L$ and $Lazy_L$ read the same nodes and in the same order. We start by noticing that both operations read from their respective head node. Assume that the first two nodes that they read, that have different values, are $n_{Seq}$ from $Seq_L$ and $n_{Lazy}$ from $Lazy_L$ with $n_{Seq}.\mathsf{value} \neq n_{Lazy}.\mathsf{value}$. Since the nodes have different values, assume that $n_{Seq}.\mathsf{value} < n_{Lazy}.\mathsf{value}$, this means that the $Lazy_L$ never visited a node with value $n_{Seq}.\mathsf{value}$ and since the nodes are stored in increasing order of their values (Lemma 5). this means that value $v$, is never going to be read by $Lazy_L$, a contradiction. The same argument can be applied when $n_{Seq}.\mathsf{value} > n_{Lazy}.\mathsf{value}$. Both programs are eventually going to stop when they read the first node that contains a value greater or equal to $v$, which should be the same in both algorithms for the aforementioned reason. Since both search operations read the same values for each node, due to Lemma 20 they are going to read the same relative nodes.

[2] Note that $Lazy_L$ and $Seq_L$ execute the exact same sequential history $S$.

We can now discuss about update operations. Assume that $op$ is an insert or delete operation for a value $v$. The same argument as before can be used to show that a parse phase is going to read the exact same nodes in both $Seq_L$ and $Lazy_L$. Therefore since both algorithms write the previous pointer, which corresponds to the same node, they are going to have equal sets of written nodes. Note that since we are talking about solo executions, due to what was described in the proof of Lemma 12, at most one modify phase takes place during an update operation. Since parse phases do not issue any write instruction in $Lazy_L$, the written nodes during all the modifications of an update operation correspond to the written nodes of the operation.

**Theorem 1.** *$Lazy_L$ is sequentially proximal with respect to $Seq_L$.*

*Proof.* Lemma 2 entails that search operations of $Lazy_L$ follow $\mathrm{SP}_1$. Lemma 7 implies they respect $\mathrm{SP}_2$, Lemma 6 shows that they fulfill $\mathrm{SP}_3$, while Lemma 10 proves that they respect $\mathrm{SP}_4$.

By Lemma 9, the parse phases of update operations in $Lazy_L$ follow $\mathrm{SP}_2$. Lemma 8 shows that they also fulfill $\mathrm{SP}_3$, Lemma 10 proves they verify $\mathrm{SP}_4$, and Lemma 11 ensures that they respect $\mathrm{SP}_5$.

Finally, Lemma 12 implies that the modify phases of update operations in $Lazy_L$ verify $\mathrm{SP}_6$, Lemma 19 shows they fulfill $\mathrm{SP}_7$, by Lemma 13 they respect $\mathrm{SP}_8$, and by Lemma 21 $\mathrm{SP}_9$. Moreover, Lemma 10 entails that delete operations of $Lazy_L$ verify $\mathrm{SP}_{10}$.

**Theorem 2.** *The original lazy linked list algorithm [17] (denoted with $OLazy_L$) is **not** sequentially proximal with respect to $Seq_L$.*

*Proof.* $OLazy_L$ is not sequential proximal with respect to $Seq_L$ since it does not satisfy $\mathrm{SP}_6$. This means that $OLazy_L$ does not have insert read-only unsuccessful modifications. Consider that we execute solo the following history $S = (p,$ entry insert $v)$, $(p,$ exit insert true$)$, $(p,$ entry insert $v)$, $(p,$ exit insert false$)$. The first insertion returns true since the list does not contain $v$ initially, but the second insert operation is unsuccessful since $v$ resides already in the list. This means that the second unsuccessful insert operation should not issue any write. But $OLazy_L$ first locks a node, and therefore issues a global write (e.g., write or compare-and-swap instruction), and then verifies if $v$ is in the list or not. This means that $OLazy_L$ does not satisfy $\mathrm{SP}_6$ and therefore it is not sequential proximal.

## 6 Harris Linked List Proof

In this section we prove that a slightly modified version (Figure 6.1) of Harris concurrent linked list [16] is sequentially proximal w.r.t. the sequential linked list of Figure 5.1.

The differences between the program presented in Figure 6.1 (denoted $Harris_L$) and the original Harris concurrent linked list [16] are the following: (a) the search operation is modified so that it becomes read-only (in the original algorithm, cleaning-up stores were issued during parse phases) and (b) the parse phases of update operations do not restart on failed cleaning-up stores anymore.

The proof being similar to the one of the lazy linked list in Section 5, we present detailed proofs of lemmas only when they differ from those of this previous proof.

**Lemma 22.** ***Every execution $\pi$ of $Harris_L$ is well-formed.***

*Proof.* Similar arguments as those used in the proof of Lemma 1 apply here.

**Lemma 23.** *$Harris_L$ **has search read-only traversals**: For any search entry transition $t_{en}$ in an execution $\pi \in [\![Lazy_L]\!]$, there is no transition executing a write instruction in any sequence of $traversals(t_{en}, \pi)$.*

*Proof.* The proof is similar to the proof of Lemma 2.

**Lemma 24.** ***The value of a node remains unchanged in** $Harris_L$: The value field of a node after it is inserted in the list never changes.*

```
Node init_G

function init() {
  allocate(head)
  head.value = −∞
  allocate(tail)
  tail.value = +∞
  init_G = head
  init_G.next = tail
}

function insert(v) {
  beg-parse
  Node previous = init_G
  Node current = previous.next
  while (current.value < v) {
    previous = current
    current = current.next
  }
  pr = (current.value ≠ v)
  end-parse pr
  if (not pr)
    return false

  beg-modify
  allocate(newNode)
  newNode.value = v
  newNode.next = current
  previous.next = newNode (GW)
  end-modify true
  return true
}
```

**Algorithm 1.1.** Linked list with no global lock during traversals.

```
function search(v) {
  Node current = init_G
  while (current.value < v) {
    current = current.next
  }
  return current.value = v
}

function delete(v) {
  beg-parse
  Node previous = init_G
  Node current = previous.next
  while (current.value < v) {
    previous = current
    current = current.next
  }
  pr = (current.value = v)
  end-parse pr
  if (not pr)
    return false

  beg-modify
  previous.next = current.next (GW)
  end-modify true
  return true
}
```

**Algorithm 1.2.** Linked list with no global lock during traversals.

**Fig. 5.1.** Sequential Linked List.

*Proof.* The same arguments as in the proof of Lemma 3 apply here.

