Mindicator: A Nonblocking Set Optimized for Querying the Minimum Value

Yujie Liu and Michael Spear

Department of Computer Science and Engineering
Lehigh University

Technical Report LU-CSE-11-001

Abstract

We present the Mindicator, a set implementation customized for shared memory runtime systems. The Mindicator is optimized for constant-time querying of its minimum element, while ensuring scalability as the number of threads adding and removing elements from the set grows. We introduce a lock-free Mindicator and prove its correctness. We also introduce lock-based and quiescently consistent Mindicators, and show that all provide good scalability on both the Intel x86 and Sun SPARC platforms. The strong performance of Mindicators make them suitable for use in high-performance run-time systems, such as garbage collectors, transactional memory, and operating system kernels.

1 Introduction

Shared memory run-time systems, such as garbage collectors (GC) and transactional memory (TM) [6], often require global coordination. To keep costs low, designers of these systems identify a tradeoff that can prevent bottlenecks without affecting the common case, usually by optimizing the run time of one operation at the expense of other operations. The following variant is particularly interesting:

- There exists some set of states S, and a total order $<_t$ on the elements in S.
- There is a set of threads, T, and every thread $t_k \in T$ is always in exactly one state.
- $|S|$ is significantly larger than $|T|$, and multiple threads can be in the same state.
- The operation to optimize is a query that returns the minimum over all threads' states.

This characterization applies to recent problems encountered by Marathe et al. [13], Menon et al. [15], and Koskinen et al. [9], where S is the set of possible start times for transactions in a TM system. An efficient solution to this problem can replace existing mechanisms for ensuring progress or preventing starvation in TM [4, 18]. The formulation also describes the Epoch mechanism used to prevent early reclamation in TM [3, 8], read-copy-update (RCU) synchronization [14], and sequence lock-based critical sections [10], and also expresses the safety condition governing when some garbage collectors may run (e.g., in Python, where GC cannot run if any thread is executing C-based library code).

In existing work, library designers have chosen to (a) pay the overhead of a high latency solution (such as a sorted, lock-based doubly linked list), (b) rely on domain-specific knowledge (such as the property that thread states increase monotonically [8], and are perhaps even globally consistent [15]), or (c) accept unpredictable overheads (such as unbounded heap consumption [8]). Known solutions have either $O(|T|)$ query complexity and $O(1)$ state-change cost, or $O(1)$ query cost and $O(|T|)$ state-change cost.

set S' holds tuples of form (tid, val)
$\triangleright$ initially $S' = \{(-1, \top)\}$

procedure $Arrive(v)$ invoked by thread $\mathbf{t}$
$\triangleright$ add $(\mathbf{t}, v)$ to S'

function $Query() : \mathbb{N}$
$\triangleright$ **return** minimum val field over tuples in S'

procedure $Depart()$ invoked by thread $\mathbf{t}$
$\triangleright$ remove the tuple with $tid = \mathbf{t}$ from S'

Figure 1: Mindicator specification. Well-formedness requires that for any thread $\mathbf{t}$, the number of `Arrive()` operations invoked at any point in time is at most one greater than the number of `Depart()` operations invoked by $\mathbf{t}$, and no less than the number of `Depart()` operations invoked by $\mathbf{t}$.

Our solution is inspired by the SNZI shared object [1, 2, 11]. The SNZI is a counter-like object, where queries indicate whether the value of the counter is zero or nonzero, but not the precise value of a nonzero counter. One of the innovations in SNZI is to represent the counter as a tree: an increment (or Arrive) operation can be initiated at any node in the tree, with a matching decrement (Depart) by the same thread initiating at the same node. In the common case, operations on a SNZI only interact with a thread-local subset of the nodes of the tree. This keeps costs low by limiting migration of nodes among caches.

Though an unconventional characterization, we can think of the SNZI as tracking the minimum member in the set $S = \{0, 1\}$ where $1 <_t 0$. Under this formulation, incrementing is equivalent to moving a thread to state 1, decrementing is equivalent to moving a thread to state 0, and a query returns 1 if $\exists t \in T.\ t.state = 1$

This paper introduces the Mindicator, a tree-based datastructure optimized for the more general case where $|S|$ is large (i.e., $|S| \approx 2^{32} - 1$). A Mindicator takes $O(lg(T))$ time to change a thread's state, and $O(1)$ time to query the minimum over all thread states. The Mindicator does not require thread states to be coherent, nor does it require them to increase monotonically. The Mindicator is scalable and admits lock-free variants that are either linearizable [7] or quiescently consistent (QC).

We prove the correctness of the lock-free Mindicator, and also evaluate performance on Intel x86 and Sun SPARC architectures. The Mindicator shows good scalability, both in microbenchmarks and when used to implement STM. The QC Mindicator, which is suitable for our STM workload, offers particularly good performance, along with stronger progress guarantees than the state of the art.

2 Specification

Without loss of generality, for a multiset S of elements $\{v_i\}$, the Mindicator is a refined set implementation S' with thread specificity. S' holds tuples $\{u_i\}$ of the form (tid, val). For each element $v_i \in S$, there is a one-to-one corresponding tuple $u_i \in S'$, such that $u_i.val = v_i$ and $u_i.tid$ is the unique identity of the thread that inserted u into the set S'. At any time, the original set S is the projection of S' on the val field.

Every tuple in the Mindicator is unique, but multiple entries may be identical in the val field. We require that no two tuples are identical in the tid field, e.g., a thread may not execute two consecutive `Arrive`s without a `Depart()` in-between, or two consecutive `Depart()` operations. We place no restrictions on the values used in successive `Arrive`s by the same thread.

At any time, any thread may perform a `Query()`. Every `Query()` operation returns a value v such that $\forall u \in S'.\ v \leq u.val$. When the Mindicator is empty, a `Query()` returns $\top$, rather than blocking or causing an exception. The behavior of the Mindicator is specified in Figure 1.

2.1 A Well Formed Mindicator

As a linearizable data structure, we can reason about the correctness of a Mindicator in terms of the histories of method invocations and responses that it allows. A thread $\mathbf{t}$ may perform three operations on a Mindicator: `Arrive()`($A^{\mathbf{t}}(v)$), `Depart()` ($D^{\mathbf{t}}(v)$), and `Query()` ($Q^{\mathbf{t}}(v)$). `Query()` returns a value v, whereas `Arrive()` and `Depart()` add and remove a value v from the set, but do not return a value. When discussing well-formedness, the value parameters v are not of consequence, and are omitted. Ignoring the v parameter, the history H of all operations performed on the Mindicator is constrained only by restrictions on the operations of every given thread. We use $H^{\mathbf{t}}$ to denote the sequence of operations performed by thread $\mathbf{t}$. The constraints we wish to impose are that each `Depart()` by thread $\mathbf{t}$ follows an `Arrive()` by thread $\mathbf{t}$, and that no two `Arrive`s by thread $\mathbf{t}$ occur without an intervening `Depart()`. Accommodating an arbitrary number of `Query()` operations, at any time, yields the following expression:

$$H^{\mathbf{t}} = (Q^{\mathbf{t}})^*(A^{\mathbf{t}}(Q^{\mathbf{t}})^* D^{\mathbf{t}}(Q^{\mathbf{t}})^*)^*(A^{\mathbf{t}})^?(Q^{\mathbf{t}})^*$$

In this expression, $(A^{\mathbf{t}})^?$ indicates that the last non-query operation performed by a thread might be an `Arrive()`. We refer to an `Arrive()` that is never followed by a `Depart()` in history H as *unpaired*. In a *closed* history, a suffix of `Depart()` operations are added to H to match all unpaired `Arrive`s.

