Optimal Concurrency for List-Based Sets

Vitaly Aksenov[1], Vincent Gramoli[2], Petr Kuznetsov[3], Di Shang[4], and Srivatsan Ravi[5]

[1] ITMO University, aksenov.vitaly@gmail.com
[2] University of Sydney, vincent.gramoli@sydney.edu.au
[3] LTCI, Télécom Paris, Institut Polytechnique de Paris, petr.kuznetsov@telecom-paris.fr
[4] IBM, sapphirus22@gmail.com
[5] University of Southern California, srivatsr@usc.edu

Abstract. Designing an efficient concurrent data structure is a challenge that is not easy to meet. Intuitively, efficiency of an implementation is defined, in the first place, by its ability to process applied operations *in parallel*, without using unnecessary synchronization. As we show in this paper, even for a data structure as simple as a linked list used to implement the *set* type, the most efficient algorithms known so far are *not concurrency-optimal*: they may reject correct concurrent schedules. We propose a new algorithm for the list-based set based on a *value-aware try-lock* that we show to achieve *optimal* concurrency: it only rejects concurrent schedules that violate correctness of the implemented set type. We show that reaching this kind of optimality may be beneficial in practice. Our *concurrency-optimal* list-based set outperforms two state-of-the-art algorithms: the Lazy Linked List and the Harris-Michael List.

1 Introduction

Multicore applications require highly concurrent data structures. Yet, the very notion of concurrency is vaguely defined, to say the least. What do we mean by a "highly concurrent" data structure? Generally speaking, one could compare the concurrency of algorithms by running a game where the adversary decides on the schedules of shared memory accesses from different processes. At the end of the game, the more schedules the algorithm would accept without hampering high-level correctness, the more concurrent it would be. The algorithm that accepts all correct schedules would then be considered *concurrency-optimal* [1].

To illustrate the difficulty of optimizing concurrency, let us consider one of the most "concurrency-friendly" data structures [2]: the *sorted linked list* used to implement the *integer set* type. Since any modification on a linked list affects only a small number of contiguous list nodes, most of update operations on the list could, in principle, run concurrently without conflicts. For example, one of the most efficient concurrent list-based set to date, the Lazy Linked List [3], achieves high concurrency by holding locks on only two consecutive nodes when updating, thus accepting *concurrent* modifications of non-contiguous nodes. The Lazy Linked List is known to outperform the Java variant [4] of the CAS-based

Harris-Michael algorithm [5, 6] under low contention because all its traversals, be they for read-only look-ups or for locating the nodes to be updated, are *wait-free*, *i.e.*, they ignore locks and logical deletion marks. As we show below, the Lazy Linked List implementation is however not *concurrency-optimal*, raising two questions: Is there a more concurrent list-based set algorithm? And if so, does higher concurrency induce an overhead that precludes higher performance?

The concurrency limitation of the Lazy Linked List is caused by the locking strategy of its update operations: both insert(v) and remove(v) traverse the structure until they find a node whose value is larger or equal to v, at which point they acquire locks on two consecutive nodes. Only then is the existence of the value v checked: if v is found (resp., not found), then the insertion (resp., removal) releases the locks and returns without modifying the structure. By modifying metadata during lock acquisition without necessarily modifying the structure itself, the Lazy Linked List over conservatively rejects certain correct schedules. To illustrate that the concurrency limitation of the Lazy Linked List may lead to poor scalability, consider Figure 1 that depicts the performance of a 25-node Lazy Linked List (red curve) under a workload of 20% updates (insert/removals) and 80% contains on a 72-core machine. The list is comparatively small, hence all updates (even the failed insertions and removals) are likely to contend. We can see that when we increase the number of threads beyond 40, the performance drops significantly.

This observation suggests a desirable property that concurrent operations should conflict on metadata only when they conflict on data. To achieve this, we need to exploit the semantics of the high-level data type. [6]

Our main contribution is the Value-Based List (VBL), the most concurrent (in fact, *optimally* concurrent, as we formally prove) and probably the most efficient list-based set algorithm to date. It exploits the logical deletion technique of Harris-Michael that divides the removal of a node into a logical step (marking the node for deletion) and a physical step (unlinking the node from the list), and the wait-free traversal of the Lazy Linked List. In addition, our approach relies on a novel *value-aware* synchronization technique: first the lock, implemented using *compare-and-swap*, is taken, then the procedure checks whether the *value* in the next node has changed, if the validation is successful then the operation continues, otherwise, the operation restarts. Compared to the Lazy Linked List, this approach allows for the improvement of performance and even provides scalability in the highly contended cases (Figure 1). We show that the resulting algorithm rejects a concurrent schedule only if otherwise the high-level correctness of the implemented set type (linearizability [8]) is violated. Our al-

[6] Note that this property refines the original notion of disjoint access parallelism (DAP) [7], trivially ensured by most linked-list implementations simply because all their operations "access" the *head* node and, thus, are allowed to conflict on the metadata.

gorithm is thus *concurrency-optimal* [1]: no correct list-based set algorithm can accept more schedules. [7]