**Lemma 25.** ***Nodes are stored in increasing order of values in*** *$Harris_L$: Given two nodes $n_1$ and $n_2$ in $Harris_L$ such that the* next *field of $n_1$ points to $n_2$ then $n_1$.value$<$ $n_2$.value at any point during an execution of $Harris_L$.*

*Proof.* The proof of this lemma is similar to the proof of Lemma 25. Meaning that we initially show that the nodes are stored in increasing order and then show that the inequality still holds after an insert or delete operation is applied.

Initially, after the execution of the init operation, it is the case that head.next $=$ tail and head.value $= -\infty < +\infty =$ tail.value, so the inequality holds.

The searchHelper always returns a pair of nodes (left˙node, right˙node) such that following at least one next pointer from *left_node* will lead to *right_node*. The pair of nodes (left˙node, right˙node) returned by searchHelper satisfies the inequality left˙node.value ¡ right˙node.value. searchHelper stops when it reads something greater than the searched value v. It is then the case that left˙node.value ¡ v $\leq$ right˙node.value.

We now examine all the cases where a next field is being modified. We first check the case of an insert operation. An insert issues a compare-and-swap instruction at line 47, where it atomically does two operations: checks if previous.next points to current and, if this is the case, makes previous.next to point to newNode. Furthermore newNode.next already points to current due to the assignment at line 45. Since newNode.value corresponds to $v$ that is between previous.value and current.value, the inequality still holds.

The case of a delete operation is similar. At line 84 a physical removal that removes the node current from the list takes place. Since previous.value $\leq$ current.value $\leq$ current.next.value, after the deletion it holds that previous.next $=$ current and obviously still previous.value $\leq$ current.value. The reason is that the compare-and-swap instruction is atomic. Similarly to the delete operation, after a successful CAS write at line 26 of searchHelper, the inequality still holds.

**Lemma 26.** *$Harris_L$ **has search no back-step traversals**: For any execution $\pi \in [\![Prog]\!]$, for any search entry transition $t_{en}$ taken by process $p$ in $\pi$, in every sequence in $traversals(t_{en}, \pi)$, process $p$ has no back-steps in $trav$.*

*Proof.* Using the same argument as in Lemma 6 and applying Lemma 25 is enough to achieve this proof.

**Lemma 27.** *$Harris_L$ **has search non-blocking traversals**: There exists an $n \in \mathbb{N}$ such that for any execution $\pi \in [\![Prog]\!]$, for any search entry transition $t_{en}$ taken in $\pi$ by a process $p$, $p$ is $n$ steps non-blocking in every sequence of $traversals(t_{en}, \pi)$.*

*Proof.* The same reasoning as in Lemma 7 applies here by taking $n = 2$ and by using Lemma 25 instead of Lemma 5.

**Lemma 28.** *$Harris_L$ **has no insert or delete back-steps during a parse phase in** $Lazy_L$: For any execution $\pi \in [\![Prog]\!]$, for any update entry transition $t_{en}$ taken by process $p$ in $\pi$, $p$ has no backsteps in every sequence of $traversals(t_{en}, \pi)$.*

*Proof.* The arguments of Lemma 8 apply to this proof, here again by replacing the use of Lemma 5 by its counterpart for $Harris_L$, Lemma 25.

**Lemma 29.** *$Harris_L$ **has insert and delete non-blocking traversals**: There exists an $n \in \mathbb{N}$ such that for any execution $\pi \in [\![Prog]\!]$, for any update entry transition $t_{en}$ taken by a process $p$ in $\pi$, $p$ is $n$ steps non-blocking in every sequence of $traversals(t_{en}, \pi)$.*

*Proof.* The same reasoning as in Lemma 9 applies here by using Lemma 25. Different nodes are traversed during the searchHelper operation, following next fields until the tail of the list is read. The operation then exits the while loop. Other than this while loop, there is no possible place for blocking inside a traversal, in either insert or delete operations.

**Lemma 30.** *$Harris_L$ **has no allocation traversals and no allocation modifications**: For any search transition $t_{en}$ taken in $\pi$, there is no transition executing an allocate instruction in any sequence of $traversals(t_{en}, \pi)$. Furthermore, for any transition $t_{en}$ taken in $\pi$ that executes entry delete, there is no transition executing an allocate instruction in any sequence of $modifications(t_{en}, \pi)$.*

*Proof.* Similarly to the case of Lemma 10, the proof is done by checking that the functions insert and delete do not contain any allocate instruction.

**Lemma 31.** *$Harris_L$ **has insert and delete read-clean traversals**: For every update entry transition $t_{en}$ in $\pi \in [\![Harris_L]\!]$, if a transition $t_w$ executes a write instruction in a sequence of $traversals(t_{en}, \pi)$, $t_w$ is a cleaning-up store.*

*Proof.* As in the proof of Lemma 11, it is enough to verify that the only global write or compare-and-swap instructions that are executed between a beg-parse and an end-parse statement are cleaning-up stores. It can be verified by remarking that the only global instruction executed in parse phases is the compare-and-swap instruction of line 26 in the auxiliary function searchHelper. This instruction only takes place on logically deleted (marked) nodes and, when it succeeds, it replaces left.next, the only pointer that makes right reachable. It is consequently a cleaning-up store.

**Lemma 32.** *$Harris_L$* ***has*** **insert** ***and*** **delete** ***read-only unsuccessful modifications****: For any complete sequential history $S \in Spec_{SDS}$ and any sequence of processes $P$, the solo execution $\pi = se(S, Prog, P)$ verifies that: For every entry op transition $t_{en}$ in $\pi$ that has a matching exit op $false$ statement in $hs(\pi)$, it is the case that $modifications(t_{en}, \pi) = \emptyset$.*

*Proof.* Similarly to the case of Lemma 12. Since restarts can happen only due to a concurrent operation taking place (Lemma 36), there are no restarts in solo executions. Furthermore, a modify phase either restarts or returns true. false is only returned at line 40 and line 71 of the parse phase of an insert or a delete operation respectively. Therefore, an unsuccessful operation never enters the modification phase.