2.2 Correctness Criteria

We assume the existence of an implicit thread ($tid = -\mathbf{1}$), which performs a single `Arrive()` operation during Mindicator initialization, using the distinguished value $\top$:

$$H^{-\mathbf{1}} = A^{-\mathbf{1}}(\top)$$

We define the (changing) set of live values V_{live} as corresponding to unpaired `Arrive`s. If, for thread $\mathbf{t}$, the history has a suffix $S(H^{\mathbf{t}})$ of the form of $(A^{\mathbf{t}}(v))(Q^{\mathbf{t}})^*$, then $v \in V_{live}$ at the instant of $Q^{\mathbf{t}}$.

In an interleaved history of all threads H, there exists an equivalent sequential history H_S such that the return value of every operation is identical in both histories, the order within each thread history is identical in both histories, but in H_S, no two operations overlap in time. Our correctness criteria is that for every `Query()` operation $Q^{\mathbf{t}}(v)$ in H_S, v is the minimum value of V_{live} at the instant when $Q^{\mathbf{t}}(v)$ appears in H_S. We split this into two sub-criteria:

Safety: The value returned by a `Query()` operation is *no greater than* the minimum value in the live set. That is, for every $Q^{\mathbf{t}}(v)$, $\forall v' \in V_{live}.\ v \leq v'$. This guarantee ensures that the value returned by a `Query()` is no larger than the minimum value in the Mindicator at the time of the `Query()` operation. Thus when a `Query()` returns some value k, $\forall j < k.$ no thread is storing value j.

Liveness: The value returned by a `Query()` operation corresponds to *some* value in the live set. That is, for every $Q^{\mathbf{t}}(v)$, $\exists v' \in V_{live}.\ v = v'$. Without this criteria, 0 would be an acceptable result for all `Query()` operations. With this criteria, a `Query()` must return a value that is stored in the Mindicator. Thus when a `Query()` returns some value k, there is a thread whose current value is no greater than k.

```
datatype NODE
  sta  : B   ▷ state bit
  min  : N   ▷ minimum of children
  ver  : N   ▷ version number

initially NODE (sta, min, ver)
  = (steady, ⊤, 0)

procedure Arrive(X : NODE, n : N)
 1: while true do
 2:     x ← Read(X)
 3:     if x.min > n or x.sta = tentative
 4:         break
 5:     if CAS(X, x, (steady, x.min, x.ver + 1))
 6:         return
 7: while true do
 8:     x ← Read(X)
 9:     if x.min ≤ n
10:         break
11:     if CAS(X, x, (tentative, n, x.ver + 1))
12:         x ← (tentative, n, x.ver + 1)
13:         break
14: if x.sta = tentative
15:     Arrive(parentof(X), n)
16:     if x.min = n
17:         CAS(X, x, (steady, n, x.ver + 1))

procedure Depart(X : NODE, n : N)
18: Revisit(X)
19: x ← Read(X)
20: if x.min < n and x.sta = steady
21:     return
22: Depart(parentof(X), n)

procedure Revisit(X : NODE)
23: while true do
24:     x ← Read(X)
25:     min ← ⊤
26:     if x.sta = tentative
27:         return
28:     for C in childrenof(X) do
29:         c ← Read(C)
30:         if c.min < min
31:             min ← c.min
32:     if min < x.min
33:         if CAS(X, x, (tentative, min, x.ver + 1))
34:             return
35:     elif CAS(X, x, (steady, min, x.ver + 1))
36:         return

function Query(X : NODE) : N
37: return Read(X).min
```

Figure 2: The Lock-Free Mindicator Algorithm

3 A Lock-Free Tree-Based Mindicator

We present a tree-based implementation that conforms to the specification in Figure 1. Each thread is assigned a unique leaf node, where it begins `Arrive`s and `Depart`s. `Arrive`s and `Depart`s propagate upward, transmitting values from children to parents, until they reach a "turning point". Then the operations regress downward toward along the same path. `Query`s access only the root node. Each leaf can be read by many threads, but is only modified by one thread. Nodes can have any number of children.

Figure 2 presents our lock-free tree-based Mindicator algorithm. The existence of a logical thread storing the value $\top$ is achieved by having threads change their values to $\top$ in the course of a `Depart()` operation. We assume that the `childrenof` operator returns an empty set when applied to leaf nodes, and when X is the root node, a method invoked on `parentof(X)` will return immediately.

3.1 Types and Operations

The Mindicator is a tree composed of Node objects, where each node contains a (possibly empty) list of child nodes, a (possibly empty) pointer to a parent node, and a CAS object.[1] We treat the value stored by the CAS object as a tuple, consisting of an integer summary value (`min`), a state bit (`sta`), and a version

[1] A compare-and-swap object, or CAS object, supports two operations, both of which are atomic: a read, which returns the current value of the object; and a compare-and-swap operation. The compare-and-swap operation takes two parameters, `old` and `new`, and sets the object value to `new` if and only if the value object's equals `old` when the atomic CAS operation is performed.

number (`ver`). The `min` field summarizes the smallest summary value of all children of a Node. The `sta` bit indicates if that value is being propagated upward (in which case it is "tentative", not yet "steady"). The `ver` counter increments by one on every update to the Node, in order to prevent ABA problems and simplify the task of atomically summarizing the values of a node's children. All node objects are initialized to a summary value of $\top$ and a clear `sta` bit.

A `Query()` operation on a Node simply returns the `min` value of the node. Queries performed on the root of a Mindicator tree return the smallest value held in the Mindicator.

The `Arrive()` operation adds a value to a Mindicator. It traverses up the tree, starting at a leaf, and ending at some "turning point." Then, it traverses downward from the turning point to the leaf at which it began. We refer to these phases as the "climbing" and "regressing" phases, respectively. In the climbing phase, the thread writes its value at nodes in order to lower their summary value, and marks those values as tentative. The turning point occurs either when the root is accessed, or when the `Arrive()` reaches a node whose value is steady and $\leq$ the arriver's value. The thread then regresses, clearing the tentative bits it marked in its climbing phase. As discussed later, an `Arrive()` with value i to a node that holds tentative value j will *steal* the node if $i < j$, by overwriting j and taking responsibility for clearing the `sta` bit. If $i = j$, then either thread may clear the `sta` bit during its regressing phase.

`Depart()` begins at a leaf. It sets the leaf's `val` to $\top$, and then propagates upward. At each level, the `Depart()` operation uses `Revisit()` to update a node by reading all of the node's children, and then setting the node's value to the minimum value of all children. As the operation already removed its own value from one of the children, this action serves to remove any copies of the departer's value from nodes in higher levels of the tree, but only when no peer also stores that value. A `Depart()` of value v propagates until it reaches either the root, or some intermediate node with a `steady` value $v' < v$.

