Linearizable Iterators

Eli Rosenthal
Supervised by Maurice Herlihy

Abstract

Petrank et. al. [5] provide a construction of lock-free, linearizable iterators for lock-free linked lists. We consider the problem of extending this algorithm to arbitrary data-structures, propose sufficient conditions for data-structures admitting a linearizable iterator, and prove this property for a broad class of data-structures. We then show that this sufficient condition is also necessary, leveraging a novel application of combinatorial topology to this area.

1 Background

We consider the setting of iterators for non-blocking (i.e. lock-free or wait-free) data-structures. The key design requirements for these iterators are that they should themselves be lock-free or wait-free, and that they should be *linearizable*; the nodes that an iterating thread observes should correspond to an atomic snapshot of the data-structure at some time between starting and finishing a traversal.

Previous work in [5] provides such a scheme for linked lists and skip lists: they require concurrent mutating threads to *report* the nodes they add or remove from the data-structure to a global snapshot object; when iterating threads finish iterating over the data-structure, they reconcile their local snapshot with the reported mutations in the snapshot object.

In this work we consider the problem of generalizing this methodology to other data-structures. We provide a purely combinatorial property that data-structures must satisfy to admit a linearizable iterator of this kind.

2 Locality: A sufficient condition for linearizble iterators

Definition 1. *A* data-structure (T, r, M, Σ) *is a directed acyclic graph $T \in \mathcal{T}$, where each node has a total order on its outgoing edges, with a distinguished root node r, and with nodes labeled from some set Σ and a set of* mutations $m \in M$ *defined as functions on trees. Define $N(T) \subseteq \Sigma$ to be all of the labels present in the data-structure.*

Without loss of generality we assume each node as a fixed number of outgoing edges, where non-existent edges point to some distinguished node labeled with $\perp$.

We will consider the setting in which in nodes

in data-structures are *uniquely identifiable*, i.e. they behave like pointers and they can not appear more than once in one tree. In addition we consider the case where for all mutations m $N(T) \subseteq N(m(T))$. This models the common assumption of *memory safety*. This assumption is reasonable: if this were not the case, then a node could be freed just as an iterating thread arrived at the node, causing all subsequent operations to result in undefined behavior.

Definition 2. *Given a data-structure* $D = (T, r, M, \Sigma)$ *an* iteration scheme *for* D *is an* iteration algorithm *algorithm* $I : T \times \mathcal{T} \to 2^{\Sigma}$ *and a* report algorithm $R : M \times \mathcal{T} \to 2^{\Sigma}$

We consider the case where I is a standard depth-first search algorithm, though the results throughout are unlikely to change substantially given a different algorithm. We also assume that I starts at r.

Given an execution where there is a finite number of mutations m we have a finite sequence of DAGs, where D_i is the data-structure DAG after mutation i, and a total set of reported nodes $\textsc{Reports}_i = \bigcup_i R(m_i, d_i)$.

Definition 3. *Given a data structure* $D = (T, r, M, \Sigma)$*, define the* paths through a node $n \in T$*, denoted* $\textsc{Paths}(n)$*, as an ordered set of nodes reachable from* n*, ordered by the ordering on the children they include.*

Let $\textsc{Paths}_t(n)$ be the paths through n at time t.

We now define the following safety property

Definition 4. *A mutation occurring at time* t *is* local *if for all* $n \in D_{t-1}$*, we have*

$$\textsc{Paths}_{t-1}(n) \Delta \textsc{Paths}_t(n) \subseteq \textsc{Reports}_t$$

Where Δ is the union over the point-wise symmetric set difference of each set in Paths. In the following sections we show that this property is easy to prove for a number of efficient non-blocking data-structures. First we will show some general properties about iteration schemes of this form:

Definition 5. *A set of reports is* linearizable *if any iteration*[1] *over a data-structure with a finite set of* m *concurrent mutations results in the iterating thread having seen a set equivalent to* D_i *for some* $i \leq m$ *up to adding or removing elements of* $\textsc{Reports}_m$*.*

It turns out that locality is sufficient for this safety property:

Theorem 1. *For a data-structure* D *that supports insertions and deletions (potentially multiple in one atomic step) as possible mutations. If these mutations are local, then the report system of reporting only those nodes that were added or deleted is linearizable.*

Proof We proceed by way of contradiction. In order for this theorem to fail to obtain, it must be possible for an iterator to *fail to see* a node n that was in the data-structure at time 0.[2] Without loss of generality, we say this inconsistency is true for some execution with t mutations, but not $t - 1$. We now consider the set $V_{t-1} = \{v \ : \ \exists p \in \textsc{Paths}_{t-1}(v).\ n \in p, v \in D_{t-1}\}$. If this set is empty, then n is

[1] We still need to define iteration properly, but it can be any traversal that respects the partial order induced by reachability, i.e. it is any traversal that starts at the root and works its way down. I think the proof assumes it operates in a depth-first manner as well, which is standard. It also has to satisfy all of our correctness properties for when there are no mutations.