**Lemma 33.** ***Number of stores****: $Harris_L$ has a sequential number of stores per modification, with respect to $Seq_L$.*

*Proof.* Table 3 displays the number of write and compare-and-swap operations executed by insert and delete operations. Comparing these numbers to those of $Seq_L$ appearing in Table 1 allows to conclude that the relations of property $\mathrm{SP}_8$ are verified.

| $Harris_L$ **Number of Stores and Compare-and-Swaps** | |
|---|---|
| **insert** | $\lvert CASOps(modi) \rvert = 1$ |
| | $\lvert OtherNodeWrites(modi, \pi) \rvert = 0$ |
| **delete** | $\lvert CASOps(modi) \rvert = 2$ |
| | $\lvert OtherNodeWrites(modi, \pi) \rvert = 0$ |

**Table 3.** Number of stores and compare-and-swap operations by Harris linked list.

**Lemma 34.** ***No marked or locked nodes exist in a steady state in*** *$Harris_L$: In any steady state $\pi \in [\![Harris_L]\!]$ that is produced by a solo execution, there is no reachable marked or locked node.*

*Proof.* First, note that $Harris_L$ does not use locks. To show that no node is marked between operations of a solo execution, remark that in the absence of concurrency, the two compare-and-swap instructions of lines 76 and 84 cannot fail. The deleted node is consequently properly unlinked during each delete operation of a solo execution.

**Lemma 35.** ***Each node is associated with exactly one value in*** *$Harris_L$: There is a one to one correspondence between a node and its value.*

*Proof.* The proof is the same as the one of Lemma 17.

**Lemma 36.** *$Harris_L$* ***has valid conflict restart modifications, with respect to*** *$Seq_L$: For any complete sequential history $S'$ with at least four tuples, $(S', Seq_L, Harris_L)$ is a valid restart triple.*

*Proof.* This proof is similar as the one of Lemma 19. We assume by the way of contradiction that a complete sequential history $S' = S, en_0, ex_0, en_1, ex_1$ exists such that $(S', Seq_L, Harris_L)$ is not a valid restart triple. Similar to Lemma 19, we assume that the transitions $t_{en_0}$ and $t_{en_1}$ that correspond to the entry statements $en_0$ and $en_1$ respectively, are conflict-free in the solo execution of $se(S', Seq_L, P_S)$. Therefore, we can argue that the operations write to nodes with different values, implying that by executing the operations $en_0$ and $en_1$ concurrently in $Harris_L$ will lead to writes to disjoint nodes. Therefore, no operation can alter the behaviour of the other, which means there are no restarts. A contradiction.

**Lemma 37.** ***Nodes created by solo executions of*** $Harris_L$ ***and*** $Seq_L$ ***that contain the same value, are associated with the same relative node:*** *Consider a sequential history* $S$ *that is executed solo by* $Harris_L$ *and* $Seq_L$*. At the end of the execution,* $Harris_L$ *contains a node* $n_{Harris}$ *that is reachable from the head of the list generated by* $Harris_L$*. While* $Seq_L$ *contains a node* $n_{Seq}$ *that is reachable from the head of the list generated by* $Seq_L$*. Consider that the node* $n_{Harris}$ *is associated with the relative node* $rn_{Harris}$*, while the node* $n_{Seq}$ *is associated with the relative node* $rn_{Seq}$*. If* $n_{Harris}.value = n_{Seq}.value$ *then* $rn_{Harris} = rn_{Seq}$*.*

*Proof.* The same reasoning as the one of the proof of Lemma 20 applies here, using Lemma 34 instead of Lemma 16.

**Lemma 38.** ***Region of stores per modification****:* $Harris_L$ *has a valid region of stores per modification with respect to* $Seq_L$*.*

*Proof.* Remember that the region of stores applies only for solo execution. Meaning that the same complete sequential history is executed in both $Harris_L$ and $Seq_L$. Since we are talking about solo executions there are no marked nodes in the list of $Harris_L$ due to Lemma 34. Similarly to Lemma 21, we argue that parse phases of update operations are going to stop at the same node in both $Harris_L$ and $Seq_L$, and therefore they are going to write similar nodes. Note that as explained in Lemma 21, because the executions are solo, update operations are executing at most one modify phase, so the written nodes of all modifications correspond to the written nodes of at most one modification. The exact same argument applies to $Harris_L$.

**Theorem 3.** $Harris_L$ *is sequentially proximal with respect to* $Seq_L$*.*

*Proof.* Lemma 23 entails that search operations of $Harris_L$ follow $SP_1$. Lemma 27 implies they respect $SP_2$, Lemma 26 shows that they fulfill $SP_3$, while Lemma 30 proves that they respect $SP_4$.

By Lemma 29, the parse phases of update operations in $Harris_L$ follow $SP_2$. Lemma 28 shows that they also fulfill $SP_3$, Lemma 30 proves they verify $SP_4$, and Lemma 31 ensures that they respect $SP_5$.

Finally, Lemma 32 implies that the modify phases of update operations in $Harris_L$ verify $SP_6$, Lemma 36 shows they fulfill $SP_7$, by Lemma 33 they respect $SP_8$, and by Lemma 38 $SP_9$. Moreover, Lemma 30 entails that delete operations of $Harris_L$ verify $SP_{10}$.

**Theorem 4.** *The original* $Harris_L$ *algorithm [16] (denoted with* $OHarris_L$*) is* **not** *sequentially proximal with respect to* $Seq_L$*.*

*Proof.* $OHarris_L$ does not satisfy property $SP_1$ and therefore it is not sequential proximal. Specifically, $OHarris_L$ does not have search read-only traversals since during a search operation it could possibly issue writes. The writes are issued for cleaning-up purposes.

Additionally, in $OHarris_L$ update operations restart their parse phase if they fail a cleaning-up store. This entails that $OHarris_L$ does not verify $SP_3$ that forbids to visit several times the same shared memory location of a node if another node is accessed in between.