The evaluation of `VBL` shows that achieving optimal concurrency in list-based set implementations does not necessarily result in a costly overhead, complementing the recent analysis of concurrency-optimality for tree-based dictionaries [9]. Extensive experiments on two x86-64 architectures machines, 72-way Intel machine and 64-way AMD machine, confirmed that `VBL` outperforms the state-of-the-art algorithms [3, 4]. In particular, `VBL` outperforms the Lazy Linked List performance by 1.6× for 72 threads on the 20%-update workload of Figure 1, which can be explained by the fact that our algorithm validates list data *before* locking, and not after. In addition, as our algorithm differs from Harris-Michael by avoiding metadata accesses during traversals, it outperforms it by up to 1.6× on read-only workloads. We report the performance of the Java variant of Harris-Michael list-based set with wait-free `contains` as presented in Shavit and Herlihy's book [4] and the Java optimised implementation with RTTI [3], and, in the technical report [10],on the performance of our own C++ translations of the Lazy algorithm (without memory management).

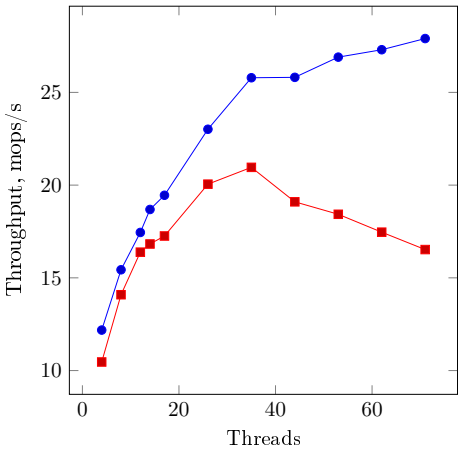

Fig. 1: The throughput of Lazy Linked List (red square curve) and `VBL` (blue circle curve). We consider the load with only 20% updates. Lazy Linked List behaves worse, as its operations potentially contend on meta-data even when they do not modify the data structure.

Roadmap. The rest of this paper is structured as follows. We present our methodology on modelling concurrency and prove the suboptimal concurrency of the Lazy and Harris-Michael linked lists in Section 2. In Section 3, we present our `VBL` list implementation. Section 4 presents the methodology for performance evaluation of concurrent list implementations and Section 5 presents a discussion of concurrency w.r.t list-based sets. The full proofs of linearizability and deadlock-freedom are deferred to the technical report [10]. Synchrobench benchmark suite [11] contains the code for all the lists considered in this paper.

2 Concurrency analysis of list-based sets

2.1 Preliminaries

We consider a standard asynchronous shared-memory system, in which $n > 1$ processes (or *threads* of computation) $p_1, \ldots, p_n$ communicate by applying operations on shared *objects*.

[7] Here we adapt to list-based sets the notion of concurrency-optimality, introduced in [1] for generic search data structures.

Sequential list-based set. An object of the *set* type stores a set of integer values, initially empty, and exports operations insert(v), remove(v), contains(v) where $v \in \mathbb{Z}$. 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 list. After insert(v) is complete, v is present in the list, and after remove(v) is complete, v is absent from the list. The contains(v) returns a boolean *true* if and only if v is present in the list. The concurrent *set* implementations considered in this paper are based on a a specific *sequential* one. The implementation, denoted *LL*, stores set elements in a *sorted linked list*, where each list node has a *next* field pointing to the successor node. Initially, the *next* field of the *head* node points to *tail*; *head* (resp. *tail*) is initialized with values $-\infty$ (resp., $+\infty$) that is smaller (resp., greater) than any other value in the list. We follow natural sequential implementations of operations insert, remove, and contains presented in detail in [10].

Executions. An *event* of a process p_i (we also say a *step* of p_i) is an invocation or response of an operation performed by p_i on a high-level object (in this paper, a set) implementation, or a *primitive* applied by p_i to a base object b along with its response. A *configuration* specifies the value of each base object and the state of each process. The *initial configuration* is the configuration in which all base objects have their initial values and all processes are in their initial states. An *execution fragment* is a (finite or infinite) sequence of events. An *execution* of an implementation I is an execution fragment where, starting from the initial configuration, each event is issued according to I and each response of a primitive matches the state of b resulting from all preceding events.

A *high-level history* $\tilde{H}$ of an execution α is the subsequence of α consisting of all invocations and responses of (high-level) operations.

Let $\alpha|p_i$ (resp. $H|p_i$) denote the subsequence of an execution α (resp. a histiry H) restricted to the events of process p_i. Executions α and α' (resp. histories H and H') are *equivalent* if for every process p_i, $\alpha|p_i = \alpha'|p_i$ (resp. $H|p_i = H'|p_i$). An operation π *precedes* another operation π' in an execution α (resp. history H), denoted $\pi \rightarrow_\alpha \pi'$ (resp. , $\pi \rightarrow_H \pi'$) if the response of π occurs before the invocation of π' in α (resp. H). Two operations are *concurrent* if neither precedes the other.

An execution (resp. history) is *sequential* if it has no concurrent operations. An operation is *complete* in α if the invocation event is followed by a *matching* response; otherwise, it is *incomplete* in α. Execution α is *complete* if every operation is complete in α.

High-level histories and linearizability. 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 $\tilde{H}$ 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 [8].

