In the Search for Optimal Concurrency

Vincent Gramoli[1] Petr Kuznetsov[2] Srivatsan Ravi[3]

[1]Data61-CSIRO and University of Sydney
[2]Télécom ParisTech
[3]Purdue University

Abstract

It is common practice to use the epithet "highly concurrent" referring to data structures that are supposed to perform well in concurrent environments. But how do we measure the concurrency of a data structure in the first place? In this paper, we propose a way to do this, which allowed us to formalize the notion of a *concurrency-optimal* implementation.

The concurrency of a program is defined here as the program's ability to accept concurrent schedules, *i.e.*, interleavings of steps of its sequential implementation. To make the definition sound, we introduce a novel correctness criterion, *LS-linearizability*, that, in addition to classical linearizability, requires that the interleavings of memory accesses to be *locally* indistinguishable from sequential executions. An implementation is then *concurrency-optimal* if it accepts all LS-linearizable schedules. We explore the concurrency properties of *search* data structures which can be represented in the form of directed acyclic graphs exporting insert, delete and search operations. We prove, for the first time, that *pessimistic* (*e.g.*, based on conservative locking) and *optimistic serializable* (*e.g.*, based on serializable transactional memory) implementations of search data-structures are incomparable in terms of concurrency. Thus, neither of these two implementation classes is concurrency-optimal, hence raising the question of the existence of concurrency-optimal programs.

1 Introduction

In the concurrency literature, it is not unusual to meet expressions like "highly concurrent data structures", used as a positive characteristics of their performance. Leaving aside the relation between performance and concurrency, the first question we should answer is what is the concurrency of a data structure in the first place. How do we measure it?

At a high level, concurrency is the ability to serve multiple requests in parallel. A data structure designed for the conventional sequential settings, when used *as is* in a concurrent environment, while being intuitively very concurrent, may face different kinds of inconsistencies caused by races on the shared data. To avoid these races, a variety of synchronization techniques have been developed [9]. Conventional pessimistic synchronization protects shared data with locks before reading or modifying them. Optimistic synchronization, achieved using transactional memory (TM) or conditional instructions such as CAS or LL/SC, optimistically executes memory accesses with a risk of aborting them in the future. A programmer typically uses these synchronization techniques to "wrap" fragments of a sequential implementation of the desired data structure, in order to preserve a correctness criterion.

It is however difficult to tell in advance which of the techniques will provide more concurrency, *i.e.*, which one would allow the resulting programs to process more executions of concurrent operations without data conflicts. Implementations based on TMs [20, 28], which execute concurrent accesses speculatively, may seem more concurrent than lock-based counterparts whose concurrent accesses are blocking. But TMs conventionally impose *serializability* [26] or even stronger properties [15] on operations encapsulated within transactions. This may prohibit certain concurrent scenarios allowed by a large class of dynamic data structures [10].

In this paper, we reason formally about the "amount of concurrency" one can obtain by turning a sequential program into a concurrent one. To enable fair comparison of different synchronization techniques, we (1) define what it means for a concurrent program to be correct regardless of the type of synchronization it uses and (2) define a metric of concurrency. These definitions allow us to compare concurrency properties offered by serializable optimistic and pessimistic synchronization techniques, whose popular examples are, respectively, transactions and conservative locking.

Correctness. Our novel consistency criterion, called *locally-serializable linearizability*, is an intersection of *linearizability* and a new *local serializability* criterion.

Suppose that we want to design a concurrent implementation of a data type $\mathcal{T}$ (*e.g.*, integer set), given its sequential implementation $\mathcal{S}$ (*e.g.*, based on a sorted linked list). A concurrent implementation of $\mathcal{T}$ is *locally serializable* with respect to $\mathcal{S}$ if it ensures that the local execution of *reads* and *writes* of each operation is, in precise sense, equivalent to *some* execution of $\mathcal{S}$. This condition is weaker than serializability since it does not require the existence of a *single* sequential execution that is consistent with all local executions. It is however sufficient to guarantee that executions do not observe an inconsistent transient state that could lead to fatal arithmetic errors, *e.g.*, division-by-zero.

In addition, for the implementation of $\mathcal{T}$ to "make sense" globally, every concurrent execution should be *linearizable* [23, 3]: the invocation and responses of high-level operations observed in the execution should constitute a correct sequential history of $\mathcal{T}$. The combination of local serializability and linearizability gives a correctness criterion that we call *LS-linearizability*, where LS stands for "locally serializable". We show that LS-linearizability, just like linearizability, is *compositional* [23, 21]: a composition of LS-linearizable implementations is also LS-linearizable.

Concurrency metric. We measure the amount of concurrency provided by an LS-linearizable implementation as the set of schedules it accepts. To this end, we define a concurrency metric inspired by the analysis of parallelism in database concurrency control [32, 18] and transactional memory [12]. More specifically, we assume an external scheduler that defines which processes execute which steps of the corresponding sequential program in a dynamic and unpredictable

fashion. This allows us to define concurrency provided by an implementation as the set of *schedules* (interleavings of reads and writes of concurrent sequential operations) it *accepts* (is able to effectively process).

Our concurrency metric is platform-independent and allows for measuring relative concurrency of LS-linearizable implementations using arbitrary synchronization techniques. The combination of our correctness and concurrency definitions provides a framework to compare the concurrency one can get by choosing a particular synchronization technique for a specific data type.

Measuring concurrency: pessimism vs. serializable optimism. We explore the concurrency properties of a large class of *search* concurrent data structures. Search data structures maintain data in the form of a rooted directed acyclic graph (DAG), where each node is a $\langle key, value \rangle$ pair, and export operations *insert*$(key, value)$, *delete*(key), and *find*(key) with the natural sequential semantics. The class includes many popular data structures, such as linked lists, skiplists, and search trees, implementing various abstractions like sets, multi-sets and dictionaries.

In this paper, we compare the concurrency properties of two classes of search-structure implementations: *pessimistic* and *serializable optimistic*. Pessimistic implementations capture what can be achieved using classic conservative locks like mutexes, spinlocks, reader-writer locks. In contrast, optimistic implementations, however proceed speculatively and may roll back in the case of conflicts. Additionally, *serializable* optimistic techniques, *e.g.*, relying on conventional TMs, like TinySTM [8] or NOrec [5] allow for transforming any sequential implementation of a data type to a LS-linearizable concurrent one.

Main contributions. The main result of this paper is that synchronization techniques based on pessimism and serializable optimism, are not concurrency-optimal: we show that no one of their respective set of accepted concurrent schedules include the other.