## 7 Sequential Proximity in Action

We illustrate the usefulness of sequential proximity (SP). We start by showing how we can create a scalable linked list using SP. We then present a table that includes 24 state-of-the-art algorithms with details about which SP properties each algorithm follows. Finally, we discuss specific examples of algorithms violating each SP property.

### 7.1 Using SP to Design a Linked List

The simplest concurrent linked list is a sorted sequential list protected by a global lock. We show that such an algorithm does not satisfy most SP properties. By fixing those SP properties, following simple steps, we gradually improve the scalability of the linked list. The end result is an SP-compliant highly-scalable algorithm. For simplicity, we omit memory reclamation in our algorithms.

**Introducing Global Lock.** Our first concurrent linked list corresponds to a sequential linked list augmented with a global lock (Algorithm 1.7). As the name suggests, the sequential_search and sequential_parse functions correspond to sequential implementations for search and parse, respectively. Both these functions traverse the list looking for the target value v, and return true if v is found or false otherwise. sequential_parse additionally returns two pointers to nodes, previous and current, such that the node that corresponds to previous points to current during traversal. Furthermore, if value v is found, node current contains it, otherwise value v is in-between the values of nodes previous and current. We can easily prove that this algorithm does not satisfy $SP_1$ since memory is written during traversals (e.g., write at line 2). It does not satisfy $SP_2$ since a traversal needs to grab the lock and subsequently wait until a lock is released (e.g., line 10).

Furthermore, $SP_3$ is not satisfied because glock is read at the beginning of an operation and later, after traversing other nodes, glock is accessed again. $SP_4$ is satisfied since no allocation takes place during traversals. $SP_5$ is not satisfied since non-cleaning-up writes are issued during traversals (writes to glock). $SP_6$ is not satisfied since an insert or a delete operation issues a store even if the operation returns false. $SP_7$ is satisfied since the algorithm has no restarts. $SP_8$ is satisfied since there is at most one lock acquisition (glock) and one write both while inserting and deleting. $SP_9$ is not satisfied: Assume that an insertion takes place between two nodes a and b. In this case the insertion is going to write to node a, as well as the node that contains glock. In contrast, a standard sequential linked list would have only written to node a. Finally, $SP_{10}$ is satisfied since no memory is allocated during deletions.

```
1  function search(v) {
2    lock(glock)
3    res = sequential_search(v)
4    unlock(glock)
5    return res
6  }
7
8  function insert(v) {
9    beg-parse
10   lock(glock)
11   (res, previous, current) = sequential_parse(v)
12   if (res) unlock(glock)
13   end-parse (not res)
14   if (res) return false
15
16   beg-modify
17   allocate(n)
18   n.value = v; n.next = current; previous.next = n
19   unlock(glock)
20   end-modify true
21   return true
22 }
23
24 function delete(v) {
25   beg-parse
26   lock(glock)
27   (res, previous, current) = sequential_parse(v)
28   if (not res) unlock(glock)
29   end-parse res
30   if (not res) return false
```

```

  beg-modify
  previous.next = current.next
  unlock(glock)
  end-modify true
  return true
}
```

**Algorithm 1.7.** Linked list–global lock.

```
function insert(v) {
start:
  beg-parse
  vn = glock.version
  (res, previous, current) = sequential_parse(v)
  end-parse (not res)
  if (res) return false

  beg-modify
  lock(glock)
  if (vn ≠ glock.version − 1)
    mr = restart
  else
    allocate(n)
    n.value = v; n.next = current; previous.next = n
    mr = true
  unlock(glock)
  end-modify mr

  if (mr = restart) goto start
  else return true
}

function delete(v) {
start:
  beg-parse
  vn = glock.version
  (res, previous, current) = sequential_parse(v)
  end-parse res
  if (not res) return false

  beg-modify
  lock(glock)
  if (vn ≠ glock.version − 1)
    mr = restart
  else
    previous.next = current.next
  unlock(glock)
  end-modify mr

  if (mr = restart) goto start
  else return true
}
```

**Algorithm 1.8.** Linked list–Lock-free traversals.

```
function validate(p, c) {
```

```
  return (not p.marked) ∧ (p.next = c)
}

function insert(v) {
  beg-parse
  (res, previous, current) = sequential_parse(v)
  end-parse (not res)
  if (res) return false

  beg-modify
  lock(previous.lockf)
  if (not validate(previous, current))
    mr = restart
  else
    allocate(n)
    n.value = v; n.next = current; previous.next = n
    mr = true
  unlock(previous.lockf)
  end-modify mr

  if (mr = restart) goto start
  else return true
}

function search(v) {
  Node current = head_G
  while (current.value < v)
    current = current.next

  return (current.value = v) ∧ (not current.marked)
}

function delete(v) {
  beg-parse
  (res, previous, current) = sequential_parse(v)
  end-parse res
  if (not res) return false

  beg-modify
  lock(previous.lockf)
  lock(current.lockf)
  if (not validate(previous, next))
    mr = restart
  else
    current.marked = true
    previous.next = current.next
    mr = true
  unlock(current.lockf)
  unlock(previous.lockf)
  end-modify mr

  if (mr = restart) goto start
  else return true
}
```

**Algorithm 1.9.** Linked list–Fine-grained locking.

**Fixing $SP_1$.** Algorithm 1.7 does not satisfy (among others) $SP_1$. Search operations do not apply any modifications, thus we can remove the acquisition and release of the lock from the search operation. The algorithm can be proven correct since if a search operation finds an element that was just removed, the deletion was concurrent with the search and the order of their linearization can be fixed. Not acquiring the lock additionally fixes $SP_2$ for search operations. Still, our new algorithm satisfies $SP_1$ but does not overall satisfy $SP_{2-3}$, $SP_{5-6}$, and $SP_9$.

**Fixing $SP_2$, $SP_5$, and $SP_6$.** To fix $SP_2$, we remove the global lock from traversals. As a result, insertions and deletions acquire the global lock only in the modify phase. This modification introduces the following problem: If an update wants to modify node a, between accessing a in the parse phase and locking a, another process can modify a. In order to solve this problem, we augment the global lock with a version number that is incremented whenever the lock is acquired (based on the idea of OPTIK locks [15]) as seen in Algorithm 1.8. We can detect whether there were any modifications on the data structure by comparing the current version number with the version that is read in the beginning of the parse phase. We can prove that this new algorithm satisfies $SP_2$. It also satisfies $SP_5$ since no writes are issued during traversals anymore and $SP_6$ because if an operation is going to return false it does not issue any write. However, this algorithm does not satisfy $SP_7$ because an update operation can restart due to a modification in a totally unrelated part of the list.