Most of the work of a `Depart()` is delegated to the `Revisit()` operation. `Revisit()` calculates the minimum value of a node's children, using the node's `ver` field to ensure that the multi-location read is atomic. Since `Revisit()` only removes `steady` values, it does not modify any node with a tentative value: such a value could not have been written by the departing thread. However, this alone does not suffice as a termination condition, since the `Arrive()` responsible for writing that tentative value may not have completed its climbing phase.

3.2 Concurrent Operations

Here, we briefly summarize the key invariants and interactions between operations. For convenience, we will consider a generic tree where intermediate node I has children X and Q, and X has children Y and Z. Y and Z are leaves assigned to threads `y` and `z`.

The most important property of our algorithm is that if the `sta` bit of node N is set, then it cannot be guaranteed that the `min` value of any ancestor of N is less than or equal to $N.min$. This condition has a strong effect on determining when a recursion must take place. There are two cases. First, let us consider a `Depart()` operation. Suppose a `Depart()` of value z has just updated Node Z ($D^{\mathtt{z}}(z)$) and is now at Node X. If $X.sta$ is not zero, then without looking at X's children, $D^{\mathtt{z}}(z)$ must assume that X may have been updated by an `Arrive()` on node Y with value y ($A^{\mathtt{y}}(y)$). Since X is not steady, $A^{\mathtt{y}}(y)$ may not have updated the value at I yet. Independent of the value of y, it is possible that $I.min = z$, and that $Q.min > z$. Thus, if this `Depart()` does not visit Node I, but instead returns, then it is possible that a subsequent `Query()` ($Q^{\mathtt{z}}()$) would return z, even though the calling thread just removed z from the Mindicator. This would violate liveness.

The second case is similar, but deals with `Arrive()` operations. Suppose that threads Y and Z are simultaneously arriving at X, via operations $A^{\mathtt{y}}(y)$ and $A^{\mathtt{z}}(z)$. If thread Y discovers value z in node X,

and $z \leq y$, then Y need not modify the value at X. However, since $X.sta$ is not zero, again it can be the case that thread Z has not yet propagated its update to I. If all threads other than Y and Z have completed a `Depart()`, then I and all other ancestors could have the value $\top$. If $A^{\mathbf{y}}(y)$ returns without recursing up to Node I, a subsequent `Query()` by thread Y ($Q^{\mathbf{y}}()$) could return $\top > y$, violating safety.

3.3 Cooperation: Helping and Stealing

The above discussion illustrates a performance risk: if an operation delays after setting the `sta` bit of some node X, then all concurrent operations that reach X must traverse upward to I. There are a number of cases where we allow an `Arrive()` or `Depart()` to help or steal work, to include resetting the `sta` bit.

Since a `Depart()` operation does not set the `sta` bit except as part of helping a concurrent arriver, it does not make sense for an `Arrive()` to steal from, or help, a `Depart()`. The only interaction of interest here is that a concurrent `Arrive()` may cause a `Depart()` to fail in the CAS of its `Revisit()` operation, and then discover that no further CAS is needed on the current node (Line 27).

A Departer *can* steal from an Arriver. If $A^{\mathbf{y}}(y)$ and $D^{\mathbf{z}}(z)$ are concurrent, then during its `Revisit()` of X, $D^{\mathbf{z}}(z)$ can "pull" y from Y into X. In addition to updating min on behalf of thread Y, the depart might steal all responsibility for propagation, and for managing the `sta` bit. If $y < z$, then line 33 will execute, setting $X.sta$ and requiring Y to recurse upward. Intuitively, this means that the `Depart()` will not propagate the arrival of y higher than the highest point to which the removal of z must propagate, and thus Y cannot take an early exit. However, if $y \geq z$, or if there exists some other child C of X such that $C.min \leq y$, then thread Z knows that the common ancestor chain of Z and Y currently holds only values that are $\leq y$. Thus Z can clear $X.sta$, knowing that as Z propagates its value upward, it will not ever update a common ancestor of Z and Y to a value larger than y.

Arrivers can steal from each other while propagating values upward. If $A^{\mathbf{y}}(y)$ sees that $X.sta$ is set with $X.min > y$, then there is a concurrent `Arrive()` in another descendant of X. If $A^{\mathbf{y}}(y)$ climbs and then returns to X before the concurrent arriver, Y can clear the $X.sta$ bit: Once y has propagated upward, an invariant that all ancestors are $\leq y$ ensures those ancestors are $\leq$ the old $X.min$. If $X.min = y$, then $A^{\mathbf{y}}(y)$ and the concurrent Arriver at X will help each other to clear the $X.sta$ bit.

4 Correctness

We now provide an outline of the correctness proof of the Mindicator algorithm. The detailed proof appears in Section 5.

4.1 Preliminaries and Definitions

Our proof does not rely on linearization points. As discussed in Section 3, `Arrive()` and `Depart()` operations appear to take effect at the moment where their operands become visible to a concurrent `Query()` operation. Since `Depart()` can propagate a concurrent operation to the root, the linearization point for an operation may not even correspond to an instruction performed by the caller.

Instead, we give a construction procedure to reproduce the linearization order from the interleaved history (which is a partial order on all operations, determined by specific interleavings of execution). The procedure introduces an explicit ordering between concurrent (interleaved) operations. We show that every total order constructed by the procedure 1) preserves the original partial order of operations and 2) is equivalent to some legal sequential history that conforms with the specification in Section 2.

We use an augmented algorithm in the proof to distinguish different `Arrive()` operations with the same number. In the augmented algorithm, threads `Arrive()` and `Depart()` with tokens instead of integers, where a token is the concatenation of the number with a unique id. An `Arrive()` and its pairing `Depart()` take the same token as parameter. The datatype $Node$ is also extended with an auxiliary id field to store the identifiers. Since the augmented algorithm is a refinement of the Mindicator algorithm, the correctness of the augmented algorithm implies the correctness of the original algorithm.

Without loss of generality, we assume that there is a unique leaf node for each thread t, which serves as t's starting point for `Arrive()` and `Depart()` operations. The full path of t is the sequence of nodes on the path from its uniquely assigned leaf node to the root node of the Mindicator.

Since `Arrive()` and `Depart()` are recursive, when a call is made on the ith node of $\mathbf{t}$'s full path, we refer to the sequence of nodes from the leaf to i as the trace, and the nodes from the parent of i to the root as the complementary trace. $(n|\alpha)$ denotes a token with number n and identifier α. $\langle A^{\mathbf{t}}(n|\alpha), D^{\mathbf{t}}(n|\alpha)\rangle$ denotes an `Arrive()` and `Depart()` pair performed by $\mathbf{t}$ with token $(n|\alpha)$. We refer to the ith recursive invocation of an `Arrive()` or `Depart()` as $A_i^{\mathbf{t}}(n|\alpha)$ or $D_i^{\mathbf{t}}(n|\alpha)$. $Q^{\mathbf{t}}(n|\alpha)$ refers to a `Query()` by $\mathbf{t}$ returning token $(n|\alpha)$. We omit α when it is not involved in the discussion.