On the one hand, we prove that there exist simple schedules that are not accepted by *any* pessimistic implementation, but accepted by a serializable optimistic implementation. Our proof technique, which is interesting in its own right, is based on the following intuitions: a pessimistic implementation has to proceed irrevocably and over-conservatively reject a potentially acceptable schedule, simply because it *may* result in a data conflict leading the data structure to an inconsistent state. However, an optimistic implementation of a search data structure may (partially or completely) restart an operation depending on the current schedule. This way even schedules that potentially lead to conflicts may be optimistically accepted.

On the other hand, we show that pessimistic implementations can be designed to exploit the semantics of the data type. In particular, they can allow operations updating disjoint sets of data items to proceed independently and preserving linearizability of the resulting history, even though the execution is not serializable. In such scenarios, pessimistic implementations carefully adjusted to the data types we implement can supersede the "semantic-oblivious" optimistic serializable implementations. Thus, neither pessimistic nor serializable optimistic implementations are concurrency-optimal.

Our comparative analysis of concurrency properties of pessimistic and serializable optimistic implementation suggests that combining the advantages of pessimism, namely its semantics awareness, and the advantages of optimism, namely its ability to restart operations in case of conflicts, enables implementations that are strictly better-suited for exploiting concurrency than any of these two techniques taken individually. To the best of our knowledge, this is the first formal analysis of the relative abilities of different synchronization techniques to exploit concurrency in dynamic data structures and lays the foundation for designing concurrent data structures that are concurrency-optimal.

Roadmap. We define the class of concurrent implementations we consider in Section 2. In Section 3, we define the correctness criterion and our concurrency metric. Section 4 defines the class of data structures for which our concurrency lower bounds apply. In Section 5, we analyse

the concurrency provided by pessimistic and serializabile optimistic synchronization techniques to search data structures. We discuss the related work in Sections 6 and conclude in Section 7.

2 Preliminaries

Sequential types and implementations. An *type* τ is a tuple $(\Phi, \Gamma, Q, q_0, \delta)$ where Φ is a set of operations, Γ is a set of responses, Q is a set of states, $q_0 \in Q$ is an initial state and $\delta \subseteq Q \times \Phi \times Q \times \Gamma$ is a *sequential specification* that determines, for each state and each operation, the set of possible resulting states and produced responses [2].

Any type $\tau = (\Phi, \Gamma, Q, q_0, \delta)$ is associated with a *sequential implementation IS*. The implementation encodes states in Q using a collection of elements $X_1, X_2, \ldots$ and, for each operation of τ, specifies a sequential *read-write* algorithm. Therefore, in the implementation *IS*, an operation performs a sequence of *reads* and *writes* on $X_1, X_2, \ldots$ and returns a response $r \in \Gamma$. The implementation guarantees that, when executed sequentially, starting from the state of $X_1, X_2, \ldots$ encoding q_0, the operations eventually return responses satisfying δ.

Concurrent implementations. We consider an asynchronous shared-memory system in which a set of processes communicate by applying *primitives* on shared *base objects* [19].

We tackle the problem of turning the sequential algorithm *IS* of type τ into a *concurrent* one, shared by n processes $p_1, \ldots, p_n$ ($n \in \mathbb{N}$). The idea is that the concurrent algorithm essentially follows *IS*, but to ensure correct operation under concurrency, it replaces read and write operations on $X_1, X_2, \ldots$ in operations of *IS* with their base-object implementations.

Throughout this paper, we use the term *operation* to refer to high-level operations of the type. Reads and writes implemented by a concurrent algorithm are referred simply as *reads* and *writes*. Operations on base objects are referred to as *primitives*.

We also consider concurrent implementation that execute portions of sequential code *speculatively*, and restart their operations when conflicts are encountered. To account for such implementations, we assume that an implemented read or write may *abort* by returning a special response $\perp$. In this case, we say that the corresponding (high-level) operation is *aborted*.

Therefore, our model applies to all concurrent algorithms in which a high-level operation can be seen as a sequence of reads and writes on elements $X_1, X_2, \ldots$ (representing the state of the data structure), with the option of aborting the current operation and restarting it after. Many existing concurrent data structure implementations comply with this model as we illustrate below.

Executions and histories. An *execution* of a concurrent implementation is a sequence of invocations and responses of high-level operations of type τ, invocations and responses of read and write operations, and invocations and responses of base-object primitives. We assume that executions are *well-formed*: no process invokes a new read or write, or high-level operation before the previous read or write, or a high-level operation, resp., returns, or takes steps outside its operation's interval.

Let $\alpha|p_i$ denote the subsequence of an execution α restricted to the events of process p_i. Executions α and α' are *equivalent* if for every process p_i, $\alpha|p_i = \alpha'|p_i$. An operation π *precedes* another operation π' in an execution α, denoted $\pi \rightarrow_\alpha \pi'$, if the response of π occurs before the invocation of π'. Two operations are *concurrent* if neither precedes the other. An execution is *sequential* if it has no concurrent operations. A sequential execution α is *legal* if for every object X, every read of X in α returns the latest written value of X. An operation is *complete* in α if the invocation event is followed by a *matching* (non-$\perp$) response or aborted; otherwise, it is *incomplete* in α. Execution α is *complete* if every operation is complete in α.

The *history exported by an execution* α is the subsequence of α reduced to the invocations and responses of operations, reads and writes, except for the reads and writes that return $\perp$ (the abort response).

High-level histories and linearizability. A *high-level history* $\tilde{H}$ of an execution α is the subsequence of α consisting of all invocations and responses of *non-aborted* operations. A complete high-level history $\tilde{H}$ is *linearizable* with respect to an object type τ if there exists a sequential high-level history S equivalent to H such that (1) $\rightarrow_{\tilde{H}} \subseteq \rightarrow_S$ and (2) S is consistent with the sequential specification of type τ. Now a high-level history $\tilde{H}$ is linearizable if it can be *completed* (by adding matching responses to a subset of incomplete operations in $\tilde{H}$ and removing the rest) to a linearizable high-level history [23, 3].

Optimistic and pessimistic implementations. Note that in our model an implementations may, under certain conditions, abort an operation: some read or write return $\perp$, in which case the corresponding operation also returns $\perp$. Popular classes of such *optimistic* implementations are those based on "lazy synchronization" [17, 21] (with the ability of returning $\perp$ and re-invoking an operation) or transactional memory (*TM*) [28, 5].