**Fixing $SP_3$, $SP_7$, and $SP_9$.** $SP_3$, $SP_7$ and $SP_9$ are not satisfied due to the global lock/version. To avoid using a global lock, we introduce per-node locks (field lockf). Furthermore, we introduce a marked field in our nodes (an idea taken from Harris [16] and the lazy linked list [17]). Before a node is actually removed from our list, we first mark it and then physically excise it from the list. To check if a node is actually in the list, we can now just check if it is marked or not. Algorithm 1.9 implements these changes. The validate operation has been introduced to check if the nodes returned by sequential˙parse are still in the list and previous points to current. Furthermore, the delete operation contains an extra statement at line 46 for marking the node to be deleted. Both insert and delete operations lock the node that is going to have its next field modified. Deletes also lock the node to be deleted in order to mark it. This new algorithm avoids spurious restarts and therefore satisfies $SP_7$. The resulting algorithm is almost identical to the lazy linked list by Heller et al. [17]. Still, the original lazy algorithm might acquire the lock(s) although the operation is doomed to fail, violating $SP_6$.

**Experimental Results** Figure 7.2 compares the linked lists we optimize with SP to the classic lazy linked list (LAZY) [17]. Clearly, each SP property brings significant scalability benefits. Fixing $SP_{1-2}$ (GL-$SP_1$) transforms the lock-based search operation to wait-free and brings important performance benefits. Still, update operations are fully serialized behind the global lock. GL-$SP_{2,5,6}$ improves over GL-$SP_1$ by additionally offering wait-free parsing. However, the global lock for modifications and the spurious restarts still limit scalability. The SP-compliant linked list (FG-SP) solves all the aforementioned problems and offers good scalability. FG-SP performs better than LAZY due to $SP_6$: In contrast to LAZY, FG-SP returns without locking when the operation cannot be performed.

### 7.2 The Road to SP CSDSs

Table 4 includes 24 CSDS algorithms, sorted by their release year. This table also shows which and how many (column ✓ of the table) SP properties each algorithm satisfies.[3] Clearly, over the years, there has been a tendency towards algorithms that are either sequential proximal, or satisfy most SP properties. As a rule of thumb, newer algorithms are more scalable than the older ones of the same type. Additionally, prior work [9, 15] has experimentally shown that, indeed, the SP-compliant data structures in Table 4 are more scalable than the rest. Consequently, the tendency towards SP-compliant algorithms goes hand in hand with better scalability. In what follows, we describe this tendency for linked lists and skip lists.

[3] With regard to “standard” baseline sequential algorithms–see Figure 5.1.

| | | | | | SP Property | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Year** | **DS** | **Type** | **Conf.** | **Ref.** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | ✓ | **Important characteristic(s)** |
| 1990 | LL | lock-based | Tech. Rep. | [35] | ✓ | ✓ | ✗ | ✓ | ✓ | ✗ | ✓ | ✗ | ✓ | ✓ | 7 | Deletions employ pointer reversal so that a traversal always finds a correct path. |
| 1990 | SL | lock-based | Tech. Rep. | [35] | ✓ | ✓ | ✗ | ✓ | ✓ | ✗ | ✓ | ✗ | ✓ | ✓ | 7 | Maintains several levels of [35] lists. |
| 1995 | LL | lock-free | PODC | [37] | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ | ✓ | 8 | One auxiliary node is inserted for every "real" node. |
| 2001 | LL | lock-free | DISC | [16] | ✗ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | 8 | Nodes are deleted in two steps: mark with CAS and delete with a second CAS. |
| 2002 | LL | lock-free | SPAA | [31] | ✗ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | 8 | A refactored implementation of [16] for easier memory management. |
| 2003 | HT | lock-based | – | [26] | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ | ✓ | ✓ | ✗ | ✓ | 7 | Java's *ConcurrentHashMap*. Protects the hash table with a fixed number of locks. |
| 2003 | SL | lock-free | PhD Thesis | [12] | ✗ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | 7 | Optimistically traverses the list and then does CAS at each level (for updates). |
| 2004 | LL | lock-based | – | [33] | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ | ✓ | ✓ | ✗ | ✗ | 4 | Java's *CopyOnWriteArrayList*. Updates are protected by a global lock. |
| 2004 | HT | lock-based | – | [33] | ✓ | ✗ | ✓ | ✓ | ✗ | ✗ | ✓ | ✓ | ✓ | ✗ | 6 | Uses one *CopyOnWriteArrayList* list per bucket, with a single per-bucket lock. |
| 2005 | LL | lock-based | OPODIS | [17] | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | 9 | Nodes are deleted in two steps: marking and physical deletion. |
| 2006 | HT | lock-based | – | [24] | ✗ | ✗ | ✓ | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ | ✓ | 4 | Part of Intel's Thread Building Blocks library. Uses reader-writer locks. |
| 2007 | SL | lock-based | SIROCCO | [22] | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | 10 | Optimistically find the node to update and then acquire the locks at all levels. |
| 2010 | BST | lock-based | PPoPP | [7] | ✓ | ✗ | ✗ | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ | ✓ | 6 | A search/parse can block waiting for a concurrent update to complete. |
| 2010 | BST | lock-free | PODC | [11] | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✗ | 8 | Updates help outstanding operations on the nodes that they intend to modify. |
| 2012 | BST | lock-free | SPAA | [23] | ✗ | ✓ | ✗ | ✗ | ✗ | ✓ | ✓ | ✗ | ✓ | ✗ | 4 | All three operations perform helping and might need to restart. |
| 2014 | BST | lock-based | PPoPP | [10] | ✓ | ✓ | ✗ | ✓ | ✓ | ✗ | ✓ | ✗ | ✓ | ✓ | 7 | Acquires $\geq 3$ locks for removals. Can restart while traversing. |
| 2014 | BST | lock-free | PPoPP | [32] | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | 10 | Marks edges instead of nodes for deletions in order to minimize CAS. |
| 2015 | LL | lock-based | DISC | [14] | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | 10 | Performs fine-grained locking with version-based validation. |
| 2015 | HT | lock-based | ASPLOS | [9] | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | 9 | Takes a snapshot of key/value pairs while traversing and per-bucket locking. |
| 2015 | BST | lock-based | ASPLOS | [9] | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | 10 | Protects each node with a combination of a lock and a version number. |
| 2016 | LL | lock-based | PPoPP | [15] | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✗ | ✓ | 8 | Protects the list with a combination of a global lock and a version number. |
| 2016 | LL | lock-based | PPoPP | [15] | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | 10 | Protects each node with a combination of a lock and a version number. |
| 2016 | HT | lock-based | PPoPP | [15] | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | 10 | Protects each bucket with a combination of a lock and a version number. |
| 2016 | SL | lock-based | PPoPP | [15] | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | 10 | Protects each node with a combination of a lock and a version number. |