2.2 Concurrency as admissible schedules of sequential code

Algorithm 1 Sequential implementation *LL* (*sorted linked list*) of *set* type: Shared memory reads and writes are explicitly depicted

```
1:  Shared variables:
2:    class Node: head, tail
3:      head.val = −∞
4:      tail.val = +∞
5:      head.next = tail

6:  insert(v):
7:    prev ← head
8:    curr ← read(prev.next)
9:    while (tval ← read(curr.val)) < v do
10:     prev ← curr
11:     curr ← read(curr.next)
12:   if tval ≠ v then
13:     X ← new-node(v, prev.next)
14:     write(prev.next, X)
15:   return (tval ≠ v)

16: remove(v):
17:   prev ← head
18:   curr ← read(prev.next)
19:   while (tval ← read(curr.val)) < v do
20:     prev ← curr
21:     curr ← read(curr.next)
22:   if tval = v then
23:     tnext ← read(curr.next)
24:     write(prev.next, tnext)
25:   return (tval = v)

26: contains(v):
27:   curr ← head
28:   curr ← read(prev.next)
29:   while (tval ← read(curr.val)) < v do
30:     curr ← read(curr.next)
31:   return (tval = v)
```

Schedules. Informally, a *schedule* of a list-based set algorithm specifies the order in which concurrent high-level operations access the list nodes. Consider the *sequential* implementation, *LL*, of operations insert, remove and contains. Suppose that we treat this implementation as a *concurrent* one, i.e., simply run it in a concurrent environment, without introducing any synchronization mechanisms, and let § denote the set of the resulting executions, we call them *schedules*. Of course, some schedules in § will not be linearizable. For example, concurrent inserts operating on the same list nodes may result in "lost updates": an inserted element disappears from the list due to a concurrent insert operation. But, intuitively, as no synchronization primitives are used, this (incorrect) implementation is as concurrent as it can get.

We measure the concurrency properties of a linearizable list-based set via its ability to accept all *correct* schedules in §. Intuitively, a schedule is correct if it respects the sequential implementation *LL locally*, i.e., no operation in it can distinguish the schedule from a sequential one. Furthermore, the schedule must be linearizable, even when we consider its extension in which all update operations are completed and followed with a contains(v) for any $v \in \mathbb{Z}$. Let us denote this extension of schedule σ by $\bar{\sigma}(v)$.

Given a schedule σ and an operation π, let $\sigma|\pi$ denote the subsequence of σ consisting of all steps of π.

Definition 1 (Correct schedules). *We say that a schedule σ of a concurrent list-based set implementation is* locally serializable *(with respect to the sequential implementation of list-based set LL) if for each of its operations π, there exists a sequential schedule S of LL such that $\sigma|\pi = S|\pi$. We say that a schedule is* correct *if (1) σ is locally serializable (with respect to LL), (2) for all $v \in \mathbb{Z}$, $\bar{\sigma}(v)$ is linearizable (with respect to the* set *type).*

Note that the last condition is necessary for filtering out schedules with "lost updates". Consider, for example a schedule in which insert(1) and insert(2) are applied to the initial empty set. Imagine that they first both read *head*, then both read *tail*, then both perform writes on the *head.next* and complete. The resulting schedule is, technically, linearizable and locally serializable but, obviously, not acceptable. However, in the schedule, one of the operations, say insert(1), overwrites the effect of the other one. Thus, if we extend the schedule with a complete execution of contains(2), the only possible response it may give is *false* which obviously does not produce a linearizable high-level history.

Note also that, as linearizability is a safety property [12], if $\bar{\sigma}(v)$ is linearizable, σ is linearizable too. (In the following we omit mentioning set and *LL* when we talk about local serializability and linearizability.)

Concurrency-optimality. A concurrent list-based set generally follows *LL*: every high-level operation, insert, remove, or contains, *reads* the list nodes, one after the other, until the desired fragment of the list is located. The update operation (insert or remove) then *writes*, to the *next* field of one of the nodes, the address of a new node (if it is insert) or the address of the node that follows the removed node in the list (if it is remove). Note that the (sequential) write can be implemented using a *CAS* primitive [5].

Let α denote an execution of a concurrent implementation of a list-based set. We define the *schedule σ exported by α* as the subsequence of α consisting of reads, writes and node creation events (corresponding to the sequential implementation *LL*) of operations insert, remove and contains that "take effect". Intuitively, taking effect means that they affect the outcome of some operation. The exact way an execution α is mapped to the corresponding schedule σ is implementation specific.

An implementation I *accepts* a schedule σ if there exists an execution of I that exports σ.

Definition 2 (Concurrency-optimality). *An implementation is* concurrency-optimal *if it accepts every correct schedule.*

2.3 Concurrency analysis of the Lazy and Harris-Michael Linked Lists

In this section, we show that even state-of-the-art implementations of the list-based set, namely, the Lazy Linked List and the Harris-Michael Linked list are suboptimal w.r.t exploiting concurrency. We show that each of these two algorithms rejects some correct schedules of the list-based set.