In the subclass of *pessimistic* implementations, no execution includes operations that return $\perp$. Pessimistic implementations are typically *lock-based* or based on pessimistic TMs [1]. A lock provides exclusive (resp., shared) access to an element X through synchronization primitives $\mathsf{lock}(X)$ (resp., $\mathsf{lock\text{-}shared}(X)$), and $\mathsf{unlock}(X)$ (resp., $\mathsf{unlock\text{-}shared}(X)$). A process *releases* the lock it holds by invoking $\mathsf{unlock}(X)$ or $\mathsf{unlock\text{-}shared}(X)$. When $\mathsf{lock}(X)$ invoked by a process p_i returns, we say that p_i *holds a lock on* X (until p_i returns from the subsequent $\mathsf{lock}(X)$). When $\mathsf{lock\text{-}shared}(X)$ invoked by p_i returns, we say that p_i *holds a shared lock on* X (until p_i returns from the subsequent $\mathsf{lock\text{-}shared}(X)$). At any moment, at most one process may hold a lock on an element X. Note that two processes can hold a shared lock on X at a time. We assume that locks are *starvation-free*: if no process holds a lock on X forever, then every $\mathsf{lock}(X)$ eventually returns. Given a sequential implementation of a data type, a corresponding lock-based concurrent one is derived by inserting the synchronization primitives (lock and unlock) to protect read and write accesses to the shared data.

3 Correctness and concurrency metric

In this section, we define the correctness criterion of *locally serializable linearizability (LS-linearizability)* and introduce the framework for comparing the relative abilities of different synchronization technique in exploiting concurrency.

3.1 Locally serializable linearizability

Let H be a history and let π be a high-level operation in H. Then $H|\pi$ denotes the subsequence of H consisting of the events of π, except for the last aborted read or write, if any. Let IS be a sequential implementation of an object of type τ and Σ_{IS}, the set of histories of IS.

Definition 1 (LS-linearizability)**.** *A history H is* locally serializable with respect to *IS if for every high-level operation π in H, there exists $S \in \Sigma_{IS}$ such that $H|\pi = S|\pi$.*

A history H is LS-linearizable with respect to *(IS, τ) (we also write H is (IS, τ)-LSL) if: (1) H is locally serializable with respect to IS and (2) the corresponding high-level history $\tilde{H}$ is linearizable with respect to τ.*

Observe that local serializability stipulates that the execution is seen as a sequential one by every operation. Two different operations (even when invoked by the same process) are not required to witness mutually consistent sequential executions.

A concurrent implementation I is *LS-linearizable with respect to* (IS, τ) (we also write I is (IS, τ)-LSL) if every history exported by I is (IS, τ)-LSL. Throughout this paper, when we refer to a concurrent implementation of (IS, τ), we assume that it is LS-linearizable with respect to (IS, τ).

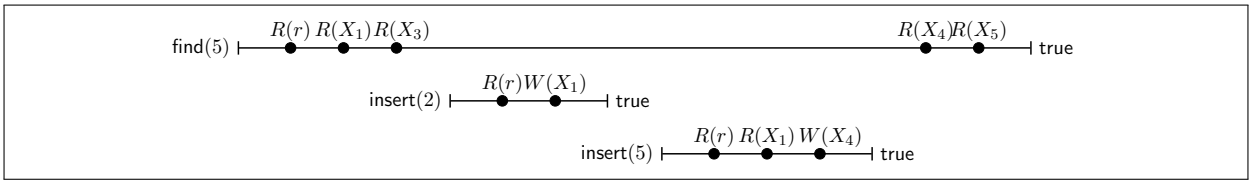

Figure 1: A concurrency scenario for a set, initially $\{1, 3, 4\}$, where value i is stored at node X_i: insert(2) and insert(5) can proceed concurrently with find(5). The history is LS-linearizable but not serializable; yet accepted by HOH-find. (Not all read-write on nodes is presented here.)

We show in Appendix A that just as linearizability, LS-linearizability is *compositional* [23, 21]: a composition of LSL implementations is also LSL. However, LS-linearizability is not non-blocking [23, 21]: local serializability may prevent an operation in a finite LS-linearizable history from having a completion, e.g., because, it might read an inconsistent system state caused by a concurrent incomplete operation.

LS-linearizability and other consistency criteria. LS-linearizability is a two-level consistency criterion which makes it suitable to compare concurrent implementations of a sequential data structure, regardless of synchronization techniques they use. It is quite distinct from related criteria designed for database and software transactions, such as serializability [26, 31] and multilevel serializability [30, 31].

For example, serializability [26] prevents sequences of reads and writes from conflicting in a cyclic way, establishing a global order of transactions. Reasoning only at the level of reads and writes may be overly conservative: higher-level operations may commute even if their reads and writes conflict [29]. Consider an execution of a concurrent *list-based set* depicted in Figure 1. We assume here that the set initial state is $\{1, 3, 4\}$. Operation find(5) is concurrent, first with operation insert(2) and then with operation insert(5). The history is not serializable: insert(5) sees the effect of insert(2) because $R(X_1)$ by insert(5) returns the value of X_1 that is updated by insert(2) and thus should be serialized after it. Operation find(5) misses element 2 in the linked list and must read the value of X_4 that is updated by insert(5) to perform the read of X_5, *i.e.*, the element created by insert(5). This history is, however, LSL since each of the three local histories is consistent with some sequential history of LL.

Multilevel serializability [30, 31] was proposed to reason in terms of multiple semantic levels in the same execution. LS-linearizability, being defined for two levels only, does not require a global serialization of low-level operations as 2-level serializability does. LS-linearizability simply requires each process to observe a local serialization, which can be different from one process to another. Also, to make it more suitable for concurrency analysis of a concrete data structure, instead of semantic-based commutativity [29], we use the sequential specification of the high-level behavior of the object [23].

Linearizability [23, 3] only accounts for high-level behavior of a data structure, so it does not imply LS-linearizability. For example, Herlihy's universal construction [19] provides a linearizable implementation for any given object type, but does not guarantee that each execution locally appears sequential with respect to any sequential implementation of the type. Local serializability, by itself, does not require any synchronization between processes and can be trivially implemented without communication among the processes. Therefore, the two parts of LS-linearizability indeed complement each other.

3.2 Concurrency metric

To characterize the ability of a concurrent implementation to process arbitrary interleavings of sequential code, we introduce the notion of a *schedule*. Intuitively, a schedule describes the order in which complete high-level operations, and sequential reads and writes are invoked by the user. More precisely, a schedule is an equivalence class of complete histories that agree on the *order* of

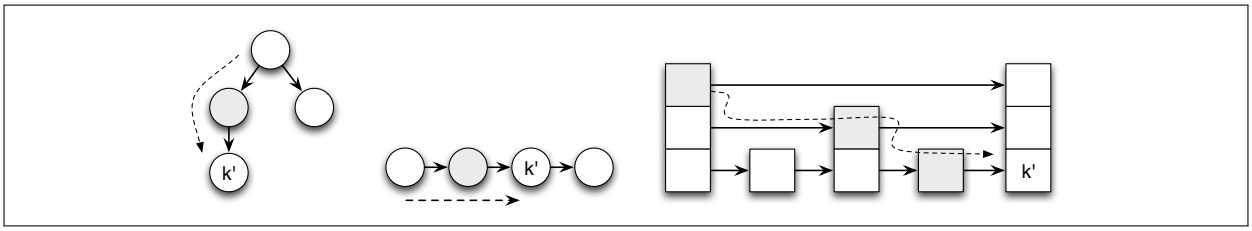

Figure 2: Three search structures, a binary tree, a linked list and a skip list, whose k-relevant set has grey nodes and whose k-relevant path is indicated with a dashed arrow

invocation and response events of reads, writes and high-level operations, but not necessarily on the responses of read operations or of high-level operations. Thus, a schedule can be treated as a history, where responses of read and high-level operations are not specified.