**Table 4.** State-of-the-art algorithms for linked lists (LL), hash tables (HT), skip lists (SL), and binary search trees (BST), sorted by release year. The table highlights which and how many (column ✓) SP properties each algorithm satisfies.

**Linked Lists.** Valois [37] introduced the first lock-free linked-list algorithm. This list employs auxiliary nodes in order to avoid concurrency issues, such as the ABA problem [37]. These extra nodes are not allowed by SP for various reasons: (i) the number of writes/atomic operations is large (against $\mathrm{SP}_8$), and (ii) the region of stores is not similar to a standard sequential implementation (violating $\mathrm{SP}_9$).

Harris [16] designed a much simpler lock-free linked list, where instead of extra nodes, updates employ pointer marking to indicate deletion. The resulting algorithm solves the $\mathrm{SP}_{8-9}$ problems of Valois' list, but assigns some cleaning-up tasks to search operations that might therefore write (violating $\mathrm{SP}_1$) and might restart (against $\mathrm{SP}_3$).

Heller et al. [17] recognized Harris list's shortcomings and opted for a "lazy" lock-based design with wait-free traversals (adhering to $\mathrm{SP}_1$ and $\mathrm{SP}_3$). However, updates grab the lock although the operation is doomed to be unsuccessful (e.g., deleting a non-existent key), thus violating $\mathrm{SP}_6$.

David et al. [9] (as well as [14, 15]) recognized and fixed the $\mathrm{SP}_6$ problem of the lazy linked list. The former ([9]) fix the problem directly in the lazy list algorithm, while the latter two ([14, 15]) introduce new list algorithms based on trylocks and version numbers.

**Hash Tables.** The history of the hash tables in Table 4 is not as interesting as the history of lists. Still, the latest two hash tables in the table are almost sequential proximal. Interestingly, the ASPLOS'15 hash table [9] trades $\mathrm{SP}_3$ for $\mathrm{SP}_6$. In short, in paragraph to return without locking in case the operation is not feasible, this hash table performs a read-only parse of the bucket before locking and re-parsing (iff the operation is feasible). Parsing twice violates $\mathrm{SP}_3$.

**Skip lists.** The first concurrent skip-list design by Pugh [35] might acquire a large number of locks, thus violating $\mathrm{SP}_8$. Additionally, the traversal path of a search or an update might back-step due to concurrency, violating $\mathrm{SP}_3$. Finally, failed updates acquire locks (against $\mathrm{SP}_6$).

Fraser [12] designed a lock-free skip list that solves some of the issues of Pugh's algorithm, but introduces others. Similarly to Harris' list [16], Fraser's skip list marks pointers for deletion and might perform cleaning-up of those marked nodes while searching (violating $\mathrm{SP}_1$). If cleaning-up fails, the operations are restarted (violating $\mathrm{SP}_3$).

Herlihy et al. [22] recognized and solved the shortcomings of Fraser's skip list with a new lock-based algorithm that adheres SP. Guerraoui and Trigonakis [15] introduced another SP-compliant skip list based on version-number validation.

### 7.3 Violating SP

The negative scalability effect of violating an SP property depends (i) on the property, and (ii) on the way it is violated. For example, violating $\mathrm{SP}_9$ because of a global lock is much worse for scalability than just writing on a node that is one hop away than the nodes that should be normally written. We illustrate these differences through various examples.

**$\mathbf{SP}_1$.** Intel's TBB hash table [24] protects search operations with reader-writer locks (translates to writing). This violation is more heavyweight than Harris' linked list [16] that might infrequently write to the list for cleaning up.

**$\mathbf{SP}_2$.** Again, Intel's TBB hash table [24] might block waiting for the lock, which is more heavyweight than the infrequent waiting in the BST by Bronson et al. [7].

**$\mathbf{SP}_3$.** Double parsing in the hash table by David et al. [9] is more lightweight than potential restarts after traversing a large list in linked lists (e.g., [16, 31]).

**$\mathbf{SP}_4$.** The lock-free tree by Howley and Jones [23] performs helping in all operations and therefore might need to allocate nodes or help records while traversing. *CopyOnWriteArrayList* [33] inherently requires the allocation of a new copy of the data structure on every update.

**$\mathbf{SP}_5$.** The lock-free tree by Howley and Jones [23] performs heavyweight helping while parsing the list. Still, this helping is lighter than acquiring the lock before traversing the list as in Java's *CopyOnWriteArrayList* list [33].

**$\mathbf{SP}_6$.** Many algorithms (e.g., [17, 26, 35]) acquire locks although the operation is doomed to fail. In hash tables, such as [26], violating $\mathrm{SP}_6$ is more problematic than in lists, such as [17, 35], because the operations are much shorter.

**$\mathbf{SP}_7$.** Most algorithms restart "appropriately." Only the global-lock-based with version validation list [15] can restart due to unrelated modifications in the list.

**$\mathbf{SP}_8$.** The list by Pugh [35] employs pointer reversal and might thus acquire more locks than allowed. This $\mathrm{SP}_8$ violation is far less expensive than the BST from by Drachsler et al. [10] that acquires more than three locks per update.