Lazy Linked List. In this deadlock-free algorithm [3], the list is traversed in the wait-free manner and the locks are taken by update operations only when the desired interval of the list is located. A remove operation first marks a node for *logical* deletion and then *physically* unlinks it from the list. To take care of conflicting updates, the locked nodes are *validated*, which involves checking if they are not logically deleted. If validation fails, the traversal is repeated. The schedule of an execution of this algorithm is naturally derived by considering only the *last* traversal of an operation.

Figure 2 illustrates how the post-locking validation strategy employed by the Lazy Linked List makes it concurrency sub-optimal. As explained in the introduction, the insert operation of the Lazy Linked List acquires the lock on the nodes it writes to, prior to the check of the node's state.

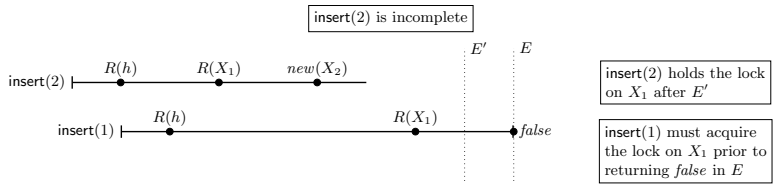

Fig. 2: A schedule rejected by the Lazy Linked List; initial list state is $\{X_1\}$ that stores value 1; $R(X_1)$ refers to reads of both *val* and *next* fields; $new(X_2)$ creates a new node storing value 2

One can immediately see that the Lazy Linked List is not concurrency optimal. Indeed, consider the schedule depicted in Figure 2. Two operations, insert(1) and insert(2) are concurrently applied to the list containing a single node X_1 storing value 1. Both operations first read h, the head of the list, then operation insert(2) reads node X_1 and creates a new node, X_2, storing 2. Immediately after that, operation insert(1) reads X_1 and returns *false*.

The schedule is correct: it is linearizable and locally serializable. However, it cannot be accepted by the Lazy Linked List, as insert(1) needs a lock on X_1 previously acquired by insert(2). Thus, the implementation is concurrency sub-optimal: an operation may engage in synchronization mechanisms even if it is not going to update the list.

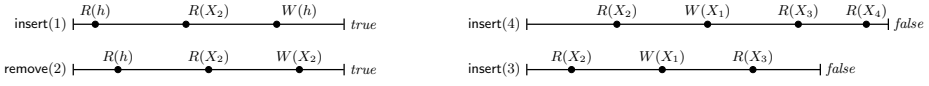

Fig. 3: A schedule rejected by the Harris-Michael Linked List; the initial state of the list is $\{X_2, X_3, X_4\}$; each X_i stores value i; note that not all schedules are depicted for succinctness.

Harris-Michael Linked List. Like the Lazy Linked List, the *lock-free* Harris-Michael algorithm (cf. [4, Chapter 9]) separates logical deletion of a node from its physical removal (both steps use *CAS* primitives). If a CAS associated with logical deletion fails, the operation is restarted. Unlike the Lazy Linked List, however, if the physical removal fails (e.g., a concurrent update performed a successful CAS on the preceding node) the operation completes, and unlinking the logically deleted node from the list is then left for a future operation. Every update operation, as it traverses the list, attempts to physically remove such nodes. If the attempt fails, the operation is restarted. The delegation of physical removals to future operations is crucial for lock-freedom: an operation may only be restarted if there is a concurrent operation that took effect, i.e., global progress is made. But, as we show below, this delegation precludes some legitimate schedules.

Strictly speaking, this algorithm is not locally serializable with respect to the sequential implementation *LL*. Indeed, if a remove operation completes after logical deletion, we may not be able to map its steps to a write to a next field of the preceding node without "over-writing" a concurrent update. Therefore, for the sake of concurrency analysis, we consider a variant of *LL*in which remove operations only remove nodes *logically* and physical removals are put to the traversal procedure of future update operations. Now to define the schedule incurred by an execution of the algorithm, we consider the read and write steps that are part of the last traversal of an operation, node creation steps by insert operations, and successful logical deletions by remove operations. However, the Harris-Michael Linked List is not concurrency-optimal even with respect to this adjusted sequential specification.

Consider the schedule depicted in Figure 3. Two operations, insert(1) and remove(2) are concurrently applied to the list containing three nodes, X_2, X_3 and X_4, storing values 2, 3 and 4, respectively. Note that operation remove(2) marks node X_2 for deletion but does not remove it physically by unlinking it from h. (Here we omit steps that are not important for the illustration.) Note that so far the schedule is accepted by the Harris-Michael algorithm: an earlier update of h by operation insert(1) causes the corresponding CAS primitive performed on h by remove(2) to fail.

After the operation completes, we schedule two concurrent operations, insert(4) and insert(3). Suppose that the two operations concurrently read head, X_1 and X_2. As they both witness X_2 to be marked for logical deletion, they both will try to physically remove it by modifying the next field of X_1. We let insert(3) to do it first and complete by reading X_3 and returning *false*. In the schedule depicted in Figure 3, insert(4) also writes to X_1, and then successfully reads X_3 and X_4, and returns *false*. However, in the execution of the Harris-Michael algorithm, the attempt of insert(4) to physically remove X_2 will fail, causing it to restart traversing the list from the head. Thus, this schedule cannot be accepted.