We say that an implementation I *accepts* a schedule σ if it exports a history H such that $complete(H)$ exhibits the order of σ, where $complete(H)$ is the subsequence of H that consists of the events of the complete operations that returned a matching response. We then say that the execution (or history) *exports* σ. A schedule σ is (IS, τ)-LSL if there exists an (IS, τ)-LSL history exporting σ.

An (IS, τ)-LSL implementation is therefore *concurrency-optimal* if it accepts all (IS, τ)-LSL schedules.

4 Search data structures

In this section, we introduce a class $\mathcal{D}$ of *dictionary-search* data structures (or simply *search structures*), inspired by the study of *dynamic databases* undertaken by Chaudhri and Hadzilacos [4].

Data representation. At a high level, a search structure is a dictionary that maintains data in a directed acyclic graph (DAG) with a designated *root* node (or element). The vertices (or nodes) of the graph are key-value pairs and edges specify the *traversal function*, *i.e*, paths that should be taken by the dictionary operations in order to find the nodes that are going to determine the effect of the operation. Keys are natural numbers, values are taken from a set V and the outgoing edges of each node are locally labelled. By a light abuse of notation we say that G *find* both nodes and edges. Key values of nodes in a DAG G are related by a partial order $\prec_G$ that additionally defines a property $\mathbb{P}_G$ specifying if there is an outgoing edge from node with key k to a node with key k' (we say that G *respects* $\mathbb{P}_G$).

If G contains a node a with key k, the *k-relevant* set of G, denoted $V_k(G)$, is a plus all nodes b, such that G contains (b, a) or (a, b). If G contains no nodes with key k, $V_k(G)$ consists of all nodes a of G with the smallest $k \prec_G k'$ plus all nodes b, such that (b, a) is in G. The *k-relevant graph of* G, denoted $R_k(G)$, is the subgraph of G that consists of all paths from the root to the nodes in $V_k(G)$.

Sequential specification. Every data structure in $\mathcal{D}$ exports a sequential specification with the following operations: (i) $insert(k, v)$ checks whether a node with key k is already present and, if so, returns false, otherwise it creates a node with key k and value v, *links* it to the graph (making it reachable from the root) and returns true; (ii) $delete(k)$ checks whether a node with key k is already present and, if so, *unlinks* the node from the graph (making it unreachable from the root) and returns true, otherwise it returns false; (iii) $find(k)$ returns the pointer to the node with key k or false if no such node is found.

Traversals. For each operation $op \in \{insert(k, v), delete(k), find(k)\}_{k \in \mathbb{N}, v \in V}$, each search structure is parameterized by a (possibly randomized) *traverse function* τ_{op}. Given the *last visited* node

a and the DAG of already visited nodes G_{op}, the traverse function τ_{op} returns a new node b to be *visited*, *i.e.*, accessed to get its *key* and the list of descendants, or $\emptyset$ to indicate that the search is complete.

Find, insert and delete operations. Intuitively, the traverse function is used by the operation *op* to explore the search structure and, when the function returns $\emptyset$, the sub-DAG G_{op} explored so far contains enough information for operation *op* to complete.

If $op = \mathit{find}(k)$, G_{op} either contains a node with key k or ensures that the whole graph does not contain k. As we discuss below, in *sorted* search structures, such as sorted linked-lists or skiplists, we can stop as soon as all outgoing edges in G_{op} belong to nodes with keys $k' \geq k$. Indeed, the remaining nodes can only contain keys greater than k, so G_{op} contains enough information for *op* to complete.

An operation $op = \mathit{insert}(k, v)$, is characterized by an *insert function* $\mu_{(k,v)}$ that, given a DAG G and a new node $\langle k, v\rangle \notin G$, returns the set of edges from nodes of G to $\langle k, v\rangle$ and from $\langle k, v\rangle$ to nodes of G so that the resulting graph is a DAG containing $\langle k, v\rangle$ and respects $\mathbb{P}_G$.

An operation $op = \mathit{delete}(k)$, is characterized by a *delete function* ν_k that, given a DAG G, gives the set of edges to be removed and a set of edges to be added in G so that the resulting graph is a DAG that respects $\mathbb{P}_G$.

Sequential implementations. We make the following natural assumptions on the sequential implementation of a search structure: (i) *Traverse-update*: Every operation *op* starts with the read-only *traverse* phase followed with a write-only *update* phase. The traverse phase of an operation *op* with parameter k completes at the latest when for the visited nodes G_{op} contains the k-relevant graph. The update phase of a $\mathit{find}(k)$ operation is empty; (ii) *Proper traversals and updates*: For all DAGs G_{op} and nodes $a \in G_{op}$, the traverse function $\tau_{op}(a, G_{op})$ returns b such that $(a, b) \in G$. The update phase of an $\mathit{insert}(k)$ or $\mathit{delete}(k)$ operation modifies outgoing edges of k-relevant nodes; (iii) *Non-triviality*: There exist a key k and a state G such that (1) G contains no node with key k, (2) If G' is the state resulting after applying $\mathsf{insert}(k, v)$ to G, then there is exactly one edge (a, b) in G' such that b has key k, and (3) the shortest path in G' from the root to a is of length at least 2.

The non-triviality property says that in some cases the read-phase may detect the presence of a given key only at the last step of a traverse-phase. Moreover, it excludes the pathological DAGs in which all the nodes are always reachable in one hop from the root. Moreover, the traverse-update property and the fact that keys are natural numbers implies that every traverse phase eventually terminates. Indeed, there can be only finitely many vertices pointing to a node with a given key, thus, eventually a traverse operation explores enough nodes to be sure that no node with a given key can be found.

Examples of search data structures. In Figure 2, we describe few data structures in $\mathcal{D}$. A *sorted linked list* maintains a single path, starting at the *root* sentinel node and ending at a *tail* sentinel node, and any traversal with parameter k simply follows the path until a node with key $k' \geq k$ is located. The traverse function for all operations follows the only path possible in the graph until the two relevant nodes are located.

A *skiplist* [27] of n nodes is organized as a series of $O(\log n)$ sorted linked lists, each specifying shortcuts of certain length. The bottom-level list contains all the nodes, each of the higher-level lists contains a sublist of the lower-level list. A traversal starts with the top-level list having the longest "hops" and goes to lower lists with smaller hops as the node with smallest key $k' \geq k$ get closer.