**$\mathbf{SP}_9$.** $\mathrm{SP}_9$ is often violated due to the granularity of locks. For example, Table 4 includes list algorithms that use global locks [15, 33]. This violation is more problematic than other algorithms, such as Java's *ConcurrentHashMap* [26], that employ lock striping.

**$\mathbf{SP}_{10}$.** Allocations during delete operations are typically due to helping: The operation creates a "help record" to be inserted in the set so that others can later help (e.g., [11, 23]).

```
Node init_G

function init() {
  allocate(head)
  head.value = −∞
  allocate(tail)
  tail.value = +∞
  init_G = head
  init_G.next = tail
}

function validate(previous, current) {
 return (not previous.marked)∧(not current.marked)∧
  (previous.next = current)
}

function insert(v)} {
  restart:
    beg-parse
    Node previous = init_G
    Node current = previous.next
    while (current.value < v) {
      previous = current
      current = current.next
    }
    pr = (current.value≠v∨current.marked∨
      previous.marked)
    end-parse pr
    if (not pr)
      return false

    beg-modify
    lock(previous.lockf)
    if (validate(previous, current)) {
      if (current.value = v)
        mr = false
      else
        allocate(newNode)
        newNode.value = v
        newNode.next = current

        previous.next = newNode    (GW)

        mr = true
    }
    else {
      mr = restart
    }
    unlock(previous.lockf)
    end-modify mr
    if (mr = restart) goto restart
    else return mr
}
```

**Algorithm 1.3.** Linked list with no global lock during traversals.

```
function search(v) {
  Node current = init_G
  while (current.value < v) {
    current = current.next
  }
  return (current.value = v)∧(not current.marked)
}

function delete(v) {
  restart:
    beg-parse
    Node previous = init_G
    Node current = previous.next
    while (current.value < v) {
      previous = current
      current = current.next
    }
    pr = (current.value = v∨current.marked∨
      previous.marked)
    end-parse pr
    if (not pr)
      return false

    beg-modify
    lock(previous.lockf)
    lock(current.lockf)
    if (validate(previous, current)) {

      if (v≠current.value) {
        mr = false
      }
      else {
        current.marked = true (GW)
        previous.next = current.next (GW)
        mr = true
      }
    }
    else {
      mr = restart
    }
    unlock(current.lockf)
    unlock(previous.lockf)
    end-modify mr
    if (mr = restart) goto restart
    else return mr
}
```

**Algorithm 1.4.** Linked list with no global lock during traversals.

**Fig. 5.2.** Concurrent Lazy Linked List.

```
Node head_G
Node tail_G

function init() {
  allocate(head_G)
  head_G.value = −∞
  allocate(tail_G)
  tail_G.value = +∞
  head_G.next = tail_G
}

function searchHelper(v) {
  Node left_node = head_G
  Node right_node = head_G.next

  while(true) {
    if (right_node.next is not marked) {
        if (right_node.value ≥ v) {
            break
        }
        left_node = right_node
    }
    else {
        // unmarked right_node
        unm_right = unmarked(right_node.next)
        CAS(&left_node.next, right_node, unm_right)(GW)
    }

    right_node = unmarked(right_node.next)
  }
  return (left_node, right_node)
}

function insert(v) {
 restart:
  beg-parse
  (previous, current) = searchHelper(v)
  pr = (current = tail_G) ∨ (current.value ≠ v)
  end-parse pr
  if (not pr) return false

  beg-modify
  allocate(newNode)
  newNode.value = v
  newNode.next = current

  if(CAS(&previous.next, current, newNode)= current)(GW)
      mr = true
  else
      mr = restart
  end-modify mr
  if (mr = restart) goto restart
  else return mr
}
```

**Algorithm 1.5.** Linked list with no global lock during traversals.

```
function search(v) {
  Node current = head_G

  while (current.value < v) {
      current = unmarked(current.next)
  }
  return (current.value = v)∧
         (current.next is not marked)
}

function delete(v) {
 restart:
  beg-parse
  (previous, current) = searchHelper(v)
  pr = (current≠tail)∧(current.value = v)
  end-parse pr
  if (not pr) return false

  beg-modify
  tmp = current.next
  if (tmp is not marked) {
    if!(CAS(&current.next,tmp,marked(tmp)) = tmp)(GW)
        mr = true
    else
        mr = restart
  }
  else
    mr = restart

  CAS(&previous.next, current, tmp) (GW)
  end-modify mr
  if (mr = restart) goto restart
  else return mr
}
```