3 The VBL list

In this section, we address the challenges of extracting maximum concurrency from list-based sets and present our VBL list. As we have shown in the previous section, an update in the Lazy Linked List acquires locks on nodes it is about to modify prior to checking the node's state. Thus, it may reject a correct schedule in which the operation does not modify the list. The schedule rejected by the Harris-Michael Linked List (Figure 3) is a bit more intricate: it exploits the fact that Harris-Michael List involves *helping* which in turn induces additional synchronization steps leading to rejection of correct schedules.

Deriving a concurrency-optimal list requires introducing *value-based* node validation along with the logical-deletion technique. This observation inspired our *value-aware try-lock*.

3.1 Value-aware try-lock

The class Node now contains the fields: (i) val for the value of the node; (ii) next providing a reference to the next node in the list; (iii) a boolean deleted to indicate a node to be marked for deletion and (iv) a lock to indicate a mutex associated with the node.

The value-aware try-lock supports the following operations:

(1) lockNextAt(Node node) first acquires the lock on the invoked node, checks if the node is marked for deletion or if the next field does not point to the node passed as an argument, then releases the lock and returns *false*; otherwise, the operation returns *true*.

(2) lockNextAtValue(V val) acquires the lock on the invoked node, checks if the node is marked for deletion or if the value of the next node is not val, then releases the lock and returns *false*; otherwise returns *true*.

3.2 VBL list

We now describe our VBL implementation. The list is initialized with 2 nodes: *head* (storing the minimum sentinel value) and *tail* (storing the maximum value), *head.next* stores the pointer to *tail*, both *deleted* flags are set to *false*. The pseudo-code is presented in Figure 2.

Contains. The contains(v) algorithm starts from the *head* node and follows *next* pointers until it finds a node with the value that is equal to or bigger than v. Then, the algorithm simply compares the value in the found node with v.

Inserting a node. The algorithm of insert(v) starts with the traversal (Line 24) to find a pair of nodes $\langle prev, curr \rangle$ such that $prev.val$ is less than v and $curr.val$ is equal to or bigger than v. The traversal is simple: it starts from *head* and traverses the list in a wait-free manner until it finds the desired nodes. If $curr.val$ is equal to v (Line 25) then there is no need to insert. Otherwise, the new node with value v should be between *prev* and *curr*. We create a node with value v (Lines 26-27). Then, the algorithm locks *prev* and checks that it still can insert

the node correctly (Line 28): *prev.next* still equals to *curr* and *prev* is not marked as deleted. If both of these conditions are satisfied, the new node can be linked. Otherwise, it cannot: the correctness of the algorithm (namely, linearizability) would be violated; so the operation restarts from the traversal (Line 24). Note that to improve the performance, the algorithm starts the traversal not from *head* but from *prev*.

Removing a node. The algorithm of remove(v) follows the lines of insert(v): first it finds the desired pair of nodes $\langle prev, curr \rangle$. If *curr.val* is not equal to v then there is nothing to remove (Line 36). Otherwise, the algorithm has to remove the node with value v. At first, it takes the lock on *prev* and checks two conditions (Line 39): *prev.next.val* equals to v and *prev* is not marked as deleted. The first condition ensures concurrency-optimality by taking care of the scenario described above: one could have removed and inserted v while the thread was asleep. The second condition is necessary to guarantee correctness, i.e., the node *next* is not linked to *deleted* node, which might result in a "lost update" scenario. If any of the conditions is violated, the algorithm restarts from Line 35. Then, the algorithm takes the lock on *curr* = *prev.next* and checks a condition *curr.next* equals to *next* in Line 41 (note that the second condition is satisfied by the lock on *prev* as *curr* is not marked as deleted). This condition ensures correctness: otherwise, the link *next* to *prev* will be incorrect. If it is not satisfied, the algorithm restarts from Line 35. Afterwards, the algorithm sets *curr.deleted* to *true* (Line 44) and unlinks *curr* (Line 45).

Correctness. We show that the VBL list accepts only correct schedules of the list-based set. We then show that the VBL list accepts *every* correct schedule of the list-based set, thus establishing its concurrency-optimality.

Theorem 1. *Every schedule of the VBL list is linearizable w.r.t the set.*

The full proof is deferred to the companion technical report. Observe that the only nontrivial case to analyse for proving deadlock-freedom is the execution of the update operations. Suppose that an update operation π fails to return a matching response after taking infinitely many steps. However, this means that there exists a concurrent insert or remove that successfully acquires its locks and completes its operation, thus implying progress for at least one correct process.

Theorem 2. *The VBL implementation accepts only correct list-based set schedules locally serializable (wrt LL).*

Concurrency-optimality. We prove that the VBL accepts every correct interleaving of the sequential code. The goal is to show that any finite schedule rejected by our algorithm is not correct. Recall that a correct schedule σ is locally serializable and, when extended with all its update operations completed and contains(v), for any $v \in \mathbb{Z}$, we obtain a linearizable schedule.