A *binary search tree* represents data items in the form of a binary tree. Every node in the tree stores a key-value pair, and the left descendant of a non-leaf node with key k roots a subtree storing all nodes with keys less than k, while the right descendant roots a subtree storing all nodes with keys greater than k. Note that, for simplicity, we do not consider *rebalancing* operations

used by balanced trees for maintaining the desired bounds on the traverse complexity. Though crucial in practice, the rebalancing operations are not important for our comparative analysis of concurrency properties of synchronization techniques.

Non-serializable concurrency. There is a straightforward LSL implementation of any data structure in $\mathcal{D}$ in which updates (*inserts* and *deletes*) acquire a lock on the root node and are thus sequential. Moreover, they take exclusive locks on the set of nodes they are about to modify (k-relevant sets for operations with parameter k).

A *find* operation uses *hand-over-hand shared* locking [29]: at each moment of time, the operation holds shared locks on all outgoing edges for the currently visited node a. To visit a new node b (recall that b must be a descendant of a), it acquires shared locks on the new node's descendants and then releases the shared lock on a. Note that just before a *find*(k) operation returns the result, it holds shared locks on the k-relevant set.

This way updates always take place sequentially, in the order of their acquisitions of the root lock. A *find*(k) operation is linearized at any point of its execution when it holds shared locks on the k-relevant set. Concurrent operations that do not contend on the same locks can be arbitrarily ordered in a linearization.

The resulting *HOH-find* implementation is described in Algorithm 1. The fact that the operations acquire (starvation-free) locks in the order they traverse the directed acyclic graph implies:

Theorem 1. HOH-find *is a starvation-free LSL implementation of a search structure.*

As we show in Section 5, the implementation is however not (safe-strict) serializable.

5 Pessimism vs. serializable optimism

In this section, we show that, with respect to search structures, pessimistic locking and optimistic synchronization providing safe-strict serializability are *incomparable*, once we focus on LS-linearizable implementations.

5.1 Classes $\mathcal{P}$ and $\mathcal{SM}$

A *synchronization technique* is a set of concurrent implementations. We define below a specific optimistic synchronization technique and then a specific pessimistic one.

$\mathcal{SM}$: serializable optimistic. Let α denote the execution of a concurrent implementation and $ops(\alpha)$, the set of operations each of which performs at least one event in α. Let α^k denote the prefix of α up to the last event of operation π_k. Let $Cseq(\alpha)$ denote the set of subsequences of α that consist of all the events of operations that are complete in α. We say that α is *strictly serializable* if there exists a legal sequential execution α' equivalent to a sequence in $\sigma \in Cseq(\alpha)$ such that $\rightarrow_\sigma \subseteq \rightarrow_{\alpha'}$.

This paper focuses on optimistic implementations that are strictly serializable and whose operations (even aborted or incomplete) observes correct (serial) behavior. More precisely, an execution α is *safe-strict serializable* if (1) α is strictly serializable, and (2) for each operation π_k, there exists a legal sequential execution $\alpha' = \pi_0 \cdots \pi_i \cdot \pi_k$ and $\sigma \in Cseq(\alpha^k)$ such that $\{\pi_0, \cdots, \pi_i\} \subseteq ops(\sigma)$ and $\forall \pi_m \in ops(\alpha') : \alpha'|m = \alpha^k|m$.

Safe-strict serializability captures nicely both local serializability and linearizability. If we transform a sequential implementation IS of a type τ into a *safe-strict serializable* concurrent one, we obtain an LSL implementation of (IS, τ). Thus, the following lemma is immediate.

Lemma 2. *Let I be a safe-strict serializable implementation of (IS, τ). Then, I is LS-linearizable with respect to (IS, τ).*

Figure 3: (a) a history of integer set (implemented as linked list or binary search tree) exporting schedule σ, with initial state $\{1, 2, 3\}$ (r denotes the root node); (b) a history exporting a problematic schedule σ', with initial state $\{3\}$, which should be accepted by any $I \in \mathcal{P}$ if it accepts σ

Indeed, we make sure that completed operations witness the same execution of IS, and every operation that returned $\perp$ is consistent with some execution of IS based on previously completed operations. Formally, $\mathcal{SM}$ denotes the set of optimistic, safe-strict serializable LSL implementations.

$\mathcal{P}$: deadlock-free pessimistic. Assuming that no process stops taking steps of its algorithm in the middle of a high-level operation, at least one of the concurrent operations return a matching response [22]. Note that $\mathcal{P}$ includes implementations that are not necessarily safe-strict serializable.

5.2 Suboptimality of pessimistic implementations

We show now that for any search structure, there exists a schedule that is rejected by *any* pessimistic implementation, but accepted by certain optimistic strictly serializable ones. To prove this claim, we derive a safe-strict serializable schedule that cannot be accepted by any implementation in $\mathcal{P}$ using the *non-triviality* property of search structures. It turns out that we can schedule the traverse phases of two *insert*(k) operations in parallel until they are about to check if a node with key k is in the set or not. If it is, both operations may safely return false (schedule σ). However, if the node is not in the set, in a pessimistic implementation, both operations would have to modify outgoing edges of the same node a and, if we want to provide local serializability, both return true, violating linearizability (schedule σ').

In contrast, an optimistic implementation may simply abort one of the two operations in case of such a conflict, by accepting the (correct) schedule σ and rejecting the (incorrect) schedule σ'.

Proof intuition. We first provide an intuition of our results in the context of the *integer set* implemented as a *sorted linked list* or *binary search tree*. The set type is a special case of the dictionary which stores a set of integer values, initially empty, and exports operations *insert*(v), *remove*(v), *find*(v); $v \in \mathbb{Z}$. The update operations, *insert*(v) and *remove*(v), return a boolean response, true if and only if v is absent (for *insert*(v)) or present (for *remove*(v)) in the set. After *insert*(v) is complete, v is present in the set, and after *remove*(v) is complete, v is absent in the set. The *find*(v) operation returns a boolean, true if and only if v is present in the set.

An example of schedules σ and σ' of the set is given in Figure 3. We show that the schedule σ depicted in Figure 3(a) is not accepted by any implementation in $\mathcal{P}$. Suppose the contrary and let σ be exported by an execution α. Here α starts with three sequential insert operations with parameters 1, 2, and 3. The resulting "state" of the set is $\{1, 2, 3\}$, where value $i \in \{1, 2, 3\}$ is stored in node X_i. Suppose, by contradiction, that some $I \in \mathcal{P}$ accepts σ. We show that I then accepts the schedule σ' depicted in Figure 3(b), which starts with a sequential execution of insert(3) storing value 3 in node X_1. We can further extend σ' with a complete find(1) (by deadlock-freedom of $\mathcal{P}$) that will return false (the node inserted to the list by insert(1) is lost)—a contradiction since I is linearizable with respect to *set*.