[2] If we relax our definitions to include seeing a node twice as a failure, and amend the definition of locality appropriately, then this same proof works for double-counting as well

the root of the tree. We assume that iterators start at the root, so this case is impossible. If this set is nonempty, then the paths containing n at $t-1$ must still contain n at t. So if the iterator algorithm misses n at time t then it must miss n at $t-1$, contradiction. □

2.1 Generalizing this Definition

In many cases, not all nodes traversed in a data-structure are "significant" in terms of the iteration. One common construction is *external* search trees that only hold values at the leaves. In this case, performing non-local mutations on the interior of the tree that leave the leaves intact should be fine! There are a number of ways to accommodate this in the framework outlined above, but the easiest is to simply assume that all internal nodes are included in $\textsc{Reports}_i$.

3 Specific Data-structures

Given the abstract definition of locality above, we can define what it means for iterators to be *easy*. Any data-structure has a linearizable iterator if each mutations results in all nodes being reported. Informally, a mutation that adds or deletes k nodes should result in $O(k)$ nodes being reported. We say a data-structure is *efficiently iterable* if this is the case.

A major advantage of the definition of locality is that it is purely combinatorial, and hence significantly simplifies safety proofs for iterators over existing data-structures. In future work, we will provide proofs and implementations for iterators over specific data-structures. We briefly summarize these results (with Zhiyu Liu, Vikram Saraph and Archita Agarwal) now:

– Brown et al. in [1] specify a general framework for implementing non-blocking search trees. In this framework, all mutations consist in picking a subtree of the existing data-structure, building a replacement subtree and atomically modifying the parent pointer and all child pointers using their novel LLX/SCX primitives. These mutations are local *only if* the ordering of children in the new subtree is the same as the original ordering.

– The non-blocking binary search trees in [4] provide only local mutations for essentially the same reason given above: they involve atomically replacing a subgraph in the tree, and they do not alter the ordering of children.

– The hash tables in [3] perform mutations by mutating buckets (called *freezable sets*) that are themselves implemented as immutable set data-structures. Each of these mutations is locality-preserving (locality is trivial in the immutable setting, when all mutations involve a single root pointer-swing), after which it is easy to conclude the entire data-structure has an iterator.

4 The Execution Complex: Locality is an if and only if

A brief discussion What happens when we try to prove that the only data-structures that support a Petrank-style iterator are those that only provide local (or some other property) mutations? Suppose we have a linearizable iterator, we need to consider an arbitrary mutation that does not displace any nodes for an arbitrary depth-first traversal and then prove some common property about these in terms of the two trees.

The if direction is useful because it provides an easy combinatorial property that imposes

structure on something large and unstructured (arbitrary depth-first, wait-free traversals over a data-structure). Analyzing these unstructured traversals an deriving the structure they imply about the data-structure is a fundamentally different problem. We need different tools to cover them.

4.1 The Execution Complex

Recall the definitions of simplicial complex and simplicial join (denoted $*$).

We return to the previous notion of a data-structure and a series of mutations as a sequence of directed graphs, with uniquely identifiable nodes (i.e. labels). We will use this as a basis for higher-dimensional structure in an *execution complex*. We imagine each of these graphs arranged in a tower, stacked on top of one another. Then, we intuitively think of a specific traversal as a path in the initial graph (D_0) that occasionally jumps to it's equivalent node at some D_i until the iteration is finished. More precisely, we add edges to a node $n \in D_i$ for each corresponding outgoing edge in $D_{j>i}$; call these edges *nondeterministic*. Importantly, these edges are arranged in *columns* corresponding to each child pointer that a node has. We assume every node has a label of the form n_t where n is the uniquely identifiable label of a node, and t corresponds to the timestep in which the node resides. This construction is inspired by that of Erdmann in [2].