**Algorithm 1.6.** Linked list with no global lock during traversals.

**Fig. 6.1.** Harris Concurrent Linked List.

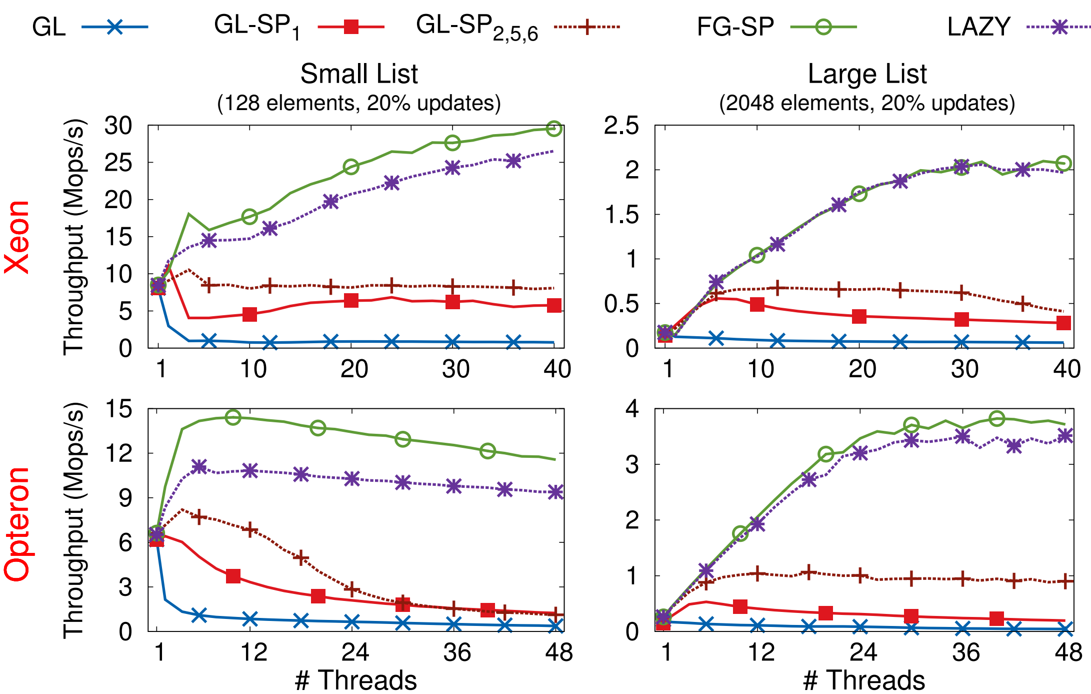


**Fig. 7.2.** Comparing the various linked lists of Section 7.1 on Xeon and Opteron. Xeon is a 2-socket, 20-core (40 hardware contexts) Intel E5-2680 v2 multi-core, while Opteron is a 48-core AMD multi-core with four 6172 Opteron multi-chip modules. Each data point is the median of 11 repetitions of 5 seconds each. We collect data points with 1, 2, 4, 6, ... threads. Threads execute in a loop; at every iteration each thread randomly chooses an operation based on the read/update ratio (updates are split 50/50 between insertions and deletions). At each iteration, the threads also randomly choose a key to operate on (the key range is twice the initial size of the list). Due to this experimental configuration, (i) roughly half of the updates are unsuccessful, and (ii) the size of the list remains close to the initial throughout the experiment. The global-lock lists are protected by a scalable MCS lock [30].

## Bibliography


[1] D. Alistarh, K. Censor-Hillel, and N. Shavit. Are Lock-free Concurrent Algorithms Practically Wait-free? STOC '14.

[2] K. Antoniadis, R. Guerraoui, J. Stainer, and V. Trigonakis. Sequential proximity: Towards provably scalable concurrent search algorithms. NETYS' 17.

[3] M. Arbel and H. Attiya. Concurrent Updates with RCU: Search Tree As an Example. PODC '14.

[4] M. Arbel and A. Morrison. Predicate RCU: An RCU for Scalable Concurrent Updates. PPoPP '15.

[5] A. Atalar, P. Renaud-Goud, and P. Tsigas. Analyzing the Performance of Lock-Free Data Structures: A Conflict-Based Model. DISC '15.

[6] H. Attiya, R. Guerraoui, D. Hendler, P. Kuznetsov, M. M. Michael, and M. T. Vechev. Laws of Order: Expensive Synchronization in Concurrent Algorithms Cannot be Eliminated. POPL '11.

[7] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. A Practical Concurrent Binary Search Tree. PPoPP '10.

[8] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler. The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors. SOSP '13.

[9] T. David, R. Guerraoui, and V. Trigonakis. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures. ASPLOS '15.

[10] D. Drachsler, M. Vechev, and E. Yahav. Practical Concurrent Binary Search Trees via Logical Ordering. PPoPP '14.

[11] F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel. Non-blocking Binary Search Trees. PODC '10.

[12] K. Fraser. *Practical Lock-Freedom*. PhD thesis, University of Cambridge, 2004.

[13] J. Gibson and V. Gramoli. Why Non-blocking Operations Should be Selfish. DC '15.

[14] V. Gramoli, P. Kuznetsov, S. Ravi, and D. Shang. Brief Announcement: A Concurrency-Optimal List-Based Set. DISC '15.

[15] R. Guerraoui and V. Trigonakis. Optimistic Concurrency with OPTIK. PPoPP '16.

[16] T. Harris. A Pragmatic Implementation of Non-blocking Linked Lists. DISC '01.

[17] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. N. Scherer, and N. Shavit. A Lazy Concurrent List-Based Set Algorithm. OPODIS '05.

[18] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat Combining and the Synchronization-parallelism Tradeoff. SPAA '10.

[19] M. Herlihy. Wait-Free Synchronization. *TOPLAS '91*.

[20] M. Herlihy and J. Moss. Transactional Memory: Architectural Support for Lock-free Data Structures. ISCA '93.

[21] M. Herlihy and J. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *TOPLAS '90*.

[22] M. Herlihy, Y. Lev, V. Luchangco, and N. Shavit. A Simple Optimistic Skiplist Algorithm. SIROCCO '07.

[23] S. V. Howley and J. Jones. A Non-blocking Internal Binary Search Tree. SPAA '12.

[24] Intel. Intel Thread Building Blocks. https://www.threadingbuildingblocks.org.

[25] A. Israeli and L. Rappoport. Disjoint-Access-Parallel Implementations of Strong Shared Memory Primitives. PODC '94.

[26] D. Lea. Overview of Package util.concurrent Release 1.3.4. http://gee.cs.oswego.edu/dl/classes/EDU/oswego/cs/dl/util/concurrent/intro.html, 2003.

[27] J.-P. Lozi, F. David, G. Thomas, J. Lawall, and G. Muller. Remote Core Locking: Migrating Critical-section Execution to Improve the Performance of Multithreaded Applications. ATC '12.

[28] A. Matveev, N. Shavit, P. Felber, and P. Marlier. Read-Log-Update: A Lightweight Synchronization Mechanism for Concurrent Programming. SOSP '15.

[29] P. E. McKenney and J. D. Slingwine. Read-copy Update: Using Execution History to Solve Concurrency Problems. PDCS '98.

[30] J. M. Mellor-Crummey and M. L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *TOCS '91*.

[31] M. M. Michael. High Performance Dynamic Lock-free Hash Tables and List-based Sets. SPAA '02.
[32] A. Natarajan and N. Mittal. Fast Concurrent Lock-free Binary Search Trees. PPoPP '14.
[33] Oracle. Java CopyOnWriteArrayList. http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/CopyOnWriteArrayList.html.
[34] C. H. Papadimitriou. The Serializability of Concurrent Database Updates. *JACM '79*.
[35] W. Pugh. Concurrent Maintenance of Skip Lists. Technical report, 1990.
[36] N. Shavit and D. Touitou. Software Transactional Memory. PODC '97.
[37] J. D. Valois. Lock-free Linked Lists Using Compare-and-swap. PODC '95.