Due to space consraints, the formal proof is moved to Appendix C.

Theorem 3. *Any abstraction in $\mathcal{D}$ has a strictly serializable schedule that is not accepted by* any *implementation in $\mathcal{P}$, but accepted by an implementation in $\mathcal{SM}$.*

5.3 Suboptimality of serializable optimism

We show below that for any search structure, there exists a schedule that is rejected by *any* serializable implementation but accepted by a certain pessimistic one (*HOH-find*, to be concrete).

Proof intuition. We first illustrate the proof in the context of the integer set. Consider a schedule σ_0 of a concurrent set implementation depicted in Figure 1. We assume here that the set initial state is $\{1, 3, 4\}$. Operation $\mathsf{find}(5)$ is concurrent, first with operation $\mathsf{insert}(2)$ and then with operation $\mathsf{insert}(5)$. The history is not serializable: $\mathsf{insert}(5)$ sees the effect of $\mathsf{insert}(2)$ because $R(X_1)$ by $\mathsf{insert}(5)$ returns the value of X_1 that is updated by $\mathsf{insert}(2)$ and thus should be serialized after it. But $\mathsf{find}(5)$ misses node with value 2 in the set, but must read the value of X_4 that is updated by $\mathsf{insert}(5)$ to perform the read of X_5, *i.e.*, the node created by $\mathsf{insert}(5)$. Thus, σ_0 is not (safe-strict) serializable. This history though is LSL since each of the three local histories is consistent with some sequential history of the integer set. However, there exists an execution of our HOH-find implementation that exports σ_0 since there is no read-write conflict on any two consecutive nodes accessed.

To extend the above idea to any search structure, we use the *non-triviality* property of data structures in $\mathcal{D}$. There exist a state G' in which there is exactly one edge (a, b) in G' such that b has key k. We schedule a $op_f = find(k)$ operation concurrently with two consecutive delete operations: the first one, op_{d1}, deletes one of the nodes explored by op_f before it reaches a (such a node exists by the *non-triviality* property), and the second one, op_{d2} deletes the node with key k in G'. We make sure that op_f is not affected by op_{d1} (observes an update to some node c in the graph) but is affected by op_{d2} (does not observe b in the graph). The resulting schedule is not strictly serialializable (though linearizable). But our HOH-find implementation in $\mathcal{P}$ will accept it.

Theorem 4. *For any abstraction in $D \in \mathcal{D}$, there exists an implementation in $\mathcal{P}$ that accepts a non-strictly serializable schedule.*

Since any strictly serializable optimistic implementation only produces strictly serializable executions, from Theorem 4 we deduce that there is a schedule accepted by a pessimistic algorithm that no strictly serializable optimistic one can accept. Therefore, Theorems 3 and 4 imply that, when applied to search structures and in terms of concurrency, the strictly serializable optimistic approach is incomparable with pessimistic locking. As a corollary, none of these two techniques can be concurrency-optimal.

6 Related work

Sets of accepted schedules are commonly used as a metric of concurrency provided by a shared memory implementation. For static database transactions, Kung and Papadimitriou [22] acknowledge that this metric may have "practical significance, if the schedulers in question have relatively small scheduling times as compared with waiting and execution times". Herlihy [18] implicitly considers a synchronization technique as highly concurrent, namely optimal, if no other technique accepts more schedules. By contrast, we focus here on a dynamic model where the scheduler cannot use the prior knowledge of all the shared addresses to be accessed.

Gramoli *et al.* [11, 12] defined a concurrency metric, the *input acceptance*, as the ability of a TM to commit classes of input patterns of memory accesses without violating conflict-serializability. Guerraoui *et al.* [14] defined the notion of *permissiveness* as the ability for a TM to abort a transaction only if committing it would violate consistency. In contrast with these definitions, our framework for analyzing concurrency is independent of the synchronization technique. David *et al.* [6] consider that the closer the throughput of a concurrent algorithm is to that of its (inconsistent) sequential variant, the more concurrent the algorithm. In contrast, the formalism proposed in our paper allows for relating concurrency properties of various correct concurrent algorithms.

Our definition of search data structures is based on the paper by Chaudhri and Hadzilacos [4] who studied them in the context of dynamic databases. Safe-strict serializable implementations ($\mathcal{SM}$) require that every transaction (even aborted and incomplete) observes "correct" serial behavior. It is weaker than popular TM correctness conditions like opacity [15] and its relaxations like TMS1 [7] and VWC [24]. Unlike TMS1, we do not require the *local* serial executions to always respect the real-time order among transactions.

7 Concluding remarks

In this paper, we presented a formalism for reasoning about the relative power of optimistic and pessimistic synchronization techniques in exploiting concurrency in search structures. We expect our formalism to have practical impact as the search structures are among the most commonly used concurrent data structures, including trees, linked lists, skip lists that implement various abstractions ranging from key-value stores to sets and multi-sets.

Our results on the relative concurrency of $\mathcal{P}$ and $\mathcal{SM}$ imply that none of these synchronization techniques might enable an optimally-concurrent algorithm. Of course, we do not claim that our concurrency metric necessarily captures efficiency, as it does not account for other factors, like cache sizes, cache coherence protocols, or computational costs of validating a schedule, which may also affect performance on multi-core architectures. In [13] we already described a *concurrency-optimal* implementation of the linked-list set abstraction that combines the advantages of $\mathcal{P}$, namely the semantics awareness, with the advantages of $\mathcal{SM}$, namely the ability to restart operations in case of conflicts. We recently observed empirically that this optimality can result in higher performance than state-of-the-art algorithms [17, 16, 25]. Therefore, our findings motivate the search for concurrency-optimal algorithms. This study not only improves our understanding of designing concurrent data structures, but might lead to more efficient implementations.

Acknowledgments

This research was supported under Australian Research Council's Discovery Projects funding scheme (project number 160104801) entitled "Data Structures for Multi-Core". Vincent Gramoli is the recipient of the Australian Research Council Discovery International Award. Petr Kuznetsov was supported the Agence Nationale de la Recherche, under grant agreement N ANR-14-CE35-0010-01, project DISCMAT.

References

[1] Y. Afek, A. Matveev, and N. Shavit. Pessimistic software lock-elision. In *DISC*, pages 297–311, Berlin, Heidelberg, 2012. Springer-Verlag.

[2] M. K. Aguilera, S. Frølund, V. Hadzilacos, S. L. Horn, and S. Toueg. Abortable and query-abortable objects and their efficient implementation. In *PODC*, pages 23–32, 2007.

[3] H. Attiya and J. Welch. *Distributed Computing. Fundamentals, Simulations, and Advanced Topics.* John Wiley & Sons, 2004.