4.2 Establishing Invariants

The first part of the proof establishes basic properties of the Mindicator data structure. We analyze the behavior of a single `Arrive()` or `Depart()` operation (which still requires reasoning about concurrency) in terms of preconditions and postconditions. We show that the following invariants are established on the trace of an operation before it recurses upward, and then that as the operation unwinds downward, the invariants continue to apply to the complementary trace. In this manner, invariants can be established that hold after the response of an operation.

Lemma. *Before invoking $A_i^{\mathbf{t}}(n)$, for every node O_k in the trace $O_k.min \leq n$.*

Lemma. *After $A_i^{\mathbf{t}}(n)$ returns, for every node O_k in the complementary trace $O_k.min \leq n$.*

Lemma. *Before invoking $D_i^{\mathbf{t}}(n|\alpha)$, for every node O_k in the trace $O_k.id \neq \alpha$.*

Lemma. *After $D_i^{\mathbf{t}}(n|\alpha)$ returns, for every node O_k in the complementary trace $O_k.id \neq \alpha$.*

The above lemmas specify that how the behavior of `Query()` operations can be constrained by non-interleaved pairs of `Arrive()` and `Depart()`operations. If a `Query()` is ordered between the `Arrive()` and `Depart()`, its return value must be no greater than the number of the `Arrive()`(guaranteed by the postcondition of `Arrive()`). Otherwise, if a `Query()` is ordered before the `Arrive()` or after the pairing `Depart()`, it cannot observe a token with the same identifier generated by the `Arrive()`(guaranteed by the postcondition of `Depart()`). This provides a "weak" form of the safety and liveness conditions (where "weak" refers to the fact that these conditions do not apply to concurrent operations):

Theorem (Weak Safety). *For every $Q(n)$, for every $\langle A(m), D(m)\rangle$ such that $m < n$, $Q(n)$ is **not** ordered between $A(m)$ and $D(m)$.*

Theorem (Weak Liveness). *For every $Q(n|\alpha)$, there exists a unique $\langle A(n|\alpha), D(n|\alpha)\rangle$ and $Q(n|\alpha)$ is **not** ordered before $A(n|\alpha)$ or after $D(n|\alpha)$.*

4.3 Placing Bounds on Linearization Order

Since all externally visible effects relate to `Query()` operations, we use them to approximate the linearization point of an `Arrive()` or `Depart()`. That is, instead of defining accurate linearization points, we assert lower and upper bounds on when an operation takes effect, with respect to the observations by concurrent `Query`s. Informally, an `Arrive()` must have taken effect if its unique token was observed by a concurrent `Query()`(lower bound), and must not have taken effect if a concurrent `Query()` observes some value greater than its token's value (upper bound). Similarly, a `Depart()` must have taken effect if a concurrent `Query()` observes some token with a value greater than its token's value (lower bound), and must not have taken effect if its unique token is observed by a concurrent `Query()`(upper bound).

We term these `Query`s as the bounding operations for an `Arrive()` or `Depart()`. When a `Query()` is determined as the bounding operation of an `Arrive()` or `Depart()`, it enables determination of the ordering between these two operations. For example, if $Q(n)$ and $A(m)$ are concurrent and $n > m$, then $Q(n)$ serves as the lower bound for the linearization point of $A(m)$, in that $A(m)$ must be ordered after $Q(n)$ in the linearization order.

4.4 Constructing a Legal Total Order

The availability of bounding operations permits the creation of a construction procedure to incrementally evolve an initial partial order $\mathcal{R}_2$ into an intermediate relation $\mathcal{R}_2^*$. The procedure elaborates $\mathcal{R}_2$ by iterations. In each iteration, it finds a pair of concurrent operations (in both $\mathcal{R}_2$ and $\mathcal{R}_2^*$) that matches a specified pattern, and assigns an explicit ordering between the operations (by adding it to $\mathcal{R}_2^*$). The procedure terminates when no more pairs can be found. When the procedure terminates, any remaining unordered operations in $\mathcal{R}_2^*$ are irrelevant to the linearization order. For example, if an `Arrive()` and `Depart()` operation introduce and remove a token, but that token is never observed by any `Query()`, then their order relative to concurrent operations (and indeed, their existence) is immaterial to the correctness of the algorithm, and any legal order can be assigned. Therefore, the problem of proving linearizability is reduced to proving the following properties of the construction procedure:

Lemma. *The Weak Safety and Weak Liveness theorems hold as invariants of the construction procedure.*

Lemma. *$\mathcal{R}_2^*$ is a superset of $\mathcal{R}_2$, the initial partial order on all operations.*

Lemma. *$\mathcal{R}_2^*$ forms a directed acyclic graph (DAG), in which the set of vertexes includes all operations, and for two operations op_1 and op_2, there is a directed edge from op_1 to op_2 iff. $op_1 \rightarrow_{\mathcal{R}_2^*} op_2$.*

Lemma. *Every topological order of $\mathcal{R}_2^*$ is a legal sequential history.*

Theorem. *The Mindicator algorithm is linearizable.*

5 Proof of Correctness

In this section, we prove the correctness of the Mindicator algorithm. We first prove properties of the Mindicator data structure, and based on these, we give a *construction procedure* in which we introduce an ordering between interleaved operations. We show that every total order constructed by the procedure is equivalent to some legal sequential history as specified in Section 2.

datatype $\mathbb{TOKEN}$

min	$: \mathbb{N}$	$\triangleright$ *minimum of children*
id	$: \mathbb{N}$	$\triangleright$ *unique timestamp*

datatype $\mathbb{NODE}$

sta	$: \mathbb{B}$	$\triangleright$ *state bit*
tok	$: \mathbb{TOKEN}$	$\triangleright$ *token indirection*
ver	$: \mathbb{N}$	$\triangleright$ *version number*

initially $\mathbb{NODE}\ (sta, (tok.min, tok.id), ver) = (steady, (\top, \phi), 0)$

procedure $Arrive(X : \mathbb{NODE}, n : \mathbb{TOKEN})$

1: **while** $true$ **do**
2: $x \leftarrow Read(X)$
3: **if** $x.tok > n$ **or** $x.sta = tentative$
4: **break**
5: **if** $CAS(X, x, (steady, x.tok, x.ver + 1))$
6: **return**
7: **while** $true$ **do**
8: $x \leftarrow Read(X)$
9: **if** $x.tok \leq n$
10: **break**
11: **if** $CAS(X, x, (tentative, n, x.ver + 1))$
12: $x \leftarrow (tentative, n, x.ver + 1)$
13: **break**
14: **if** $x.sta = tentative$
15: $Arrive(parentof(X), n)$
16: **if** $x.tok = n$
17: $CAS(X, x, (steady, n, x.ver + 1))$

procedure $Depart(X : \mathbb{NODE}, n : \mathbb{TOKEN})$

18: $Revisit(X)$
19: $x \leftarrow Read(X)$
20: **if** $x.tok < n$ **and** $x.sta = steady$
21: **return**
22: $Depart(parentof(X), n)$

procedure $Revisit(X : \mathbb{NODE})$

23: **while** $true$ **do**
24: $x \leftarrow Read(X)$
25: $min \leftarrow (\top, \phi)$
26: **if** $x.sta = tentative$
27: **return**
28: **for** C **in** $childrenof(X)$ **do**
29: $c \leftarrow Read(C)$
30: **if** $c.tok < min$
31: $min \leftarrow c.tok$
32: **if** $min < x.tok$
33: **if** $CAS(X, x, (tentative, min, x.ver + 1))$
34: **return**
35: **elif** $CAS(X, x, (steady, min, x.ver + 1))$
36: **return**

function $Query(X : \mathbb{NODE}) : \mathbb{N}$

37: **return** $Read(X).tok.min$

Figure 3: The Augmented Mindicator Algorithm Used in the Proof

5.1 The Augmented Algorithm

Since the Mindicator algorithm allows duplicated keys, threads can initiate `Arrive`s with the same number. It is critical in the proof that different `Arrive()` operations with the same number can be distinguished from each other. The motivation comes from proving liveness: we need to show that after some thread completes a `Depart()` with number n, the number should not be observed again. If a `Query()` returns n thereafter, we need to show the latter n is "different" from the former one, e.g., it is from an `Arrive()` by another thread.