Note that given a correct schedule, we can define the contents of the list from the order of the schedule's write operations. For each node that has ever been created in this schedule, we derive the resulting state of its *next* field from the last write in the schedule. Since in a correct schedule each new node is first

Algorithm 2 `VBL` list

```
1:  Shared variables:
2:    head.val ← −∞
3:    tail.val ← +∞
4:    head.next ← tail
5:    head.deleted ← false
6:    tail.deleted ← false
7:    head.lock ← new Lock()
8:    tail.lock ← new Lock()

9:  contains(v):
10:   curr ← head
11:   while curr.val < v do
12:     curr ← curr.next
13:   return curr.val = v

14: waitfreeTraversal(v, prev):
15:   if prev.deleted then
16:     prev ← head
17:   curr ← prev.next
18:   while curr.val < v do
19:     prev ← curr
20:     curr ← curr.next
21:   return ⟨prev, curr⟩

22: insert(v):
23:   prev ← head
24:   ⟨prev, curr⟩ ← waitfreeTraversal(v, prev)
25:   if curr.val = v then return false
26:   newNode.val ← v
27:   newNode.next ← curr
28:   if not prev.lockNextAt(curr) then
29:     goto Line 24
30:   prev.next ← newNode
31:   prev.lock.unlock()
32:   return true

33: remove(v):
34:   prev ← head
35:   ⟨prev, curr⟩ ← waitfreeTraversal(v, prev)
36:   if curr.val ≠ v then
37:     return false
38:   next ← curr.next
39:   if not prev.lockNextAtValue(v) then goto Line 35
40:   curr = prev.next
41:   if not curr.lockNextAt(next) then
42:     prev.unlock()
43:     goto Line 35
44:   curr.deleted ← true
45:   prev.next ← curr.next
46:   curr.lock.unlock()
47:   prev.lock.unlock()
48:   return true
```

created and then linked to the list, we can reconstruct the *state of the list* by iteratively traversing it, starting from the *head*.

Theorem 3 (Optimality). *`VBL` implementation accepts all correct schedules.*

4 Experimental evaluation

Experimental setup. In this section, we compare the performance of our solution to two state-of-the-art list-based set algorithms written in different languages (Java and C++) and on two multicore machines from different manufacturers: A 4-socket Intel Xeon Gold 6150 2.7 GHz server (Intel) with 18 cores per socket (yielding 72 cores in total), 512 Gb of RAM, running Debian 9.9. This machine has OpenJDK 11.0.3; A 4-socket AMD Opteron 6276 2.3 GHz server

(AMD) with 16 cores per socket (yielding 64 cores in total), running Ubuntu 14.04. This machine has OpenJDK 1.8.0_222 (We delegate the AMD results to the tech report).

Concurrent list implementations. We compared our `VBL` algorithm (VBL) to the lock-based Lazy Linked List (Lazy) [3] and Harris-Michael's non-blocking list (Harris-Michael) [5,6] with its wait-free and RTTI optimization suggested by Heller et al. [3] using the Synchrobench benchmark suite [11]. To compare these algorithms on the same ground we primarily used Java as it is agnostic of the underlying set up. The evaluation of the C++ implementations of these algorithms is deferred to the companion technical report [10]. The code of the implementations is part of Synchrobench at `https://github.com/gramoli/synchrobench`.

Experimental methodology. We considered the following parameters:

- **Workloads.** Each workload distribution is characterized by the percent $x\%$ of update operations. This means that the list will be requested to make $(100 - x)\%$ of contains calls, $x/2\%$ of insert calls and $x/2\%$ of remove calls. We considered three different workload distribution: 0%, 20%, and 100%. Percentages 0% and 100% were chosen as the extreme workloads, while 20% update ratio corresponds to the standard load on databases. Each operation contains, insert, and remove chooses its argument uniformly at random from the fixed key range.
- **List size.** On the workloads described above, the size of the list depends on the range from which the operations take the arguments. Under the described workload the size of the list is approximately equal to the half of the key range. We consider four different key ranges: 50, 200, $2 \cdot 10^3$, and $2 \cdot 10^4$. To ensure consistent results we pre-populated the list: each element is present with probability $\frac{1}{2}$.
- **Degree of contention.** This depends on the number of cores in a machine. We take enough points to reason about the behavior of the curves.

Results. We run experiments for each workload 5 times for 5 seconds with a warm-up of 5 seconds. Figure 4 contains the results on Intel machine. Our new list algorithm outperforms both Harris-Michael's and the Lazy Linked List algorithms, and remains scalable except for the situation with very high contention, i.e., high update ratio with small range. We find this behavior normal at least in our case, since the processes contend to get the cache-lines in exclusive mode and this traffic becomes the dominant factor of performance in the execution.