We now proceed to to build a simplicial complex from this extended graph. Vertices in the complex correspond to nodes in the graph. A k-simplex corresponds to $k+1$ nodes that an iterator could see when traversing the datastructure. Hence, maximal simplices (facets) of this complex correspond to the possible nodes an iterator can see given these mutations. We now define the complex formally, assuming a left-to-right depth-first traversal[3]:

Definition 6. *Given a hierarchical graph with nondeterministic edges, we build the* execution complex *inductively as follows, starting at the root r of some sub-dag at some level, with $k \in \mathbb{N}$ outgoing columns. Now take the set of k-tuples defined by $C = \{(x_1, \ldots, x_k) : rank(x_i) \leq rank(x_{i+1})\}$, where rank is the index into the corresponding column. The complex is then*

$$EC(r) = r * \left(\bigcup_C EC(x_1) * \cdots * EC(x_k) \right)$$

One useful way to think about the execution complex is it is a grouping of all possible combinations of nodes in the graph, except if two nodes are in the same column, or if they could not be seen due to the iteration order. Note that this model generalizes easily to multiple iterators, simply by changing the definition of C.

Theorem 2. *There is a bijective correspondence between possible traversals over a data-structure with m mutations, and facets in the execution complex.*

Given a left-to-right depth-first traversal, a thread must start at the root of the tree, and the times it begins traversing subtrees are only subject to the constraint that it sees leftmost subtrees before rightmost ones. This is exactly the rule given for constructing EC, as the simplicial join adds the root node to all simplices in the subcomplexes. The theorem therefore follows by induction over the levels of the tree. □

Reasons to be skeptical There are definite reasons to be skeptical of this defini-

[3]Note that we can relax this in many natural ways, and that the choice of left-to-right is arbitrary.

tion here. The execution complex is huge: it grows exponentially in size with the number of mutations. It is also very unstructured: there are no holes in this complex, so many common topological methods are useless here. What is useful here is that it gives a nice combinatorial description of what local mutations are: they are mutations that add or subtract vertices from a subcomplex of the deleted join of a complex. The fact that the combinatorial notion of join and deleted join are sufficiently well-behaved means we can characterize the structure of any well-behaved mutation much more directly than before.

Linearizable Mutations What are the good mutations in this model? Consider the execution complex truncated to time $t-1$ compared with time t. Note first that $t-1$ is a subcomplex of time t, we can always finish early. What are the simplices that are added? If the mutation is "good" then all facets of the new complex should only differ with facets of the old complex by adding or removing reported nodes. In other words, some facets may add or remove some number of vertices. Furthermore, these can be interleaved with nodes at different timesteps. These are the only good mutations by this definition, as if there is a facet that differs by unreported nodes from others, then it is possible to have conflicting views in a complete traversal, assuming a Petrank-style construction. [4]

Theorem 3. *Linearizable mutations are local mutations.*

Proof Given theorems 1 and 2, we have already shown locality implies linearizability. The remaining step is to show the other direction. Observe that paths of length k correspond to $k-1$-simplices (or sets of them, over time). For any node n and time t consider the facets containing n_t and n_{t+1}. Each of these must contain the simplices corresponding to each member of $\textsc{Paths}_t(n)$ and $\textsc{Paths}_{t+1}(n)$, because it is possible for iteration to finish at each of those times. However, the facets differ only by reported nodes, because they are linearizable. Hence none of the paths can either; we conclude that these mutations are local. □

5 Conclusion

We have developed a novel necessary and sufficient criterion for linearizable iteration for a lock-free data-structure. This involved the introduction of the notion of *execution complex* which could be useful in other settings. Future work could involve generalizing this notion to weaker conditions than linearizability.

References

[1] Trevor Brown, Faith Ellen, and Eric Ruppert. A general technique for non-blocking trees. In *ACM SIGPLAN Notices*, volume 49, pages 329–342. ACM, 2014.

[2] Michael Erdmann. On the topology of discrete strategies. *The International Journal of Robotics Research*, 2009.

[3] Yujie Liu, Kunlong Zhang, and Michael Spear. Dynamic-sized nonblocking hash tables. In *Proceedings of the 2014 ACM symposium on Principles of distributed computing*, pages 242–251. ACM, 2014.

[4] Note that this isn't quite the definition of linearizability, so we should maybe rename this. However, for the Petrank model of an iterator I believe the case that facets by time t linearize to $t-k$ is unrealistic.

[4] Aravind Natarajan and Neeraj Mittal. Fast concurrent lock-free binary search trees. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 317–328, New York, NY, USA, 2014. ACM.

[5] Erez Petrank and Shahar Timnat. *Distributed Computing: 27th International Symposium, DISC 2013, Jerusalem, Israel, October 14-18, 2013. Proceedings*, chapter Lock-Free Data-Structure Iterators, pages 224–238. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.