We use the augmented algorithm in Figure 3 to distinguish different `Arrive()` operations with the same number. In the augmented algorithm, threads arrive and depart with tokens instead of integers. A token is an integer (min) concatenated with an identifier (id). A thread generates a *unique identifier* (not shown in Figure 3) before invoking `Arrive()` on its leaf node. The identifier is attached to the arriver's number to form a token. An `Arrive()` and its pairing `Depart()` take the same token as parameter. We redefine $<, >, \leq, \geq$ and $=$ operators for tokens so that identifiers are ignored in the comparison, e.g., two tokens $tok_1 < tok_2$ iff. $tok_1.min < tok_2.min$, regardless of their ids.

Note that generating a globally unique identifier requires only local computation. For example, we can assign each thread a unique id and maintain a thread-local counter for each thread. The counter is

incremented on each local `Arrive()` operation. Concatenating the unique thread id and the local counter forms a globally unique identifier.

Observation 1. *Each statement of the augmented algorithm preserves the semantics of the corresponding statement in the original algorithm, by applying the following state mapping:*

Variable Type	*Name in Original Algorithm*	*Name in Augmented Algorithm*
parameter of `Arrive()`	n	$n.min$
parameter of `Depart()`	n	$n.min$
local variable of `Revisit()`	min	$min.min$
any global or local node	(sta, min, ver)	$(sta, tok.min, ver)$

Theorem 2. *The correctness of the augmented algorithm implies the correctness of the original algorithm.*

Proof. The augmented Mindicator algorithm is a refinement of the original algorithm. That is, for each execution of the original algorithm, we can perform a stepwise simulation using the augmented algorithm, and reach the exact same states by applying the state mapping in Observation 1. □

5.2 Notation

Without loss of generality, we assume that the values stored in a Mindicator are natural numbers, with a distinguished maximum value $\top$. We also assume that there is a unique leaf node for each thread, which serves as the thread's starting point for `Arrive()` and `Depart()` operations. We use the following notations in the proof: $A^{\mathbf{t}}(n)$ denotes an Arrive operation initiated by thread $\mathbf{t}$ with number n. Since Arrive operations are recursive, we use $A^{\mathbf{t}}_i(n)$ to denote the ith recursive *invocation* within $A^{\mathbf{t}}(n)$, i.e., $A^{\mathbf{t}}_1(n)$ is the invocation that operates on the leaf node assigned to thread $\mathbf{t}$. Thus, an `Arrive()` operation $A^{\mathbf{t}}(n)$ is represented as a collection of invocations: $A^{\mathbf{t}}(n) = A^{\mathbf{t}}_1(n) \subset A^{\mathbf{t}}_2(n) \subset A^{\mathbf{t}}_3(n) \subset \ldots \subset A^{\mathbf{t}}_k(n)$. The $\subset$ operator is used between two successive invocations to reflect that $A^{\mathbf{t}}_{i+1}(n)$ is always subsumed as a sub-invocation of $A^{\mathbf{t}}_i(n)$. On the other hand, we use the $=$ operator between $A^{\mathbf{t}}(n)$ and $A^{\mathbf{t}}_1(n)$, since the invocation on the leaf node is equivalent to the entire `Arrive()` operation. Similarly, $D^{\mathbf{t}}(n)$ denotes a `Depart()` operation initiated by thread $\mathbf{t}$ that removes number n. $D^{\mathbf{t}}_i(n)$ denotes the ith recursive invocation within $D^{\mathbf{t}}(n)$. A `Depart()` operation $D^{\mathbf{t}}(n)$ is represented as a collection of invocations: $D^{\mathbf{t}}(n) = D^{\mathbf{t}}_1(n) \subset D^{\mathbf{t}}_2(n) \subset D^{\mathbf{t}}_3(n) \subset \ldots \subset D^{\mathbf{t}}_k(n)$. $Q^{\mathbf{t}}(n)$ denotes the `Query()` operation of thread $\mathbf{t}$ that returns n.

For simplicity of discussion, we assume the history of each thread is closed, that is, for any `Arrive()` $A^{\mathbf{t}}(n)$, there exists a pairing `Depart()` $D^{\mathbf{t}}(n)$. $\langle A^{\mathbf{t}}(n), D^{\mathbf{t}}(n)\rangle$ denotes such a pair of operations. The pair $\langle A^{-\mathbf{1}}(\top), D^{-\mathbf{1}}(\top)\rangle$ refers to the logical first and last operations in a closed, interleaved thread history.

In the context where identifiers are emphasized, we use the notation $A^{\mathbf{t}}(n|\alpha)$ to denote an `Arrive()` operation that generates the unique identifier α and takes the token $(n|\alpha)$ (i.e., $min = n, id = \alpha$) as parameter. Similar notations apply to `Depart()` and `Query()` operations. A special identifier ϕ is reserved for $A^{-\mathbf{1}}(\top|\phi)$ and $D^{-\mathbf{1}}(\top|\phi)$, the logical first and last operations in the interleaved thread history.

We use O to refer to any instance of an object of type $Node$. To conform with the naming conventions of the original algorithm, we flatten the field structure of the node. The notation $O.tok.min$ is abbreviated to $O.min$, and $O.tok.id$ is abbreviated to $O.id$. The Mindicator is implemented as a rooted tree of these objects. Without loss of generality, we assume that one leaf node is uniquely assigned to each thread.

Definition 3. *R is the root node of the Mindicator tree.*

Definition 4. *The* full path *of thread* $\mathbf{t}$ *(*$\mathbf{F^t}$*) is the sequence of nodes on the path from the leaf node uniquely assigned to* $\mathbf{t}$ *to the root node of the Mindicator. There is a natural order for these nodes, which we define as* $\mathbf{F^t} = \{O_1^\mathbf{t}, O_2^\mathbf{t}, O_3^\mathbf{t}, \ldots, O_N^\mathbf{t}\}$ *where*

- $O_1^\mathbf{t}$ *is the leaf node assigned to* $\mathbf{t}$
- $O_N^\mathbf{t}$ *is the root node* R
- $\forall O_i^\mathbf{t}, O_{i+1}^\mathbf{t} \in \mathbf{F^t}$. $O_{i+1}^\mathbf{t}$ *is the parent node of* $O_i^\mathbf{t}$

When the thread-specificity of a path is clear, we abbreviate $O_i^\mathbf{t}$ as O_i.