Comparison against Harris-Michael. Harris-Michael's algorithm in general scales well and performs well under high contention. Even though the three algorithms feature the wait-free contains, our original implementation of the Harris-Michael's contains was slower than the other two. The reason is the extra indirection needed when reading the *next* pointer in the combined *pointer-plus-boolean* structure. To avoid reading an extra field when fetching the Java AtomicMarkableReference we implemented the run-time type identification (RTTI) variant with two subclasses that inherit from a parent node class and that represent the marked and unmarked states of the node as previously sug-

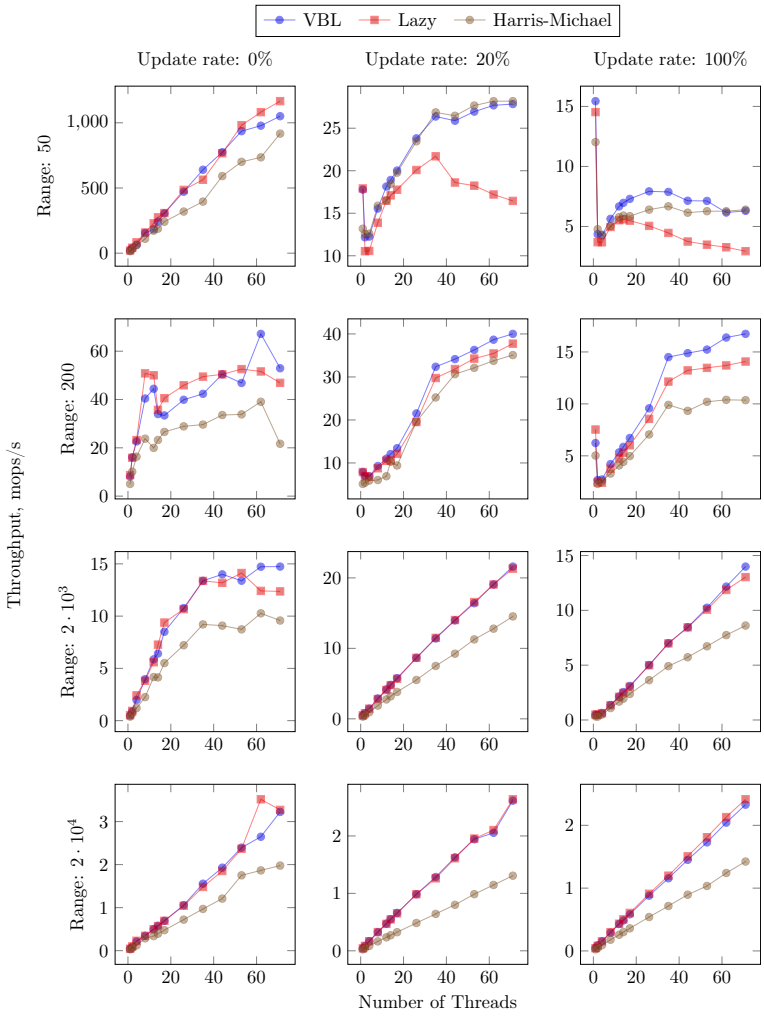

Fig. 4: Evaluation on Intel

gested [3]. This optimization requires, on the one hand, that a remove casts the subclass instance to the parent class to create a corresponding node in the marked state. It allows, on the other hand, the traversal to simply check the mark of each node by simply invoking instanceof on it to check the subclass the node instantiates. As we see, Harris-Michael's algorithm has very efficient updates because it only uses *CAS*, however it spends much longer on list traversals.

Comparison against the Lazy Linked List. The Lazy Linked List has almost the same performance as our algorithm under low contention because both algorithms share the same wait-free list traversal with zero overhead (as the sequential code does) and for the updates, when there is no interference from concurrent operations, the difference between the two algorithms becomes neg-

ligible. The difference comes back however as the contention grows. The Lazy Linked List performance drops significantly due to its intense lock competition (as briefly explained in Section 1). By contrast, there are several features in our implementation that reduce significantly the amount of contention on the locks. We observed a tremendous increase in execution time for the Lazy Linked List because of the contention on locks.

5 Related work and concluding remarks

List-based sets. Heller *et al.* [3] proposed the Lazy Linked List and mentioned the option of validating prior to locking, and using a single lock within an insert. One of the reasons why our implementation is faster than the Lazy Linked List is the use of a novel value-aware try-lock mechanism that allows validating before acquiring the lock.

Harris [5] proposed a non-blocking linked list algorithm that splits the removal of a node into two atomic steps: a logical deletion that marks the node and a physical removal that unlinks the node from the list. Michael [6] proposed advanced memory reclamation algorithms for the algorithm of Harris. In our implementation, we rely on Java's garbage collector for memory reclamation [13]. We believe that our implementation could outperform Michael's variant for the same reason it outperforms Harris' one because it does not combine the logical deletion mark with the next pointer of a node but separates metadata (logical deletion and versions) from the structural data (check [4] for variants of these list-based sets). Fomitchev and Ruppert [14] proposed a lock-free linked list where nodes have a backlink field that allows to backtrack in the list in case a conflict is detected instead of restarting from the beginning of the list. Its *contains* operation also helps remove marked nodes from the list. Gibson and Gramoli [15] proposed the *selfish linked list*, as a more efficient variant of this approach with the same amortized complexity, relying on wait-free *contains* operations. These algorithms are, however, not concurrency-optimal: schedule constructions similar to those outlined for the Harris-Michael and Lazy linked lists apply here.

Concurrency metrics. 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 [16] use the metric to capture the parallelism of a locking scheme. While acknowledging that the metric is theoretical, they insist that it may have "practical significance as well, if the schedulers in question have relatively small scheduling times as compared with waiting and execution times." Herlihy [17] employed the metric from [16] to compare various optimistic and pessimistic synchronization techniques using commutativity [18] of operations constituting high-level transactions. A synchronization technique is implicitly considered in [17] as highly concurrent, namely "optimal", if no other technique accepts more schedules. In contrast to [16, 17], we focus here on a *dynamic* model where the scheduler cannot use the prior knowledge of all the shared addresses to be accessed.