[4] V. K. Chaudhri and V. Hadzilacos. Safe locking policies for dynamic databases. *J. Comput. Syst. Sci.*, 57(3):260–271, 1998.

[5] L. Dalessandro, M. F. Spear, and M. L. Scott. NOrec: streamlining STM by abolishing ownership records. In *PPOPP*, pages 67–78, 2010.

[6] T. David, R. Guerraoui, and V. Trigonakis. Asynchronized concurrency: The secret to scaling concurrent search data structures. In *ASPLOS*, pages 631–644, 2015.

[7] S. Doherty, L. Groves, V. Luchangco, and M. Moir. Towards formally specifying and verifying transactional memory. *Electron. Notes Theor. Comput. Sci.*, 259:245–261, Dec. 2009.

[8] P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *PPoPP*, pages 237–246, 2008.

[9] V. Gramoli. More than you ever wanted to know about synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorithms. In *PPoPP*, pages 1–10, 2015.

[10] V. Gramoli and R. Guerraoui. Democratizing transactional programming. *Commun. ACM*, 57(1):86–93, Jan 2014.

[11] V. Gramoli, D. Harmanci, and P. Felber. Toward a theory of input acceptance for transactional memories. In *OPODIS*, volume 5401 of *LNCS*, pages 527–533, 2008.

[12] V. Gramoli, D. Harmanci, and P. Felber. On the input acceptance of transactional memory. *Parallel Processing Letters*, 20(1):31–50, 2010.

[13] V. Gramoli, P. Kuznetsov, S. Ravi, and D. Shang. Brief announcement: A concurrency-optimal list-based set. In *Distributed Computing - 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9*, 2015. Technical report available at `http://arxiv.org/abs/1502.01633`.

[14] R. Guerraoui, T. A. Henzinger, and V. Singh. Permissiveness in transactional memories. In *DISC*, pages 305–319, 2008.

[15] R. Guerraoui and M. Kapalka. *Principles of Transactional Memory,Synthesis Lectures on Distributed Computing Theory.* Morgan and Claypool, 2010.

[16] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *DISC*, pages 300–314, 2001.

[17] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. N. Scherer, and N. Shavit. A lazy concurrent list-based set algorithm. In *OPODIS*, pages 3–16, 2006.

[18] M. Herlihy. Apologizing versus asking permission: optimistic concurrency control for abstract data types. *ACM Trans. Database Syst.*, 15(1):96–124, 1990.

[19] M. Herlihy. Wait-free synchronization. *ACM Trans. Prog. Lang. Syst.*, 13(1):123–149, 1991.

[20] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA*, pages 289–300, 1993.

[21] M. Herlihy and N. Shavit. *The art of multiprocessor programming.* Morgan Kaufmann, 2008.

[22] M. Herlihy and N. Shavit. On the nature of progress. In *OPODIS*, pages 313–328, 2011.

[23] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[24] D. Imbs, J. R. G. de Mendívil, and M. Raynal. Brief announcement: virtual world consistency: a new condition for stm systems. In *PODC*, pages 280–281, 2009.

[25] M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *SPAA*, pages 73–82, 2002.

[26] C. H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26:631–653, 1979.

[27] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990.

[28] N. Shavit and D. Touitou. Software transactional memory. In *PODC*, pages 204–213, 1995.

[29] W. E. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Trans. Comput.*, 37(12):1488–1505, 1988.

[30] G. Weikum. A theoretical foundation of multi-level concurrency control. In *PODS*, pages 31–43, 1986.

[31] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery.* Morgan Kaufmann, 2002.

[32] M. Yannakakis. Serializability by locking. *J. ACM*, 31(2):227–244, 1984.

```
1:  Shared variables:
2:    G, initially root                                  ▷ Shared DAG

3:  find(k):
4:    G ← ∅; a ← {root}
5:    a.lock-shared()
6:    while a ≠ ∅ do
7:      ∀(a, b) ∈ G: b.lock-shared()
8:      ∀(a, b) ∈ G: G ← G ∪ (a, b)        ▷ Explore new edges
9:      last ← a
10:     a ← τ_find(k)(a, G)
11:     ∀(last, b) ∈ G, b ≠ a: b.unlock-shared()
12:     last.unlock-shared()
13:   if G contains a node with key k then
14:     return true
15:   else
16:     return false

17: insert(k, v):
18:   root.lock()
19:   G ← ∅; a ← {root}
20:   while a ≠ ∅ do
21:     ∀(a, b) ∈ G: G ← G ∪ (a, b)        ▷ Explore new edges
22:     last ← a
23:     a ← τ_insert(k,v)(a, G)
24:     ∀(last, b) ∈ G,
25:   if G contains no node with key k then
26:     a ← create-node(k, v)
27:     ∀b such that ∃(b, c) ∈ μ_k(G, a), b.lock()
28:     G ← G ∪ μ_(k,v)(G, a)                ▷ Link a to G
29:     ∀b such that ∃(b, c) ∈ μ_k(G, a), b.unlock()
30:     root.unlock()
31:     return true
32:   else
33:     root.unlock()
34:     return false

35: delete(k):
36:   root.lock()
37:   G ← ∅; a ← {root}
38:   while a ≠ ∅ do
39:     ∀(a, b) ∈ G: G ← G ∪ (a, b)        ▷ Explore new edges
40:     last ← a
41:     a ← τ_delete(k)(a, G)
42:     ∀(last, b) ∈ G,
43:   if G contains node a with key k then
44:     remove a and all edges to/from a from G
45:     ∀b such that ∃(b, c) ∈ ν_k(G, a), b.lock()
46:     G ← G ∪ ν_k(G, a)                    ▷ Shortcut edges
47:     ∀b such that ∃(b, c) ∈ ν_k(G, a), b.unlock()
48:     root.unlock()
49:     return true
50:   else
51:     root.unlock()
52:     return false
```

Algorithm 1: Abstract *HOH-find* implementation of a search structure defined by (τ_{op}, $\mu_{insert(k,v)}$, $\nu_{delete(k)}$), $op \in \{insert(k, v), delete(k), find(k)\}$, $k \in \mathbb{N}$, $v \in V$.

A LS-linearizability is compositional

We define the composition of two distinct object types τ_1 and τ_2 as a type $\tau_1 \times \tau_2 = (\Phi, \Gamma, Q, q_0, \delta)$ as follows: $\Phi = \Phi_1 \cup \Phi_2$, $\Gamma = \Gamma_1 \cup \Gamma_2$,[1] $Q = Q_1 \times Q_2$, $q_0 = (q_{01}, q_{02})$, and $\delta \subseteq Q \times \Phi \times Q \times \Gamma$ is such that $((q_1, q_2), \pi, (q_1' q_2'), r) \in \delta$ if and only if for $i \in \{1, 2\}$, if $\pi \in \Phi_i$ then $(q_i, \pi, q_i', r) \in \delta_i \wedge q_{3-i} = q_{3-i}'$.