Definition 5. *The* trace *of an* `Arrive()` *or* `Depart()` *invocation is the sequence of nodes on the path from the leaf node to the node that is being accessed by the invocation. Formally, the* trace *of* $A_k^\mathbf{t}(n)$ *or* $D_k^\mathbf{t}(n)$ *is the prefix of* $\mathbf{F^t}$ *with length* k, $\mathbf{T}_k^\mathbf{t} = \{O_1, O_2, O_3, \ldots, O_k\}$. *Note that the trace* $\mathbf{T}_0^\mathbf{t}$ *is empty before invoking* $A_1^\mathbf{t}(n)$ *or* $D_1^\mathbf{t}(n)$.

Definition 6. *The* complementary trace *of an* `Arrive()` *or* `Depart()` *invocation is the subset of nodes in the full path that are not in the trace. Formally, the* complementary trace *of* $A_k^\mathbf{t}(n)$ *or* $D_k^\mathbf{t}(n)$ *is the postfix of* $\mathbf{F^t}$ *with length* $N - k$ *(*N *is the length of* $\mathbf{F^t}$*),* $\overline{\mathbf{T}_k^\mathbf{t}} = \{O_{k+1}, O_{k+2}, O_{k+3}, \ldots, O_N\}$. *As a special case,* $\overline{\mathbf{T}_k^\mathbf{t}}$ *is empty when* $A_k^\mathbf{t}(n)$ *or* $D_k^\mathbf{t}(n)$ *is invoked on the root node (*$k = N$*).*

Observation 7. *For an invocation* $A_k^\mathbf{t}(n)$ *or* $D_k^\mathbf{t}(n)$, $\mathbf{T}_k^\mathbf{t} \bigcap \overline{\mathbf{T}_k^\mathbf{t}} = \{\}$. *That is, the trace and complementary trace of an invocation are disjoint.*

Observation 8. *For an invocation* $A_k^\mathbf{t}(n)$ *or* $D_k^\mathbf{t}(n)$, $\mathbf{T}_k^\mathbf{t} \bigcup \overline{\mathbf{T}_k^\mathbf{t}} = \mathbf{F^t}$. *That is, the union of the trace and complementary trace of an invocation is the full path.*

All nodes in the Mindicator tree are modified through CAS instructions and read through $Read$ instructions. The *CAS history* H_O^k is a sequence of CAS instructions $\{CAS_1, CAS_2, CAS_3, \ldots, CAS_k\}$ performed on node O. $O[i]$ refers to the state of node O immediately after $CAS_i \in H_O^k$ is performed. $O[0]$ refers to the initial state of node O. We assume there is a sequentially consistent *total order* on all CAS and $Read$ instructions in the interleaved history of all threads. Note that some architectures guarantee a total order on CAS instructions while $Read$ instructions are (partially) ordered with respect to the CAS instructions. A total order can still be introduced by assigning any order to $Read$ instructions that are otherwise unordered. The superscripted CAS^n and $Read^n$ refer to a CAS or $Read$ instruction at line number n of the algorithm shown in Figure 3.

Definition 9. *Relation* $\mathcal{R}_1$ *(*$\rightarrow_{\mathcal{R}_1}$*) is an irreflexive, sequentially consistent total order on all* $\{CAS, Read\}$ *instructions in the interleaved thread history.*

The *invocation* and *response* points of an `Arrive()`, `Depart()`, or `Query()`operation are the first and last instructions that access the shared memory. These points must be $Read$ or CAS instructions. We use $op.inv$ and $op.res$ to denote the invocation and response points of operation op. By definition, the invocation and response points of each operation are totally ordered under relation $\mathcal{R}_1$. Thus, $\mathcal{R}_1$ dictates a partial order on all the `Arrive()` `Depart()` and `Query()` operations:

Definition 10. *Relation* $\mathcal{R}_2$ *(*$\rightarrow_{\mathcal{R}_2}$*) is an irreflexive partial order on all* {`Arrive()`, `Depart()`, `Query()`} *operations in the interleaved thread history. For operations* $op_1, op_2 \in$ {`Arrive()`, `Depart()`, `Query()`}, $op_1 \rightarrow_{\mathcal{R}_2} op_2$ *iff.* $op_1.res \rightarrow_{\mathcal{R}_1} op_2.inv$.

Definition 11. *For operations* $op_1, op_2 \in$ {`Arrive()`, `Depart()`, `Query()`}, op_1 *and* op_2 *are concurrent in relation* $\mathcal{R}$ *iff.* $op_1 \nrightarrow_{\mathcal{R}} op_2 \wedge op_2 \nrightarrow_{\mathcal{R}} op_1$.

5.3 Properties of CAS Instructions

We now present properties of each CAS instruction in the algorithm. These properties (observations) are independent of the concurrent behavior of the algorithm, which can be verified by the local context of each operation, so we omit their proofs.

Observation 12. *Each CAS instruction in the algorithm increments the version number, and $CAS_i \in H_O^k$ always changes the version number of O from $i-1$ to i.*
$O[i].ver \equiv i$

Observation 13. *CAS^5 keeps the min field unchanged and keeps the sta field $steady$.*
$\forall CAS_i \in H_O^k.\ CAS_i\ is\ CAS^5 \Rightarrow (O[i].min = O[i-1].min) \wedge (O[i].sta = O[i-1].sta = steady)$

Observation 14. *CAS^{11} decreases the min field and sets the sta field to $tentative$.*
$\forall CAS_i \in H_O^k.\ CAS_i\ is\ CAS^{11} \Rightarrow (O[i].min < O[i-1].min) \wedge (O[i].sta = tentative)$

Observation 15. *CAS^{17} keeps the min field unchanged and changes the sta field from $tentative$ to $steady$.*
$\forall CAS_i \in H_O^k.\ CAS_i\ is\ CAS^{17} \Rightarrow (O[i].min = O[i-1].min) \wedge (O[i].sta = steady) \wedge (O[i-1].sta = tentative)$

Observation 16. *CAS^{33} decreases the min field and changes the sta field from $steady$ to $tentative$.*
$\forall CAS_i \in H_O^k.\ CAS_i\ is\ CAS^{33} \Rightarrow (O[i].min < O[i-1].min) \wedge (O[i].sta = tentative) \wedge (O[i-1].sta = steady)$

Observation 17. *CAS^{35} never decreases the min field and always keeps the sta field $steady$.*
$\forall CAS_i \in H_O^k.\ CAS_i\ is\ CAS^{35} \Rightarrow (O[i].min \geq O[i-1].min) \wedge (O[i].sta = O[i-1].sta = steady)$

Observation 18. *Only CAS^{35} can increase the min field of a node.*

Observation 19. *Only CAS^{11} and CAS^{33} can decrease the min field of a node.*

Observation 20. *Only CAS^{17} can change a node from $tentative$ to $steady$.*

Observation 21. *If the min field of a node is decreased, the node becomes $tentative$.*

Observation 22. *If the min field of a node is increased, the node was and remains $steady$.*

Lemma 23. $O[i].min < \top \wedge O[i].sta = steady \Rightarrow \exists CAS^{17} \in H_O^i$.