Optimal concurrency, originally introduced in [1], can also be seen as a variant of metrics like *permissiveness* [19] and *input acceptance* [20] defined for transac-

tional memory. The concurrency framework considered in this paper though is independent of the synchronization technique and, thus, more general. Our notion of local seriazability, also introduced in [1], is also reminiscent to the notion of *local linearizability* [21].

Concurrent interleavings of *sequential* code has been used as a base-line for evaluating performance of search data structures [22]. Defining *optimal concurrency* as the ability of accepting *all* correct interleavings has been originally proposed and used to compare concurrency properties of optimistic and pessimistic techniques in [1].

The case for concurrency-optimal data structures. Intuitively, the ability of an implementation to successfully process interleaving steps of concurrent threads is an appealing property that should be met by performance gains.

In this paper, we support this intuition by presenting a concurrency-optimal list-based set that outperforms (less concurrent) state-of-the-art algorithms. Does the claim also hold for other data structures? We believe that generalizations of linked lists, such as skip-lists or tree-based dictionaries, may allow for optimizations similar to the ones proposed in this paper. The recently proposed concurrency-optimal tree-based dictionary [9] justifies this belief. This work presents the opportunity to construct a rigorous methodology for deriving concurrency-optimal data structures that also perform well.

Also, there is an interesting intermingling between progress conditions, concurrency properties, and performance. For example, the Harris-Michael algorithm is superior with respect to both the Lazy Linked List and `VBL` in terms of progress (lock-freedom is a strictly stronger progress condition than deadlock-freedom). However, as we observe, this superiority does not necessarily imply better performance. Improving concurrency seems to provide more performance benefits than boosting liveness. Relating concurrency and progress in concurrent data structures remains an interesting research direction.

References

1. Gramoli, V., Kuznetsov, P., Ravi, S.: In the search for optimal concurrency. In: SIROCCO. Volume 9988. (2016) 143–158
2. Sutter, H.: Choose concurrency-friendly data structures. Dr. Dobb's Journal (June 2008)
3. Heller, S., Herlihy, M., Luchangco, V., Moir, M., Scherer, W.N., Shavit, N.: A lazy concurrent list-based set algorithm. In: OPODIS. (2006) 3–16
4. Herlihy, M., Shavit, N.: The art of multiprocessor programming. Morgan Kaufmann (2012)
5. Harris, T.L.: A pragmatic implementation of non-blocking linked-lists. In: DISC. (2001) 300–314
6. Michael, M.M.: High performance dynamic lock-free hash tables and list-based sets. In: SPAA. (2002) 73–82
7. Guerraoui, R., Kapalka, M.: Principles of Transactional Memory,Synthesis Lectures on Distributed Computing Theory. Morgan and Claypool (2010)
8. Herlihy, M., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3) (1990) 463–492

9. Aksenov, V., Gramoli, V., Kuznetsov, P., Malova, A., Ravi, S.: A concurrency-optimal binary search tree. In: Euro-Par. (2017) 580–593
10. Aksenov, V., Gramoli, V., Kuznetsov, P., Ravi, S., Shang, D.: A concurrency-optimal list-based set. CoRR **abs/1502.01633** (2021)
11. Gramoli, V.: More than you ever wanted to know about synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorithms. In: PPoPP. (2015) 1–10
12. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann (1996)
13. Sun Microsystems: Memory Management in the Java HotSpot Virtual Machine. (April 2006) `http://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf`.
14. Fomitchev, M., Ruppert, E.: Lock-free linked lists and skip lists. In: PODC. (2004) 50–59
15. Gibson, J., Gramoli, V.: Why non-blocking operations should be selfish. In: DISC, Springer (2015) 200–214
16. Kung, H.T., Papadimitriou, C.H.: An optimality theory of concurrency control for databases. In: SIGMOD. (1979) 116–126
17. Herlihy, M.: Apologizing versus asking permission: optimistic concurrency control for abstract data types. ACM Trans. Database Syst. **15**(1) (1990) 96–124
18. Weihl, W.E.: Commutativity-based concurrency control for abstract data types. IEEE Trans. Comput. **37**(12) (1988) 1488–1505
19. Guerraoui, R., Henzinger, T.A., Singh, V.: Permissiveness in transactional memories. In: DISC. (2008) 305–319
20. Gramoli, V., Harmanci, D., Felber, P.: On the input acceptance of transactional memory. Parallel Processing Letters **20**(1) (2010) 31–50
21. Haas, A., Henzinger, T.A., Holzer, A., Kirsch, C.M., Lippautz, M., Payer, H., Sezgin, A., Sokolova, A., Veith, H.: Local linearizability for concurrent container-type data structures. In: CONCUR 2016. Volume 59. (2016) 6:1–6:15
22. David, T., Guerraoui, R., Trigonakis, V.: Asynchronized concurrency: The secret to scaling concurrent search data structures. In: ASPLOS. (2015) 631–644