Every sequential implementation *IS* of an object $O_1 \times O_2$ of a composed type $\tau_1 \times \tau_2$ naturally induces two sequential implementations I_{S1} and I_{S2} of objects O_1 and O_2, respectively. Now a correctness criterion Ψ is *compositional* if for every history H on an object composition $O_1 \times O_2$, if Ψ holds for $H|O_i$ with respect to I_{Si}, for $i \in \{1, 2\}$, then Ψ holds for H with respect to $IS = I_{S1} \times I_{S2}$. Here, $H|O_i$ denotes the subsequence of H consisting of events on O_i.

Theorem 5. *LS-linearizability is compositional.*

Proof. Let H, a history on $O_1 \times O_2$, be LS-linearizable with respect to *IS*. Let each $H|O_i$, $i \in \{1, 2\}$, be LS-linearizable with respect to I_{Si}. Without loss of generality, we assume that H is complete (if H is incomplete, we consider any completion of it containing LS-linearizable completions of $H|O_1$ and $H|O_2$).

Let $\tilde{H}$ be a completion of the high-level history corresponding to H such that $\tilde{H}|O_1$ and $\tilde{H}|O_2$ are linearizable with respect to τ_1 and τ_2, respectively. Since linearizability is compositional [23, 21], $\tilde{H}$ is linearizable with respect to $\tau_1 \times \tau_2$.

[1] Here we treat each τ_i as a distinct type by adding index i to all elements of Φ_i, Γ_i, and Q_i.

Now let, for each operation π, S^1_π and S^2_π be any two sequential histories of I_{S1} and I_{S2} such that $H|\pi|O_j = S^j_\pi|\pi$, for $j \in \{1,2\}$ (since $H|O_1$ and $H|O_2$ are LS-linearizable such histories exist). We construct a sequential history S_π by interleaving events of S^1_π and S^2_π so that $S_\pi|O_j = S^j_\pi$, $j \in \{1,2\}$. Since each S^j_π acts on a distinct component O_j of $O_1 \times O_2$, every such S_π is a sequential history of IS. We pick one S_π that respects the local history $H|\pi$, which is possible, since $H|\pi$ is consistent with both $S_1|\pi$ and $S_2|\pi$.

Thus, for each π, we obtain a history of IS that agrees with $H|\pi$. Moreover, the high-level history of H is linearizable. Thus, H is LS-linearizable with respect to IS. □

B Non-serializable concurrency

Theorem 1. *The* HOH-find *algorithm is a starvation-free LSL implementation of a search structure.*

Proof. Take any execution E of the algorithm. The subsequence of E consisting of the events of update operations is serializable (and, thus, locally serializable). Since a *find*(k) operation protects its visited node and all its outgoing edges with a shared lock and a concurrent update with a key protect their k-relevant sets with an exclusive lock, *find*(k) observes the effects of updates as though they took place in a sequential execution—thus local serializability.

Let H be the history of E. To construct a linearization of H, we start with a sequential history S that orders all update operations in H in the order in which they acquire locks on the root. By construction, S is legal. A *find*(k) operation that returns true can only reach a node if a node with key k was reachable from the root at some point during its interval. Similarly, if *find*(k) operation returns false, then it would only fail to reach a node if it was made unreachable from the root at some point during its interval. Thus, every successful (resp., unsuccessful) *find*(k) operation *op* can be inserted in S after the latest update operation that does not succeed in the real-time order in E and after which a node k is reachable (resp., unreachable). By construction, the resulting sequential history is legal. □

C Pessimism vs. serializable optimism

Theorem 3. *Any abstraction in $\mathcal{D}$ has a strictly serializable schedule that is not accepted by* any *implementation in $\mathcal{P}$, but accepted by an implementation in $\mathcal{SM}$.*

Proof. Consider the key k and the states G and G' satisfying the conditions of the *non-triviality* property.

Consider the schedule that begins with a serial sequence of operations bringing the search to state G (executed by a single process). Then schedule the traverse phases of two identical *insert*(k, v) operations executed by *new* (not yet participating) processes p_1 and p_2 concurrently so that they perform identical steps (assuming that, if they take randomized steps, their coin tosses return the same values). Such an execution E exists, since the traverse phases are read-only. But if we allow both insert operations to proceed (by deadlock-freedom), we obtain an execution that is not LS-linearizable: both operations update the data structure which can only happen in a successful insert. But, by the sequential specification of D, since node with key k belongs to G, at least one of the two inserts must fail. Therefore, a pessimistic implementation, since it is not allowed to abort an operation, cannot accept the corresponding schedule σ.

Now consider the serial sequence of operations bringing D to state G' (executed by a single process) and extend it with traverse phases of two concurrent *insert*(k, v) operations executed by new processes p_1 and p_2. The two traverse phases produce an execution E' which is indistinguishable to p_1 and p_2 from E up to their last read operations. Thus, if a pessimistic implementation accepts the corresponding schedule σ, it must also accept σ', violating LS-linearizability.

Note, however, that an extension of E' in which both inserts complete by returning false is LS-linearizable. Moreover, any *progressive* (e.g., using progressive opaque transactional memory) optimistic strictly serializable implementation using will accept σ'. □

Theorem 4. *For any abstraction in $D \in \mathcal{D}$, there exists an implementation in $\mathcal{P}$ that accepts a non-strictly serializable schedule.*

Proof. Consider the *HOH-find* implementation described in Algorithm 1. Take the key k and the states G and G' satisfying the conditions of the *non-triviality* property.

Now we consider the following execution. Let $op_f = \mathit{find}(k)$ be applied to an execution resulting in G' (that contains a node with key k) and run op_f until it reads the only node $a = (k', v')$ in G that points to a node $b = (k, v)$ in state G'. Note that since $R_k(G) = R_k(G')$, the operation cannot distinguish the execution from than one starting with G.

The *non-triviality* property requires that the shortest path from the root to k to a in $R_k(G)$ is of length at least two. Thus, the set of nodes explored by op_f passed through at least one node $c = (k'', v'')$ in addition to a. Now we schedule two complete delete operations executed by another process: first $del_c = \mathit{delete}(k'')$ which removes c, followed by $del_b = \mathit{delete}(k)$ which removes b. Now we wake up op_f and let it read a, find out that no node with key k is reachable, and return false

Suppose, by contradiction, that the resulting execution is strictly serializable. Since op_f has witnessed the presence of some node c on the path from the root to a in the DAG, op_f must precede del_c in any serializaton. Now del_b affected the response of op_f, it must precede op_f in any serialization. Finally, del_c precedes del_b in the real-time order and, thus must precede del_b in any serialization. The resulting cycle implies a contradiction. □