Proof. By contradiction. Assume none of the $CAS_j \in H_O^i$ is CAS^{17}, then none of the $CAS_j \in H_O^i$ is either CAS^{11} or CAS^{33}, since they both change $O.sta$ to $tentative$ (Observations 14 and 16), and other than CAS^{17} there is no CAS that can change $O.sta$ from $tentative$ to $steady$ (Observation 20). So H_O^k can only contain CAS^5 and CAS^{35}. According to Observations 13 and 17, it follows that $O.min$ monotonically increases from $O[0]$ to $O[i]$. On the other hand, $O[0].min = \top$, which contradicts with $O[i].min < \top$. □

Lemma 24. *If $O[i].sta = steady$ and $CAS_j \in H_O^i$ is the most recent CAS^{17} (there is no CAS^{17} between CAS_j and CAS_i), $O.min$ monotonically increases from $O[j]$ to $O[i]$.*

Proof. Since CAS_j is CAS^{17}, $O[j].sta$ is $steady$ (Observation 15). We know for any CAS_m between CAS_j and CAS_i, CAS_m is not CAS^{17}. So CAS_m cannot be CAS^{11} or CAS^{33}, otherwise $O.sta$ remains $tentative$ after CAS_m (Observations 14 and 16), which contradicts with $O[i].sta = steady$. Therefore, CAS_m can only be CAS^5 or CAS^{35}. Thus, by Observations 13 and 17, $O.min$ monotonically increases from $O[j]$ to $O[i]$. □

5.4 Behavior of Arrive Operations

The behavior of an `Arrive()` is characterized by its preconditions and postconditions. For convenience, we first present some observations about both `Arrive()` and `Depart()`, which aid in establishing invariants governing an `Arrive()`.

Observation 25. *`Arrive()` operations never increase the min field of a node. The min field of a node can only be increased by a CAS^{35}, which is only called by `Depart()`.*

Observation 26. *`Depart()` operations only modify $steady$ nodes.*

Lemma 27. *The following precondition holds before invoking $A^{\mathbf{t}}_{k+1}(n)$ on line 15: $\forall O_i \in \mathbf{T}^{\mathbf{t}}_k.\ O_i.min \leq n$*

Proof. By induction on the length of trace k.

Basic Step. $k = 0$. Before invoking $A^{\mathbf{t}}_1(n)$, $\mathbf{T}^{\mathbf{t}}_0$ is empty, so the precondition holds trivially.

Inductive Step. $k \geq 1$. Given that $\forall O_i \in \mathbf{T}^{\mathbf{t}}_{k-1}.\ O_i.min \leq n$, we show $O_k \leq n$ holds when line 15 (of $A^{\mathbf{t}}_k(n)$) is reached. There are two paths by which line 15 can be reached: either $Read^8$ returns $O_k.sta = tentative$ and $O_k.min \leq n$, or CAS^{11} sets $O_k.sta = tentative$ and $O_k.min \leq n$. In both cases, we know that O_k reached a state where $O_k.sta = tentative$ and $O_k.min \leq n$.

We argue that after the above state is reached, no concurrent operation can update any $O_i \in \mathbf{T}^{\mathbf{t}}_k$ so that $O_i.min > n$. By the inductive hypothesis, the precondition holds for $A^{\mathbf{t}}_{k-1}(n)$, so we only need to show no concurrent operation can set $O_k.min > n$.

Since all code paths that grow the min field go through CAS^{35} (within `Revisit()` operations), we show this CAS always fails in case it is changing $O_k.min$ to some value greater than n. Suppose the CAS^{35} of some `Revisit()` tries to change $O_k.min$ to $n' > n$, it follows that the $Read^{29}$ of the same `Revisit()` returns $O_{k-1}.min \geq n' > n$, so the $Read^{29}$ must happen before the property was established for $A^{\mathbf{t}}_k(n)$, and so does the $Read^{24}$ (which returns $O_k.sta = steady$) in the same `Revisit()` operation.

As we showed above, when line 15 is reached, O_k must have reached a state where $O_k.sta = tentative$. Only a CAS^{35} can increase O_k, and we have shown that such a CAS^{35} must fail due to a concurrent `Arrive()` changing $O_k.sta$ between the $Read^{24}$ and the CAS^{35}. Thus once established, the precondition $O_k.min \leq n$ holds. □

Lemma 28. *The following postcondition holds when $A^{\mathbf{t}}_{k+1}(n)$ returns: $\forall O_i \in \overline{\mathbf{T}^{\mathbf{t}}_k}.\ O_i.min \leq n$*

Proof. By induction on the number C of `Arrive()` invocations that have returned, in the scope of the Mindicator tree.

Basic Step. $C = 0$. Initially, every node in the Mindicator tree has $min = \top$, and the postcondition trivially holds for every complementary trace.

Inductive Step. $C \geq 1$. $A^{\mathbf{t}}_{k+1}(n)$, the $(C+1)$th `Arrive()` invocation in the scope of the Mindicator tree, can return from the following paths. We argue that the postcondition holds for each case:

1. the CAS^5 succeeds
2. the test at line 14 fails
3. the test at line 16 fails
4. the CAS^{17} fails
5. the CAS^{17} succeeds

In case 1, the success of CAS^5 indicates the last $Read^2$ returned $O_{k+1}.min = m \leq n$ and $O_{k+1}.sta = steady$. Since $m \leq n < \top$, by Lemma 23 there exists a CAS^{17} in the CAS history of O_{k+1}. When the most recent CAS^{17} is performed (by some `Arrive()` invocation on O_{k+1} that has already returned), it must set the value of O_{k+1} to some $m' \leq m$, and by the inductive hypothesis we know $\forall O_i \in \overline{\mathbf{T}^{\mathbf{t}}_k}.\ O_i.min \leq m$.

After the CAS^{17}, by Lemma 24 $O_{k+1}.min$ monotonically increases from m' to m. During the time from the most recent CAS^{17} to the $Read^2$ of $A^{\mathbf{t}}_{k+1}(n)$, $\forall O_{i'} \in \overline{\mathbf{T}^{\mathbf{t}}_k}.\ O_{i'}.min \leq m$. By contradiction, assume $\exists O_{i'} \in \overline{\mathbf{T}^{\mathbf{t}}_k}.\ O_{i'}.min > m$. Then there is a `Revisit()` operation that grows $O_{i'}.min > m$ after the CAS^{17}. For this `Revisit()` operation, its $Read^{29}$ on some $O^{t?}_{i'-1}$ (a child of $O_{i'}$ not necessarily on $\mathbf{F^t}$) must return $O_{i'-1}.min > m$. It follows that $O_{i'-1}.min$ is changed greater than m after the CAS^{17}. Since $O_{i'}$ is an ancestor of O_{k+1}, the deduction leads to O_{k+1} reaching a state where $O_{k+1}.min > m$ after the CAS^{17}, which contradicts with the previous conclusion that $O_{k+1}.min$ monotonically increases from m' to m.

Now we know at the point of the $Read^2$, $\forall O_{k+i} \in \overline{\mathbf{T}^{\mathbf{t}}_k}.\ O_{k+i}.min \leq m$. To complete this case, we show that no `Revisit()` operation can grow $O_{k+1}.min > n$ after this $Read^2$. Recall that the $Read^2$ is followed by a successful CAS^5. Since the CAS^5 increments the version number of O_{k+1}, any `Revisit()` whose CAS^{35} increases $O_{k+1}.min$ must perform its $Read^{24}$ after the CAS^5. Prior to the CAS^5, the precondition on $A^{\mathbf{t}}_{k+1}(n)$ from Lemma 27 was established, so the `Revisit()`'s $Read^{29}$ of O_k must return $O_k.min \leq n$. Hence the subsequent CAS^{35} of this `Revisit()` operation cannot grow $O_{k+1}.min > n$, and the postcondition holds.

In case 2, the $Read^8$ must indicate $O_{k+1}.min \leq n$ and $O_{k+1}.sta = steady$. Hence the postcondition holds at the point of the $Read^8$. In contrast with case 1, an extra CAS is not needed for this case. In order to reach the $Read^8$, the last $Read^2$ must return $O_{k+1}.min > n$ or $O_{k+1}.sta = tentative$, so O_{k+1} is changed *between* the $Read^2$ and the $Read^8$. Since the $Read^2$ occurs after the precondition is established for $A^{\mathbf{t}}_{k+1}(n)$ (Lemma 27), any `Revisit()` operation that performs a CAS^{35} after the $Read^8$, its $Read^{24}$ must return $O_k.min \leq n$, and thus the subsequent CAS^{35} cannot grow $O_{k+1}.min > n$.

In cases 3–5, $A^{\mathbf{t}}_{k+1}(n)$ invokes $A^{\mathbf{t}}_{k+2}(n)$ at line 15. By the inductive hypothesis, when $A^{\mathbf{t}}_{k+2}(n)$ returns, the postcondition holds for $\overline{\mathbf{T}^{\mathbf{t}}_{k+1}}$. (If O_{k+1} is the root node, invoking $A^{\mathbf{t}}_{k+2}(n)$ has no effect, and the postcondition holds trivially since $\overline{\mathbf{T}^{\mathbf{t}}_{k+1}}$ is empty.) Now we argue that the postcondition holds for $\overline{\mathbf{T}^{\mathbf{t}}_k}$ when $A^{\mathbf{t}}_{k+1}(n)$ returns in these cases.

Note that the precondition from Lemma 27 on $A^{\mathbf{t}}_{k+2}(n)$ still applies when $A^{\mathbf{t}}_{k+2}(n)$ returns, because

- $A^{\mathbf{t}}_{k+2}(n)$ never modifies nodes in $\mathbf{T}^{\mathbf{t}}_{k+1}$.
- Concurrent `Arrive`s do not violate the precondition, since `Arrive`s never increase the min field.
- Concurrent `Revisit`s cannot violate the precondition until some point in time after the response of $A^{\mathbf{t}}_1(n)$. After the precondition is established, $\forall O_i, O_{i+1} \in \mathbf{T}^{\mathbf{t}}_{k+1}$, a `Revisit()` can set $O_{i+1}.min > n$ only if its $Read^{29}$ returns $O_i.min > n$. So concurrent `Revisit`s are safe for the precondition as long as $O_1.min = n$ holds, and by Lemma 27, $O_1.min = n$ holds at least until the response of $A^{\mathbf{t}}_1(n)$.

Therefore, we have $\forall O_i \in \mathbf{T}^{\mathbf{t}}_{k+1}.\ O_i.min \leq n$ when $A^{\mathbf{t}}_{k+2}(n)$ returns.

In cases 3 and 4, $A^{\mathbf{t}}_{k+1}(n)$ performs no further changes to O_{k+1} after $A^{\mathbf{t}}_{k+2}(n)$ returns, and the postcondition holds for $\overline{\mathbf{T}^{\mathbf{t}}_k}$.

In case 5, the CAS^{17} sets $O_{k+1}.min = n$. So the postcondition holds for $\overline{\mathbf{T}^{\mathbf{t}}_k}$ at the point of the CAS^{17}. For any `Revisit()` operation which performs a CAS^{35} thereafter, the CAS^{17} serves as the lower bound on when the $Read^{24}$ of such a `Revisit()` operation can happen. The subsequent $Read^{29}$ must return at least one child of O_{k+1} with $min \leq n$ (by Lemma 27, $O_k.min \leq n$), and thus the `Revisit()` cannot change $O_{k+1}.min > n$. □

Corollary 29. *The following postcondition holds when* $A^{\mathbf{t}}_{k+1}(n)$ *returns:* $\forall O_i \in \overline{\mathbf{T}^{\mathbf{t}}_k}.\ O_i.sta = tentative \Rightarrow O_i.min < n$

Proof. First, we show at the point that $A^{\mathbf{t}}_{k+1}(n)$ returns, $O_{k+1}.sta = tentative \Rightarrow X.min < n$. There are two possible paths that $A^{\mathbf{t}}_{k+1}(n)$ returns with $O_{k+1}.sta = tentative$:

1. the test at line 16 fails
2. CAS^{17} fails

In case 1, $O_{k+1}.min \neq n$ at line 16. Since Lemma 28 ensures $O_{k+1}.min \leq n$, it follows that $O_{k+1}.min < n$.

In case 2, we know that a $Read^8$ returns $O_{k+1}.min = n$ and $O_{k+1} = tentative$ in the iteration of the loop at line 7 that exits. Any number of CAS operations may be performed on O_{k+1} after this $Read^8$, but the first of these CAS operations (which we call FC) must be a CAS^{11} or CAS^{33}, both of which decrease $O_{k+1}.min$ to some value $< n$. Any subsequent CAS is either (a) one of $\{CAS^5, CAS^{11}, CAS^{17}, CAS^{33}\}$, which never increase the value of $O_{k+1}.min$; or (b) a CAS^{35} which, due to increments of the version number, obeys an ordering where $FC \rightarrow_{\mathcal{R}_1} Read^{24} \rightarrow_{\mathcal{R}_1} CAS^{35}$. This condition ensures that the CAS^{35} will not increase the value of $O_{k+1}.min$ any higher than n, and if it does so, it will make O_{k+1} $steady$. Thus the property is established when the operation returns.

Now we show that after $A^{\mathbf{t}}_{k+1}(n)$ returns, $O_{k+1}.min$ can never grow back to n with $O_{k+1}.sta = tentative$ at the same time. By Lemma 28, $O_{k+1}.min$ cannot exceed n. On the other hand, only CAS^{35} can grow the min field and CAS^{35} always sets the sta field to $steady$. So if CAS^{35} changes $O_{k+1}.min = n$ from $n' < n$, it also sets $O_{k+1}.sta = steady$. Any subsequent CAS that makes the node $tentative$ will also decrease the min field, and thus it will never hold that O_{k+1} has $min = n$ and $sta = steady$. □

Invariant 30. $\forall O_i \in \mathbf{F^t}.\ O_i.sta = steady \Rightarrow \forall O_a \in \overline{\mathbf{T}^{\mathbf{t}}_i}.\ O_a.min \leq O_i.min$

Proof. By induction on the number C of CAS instructions that have been performed, in the scope of Mindicator tree.

Basic Step. $C = 0$. Before any CAS instruction is performed, the invariant holds, since every node in the Mindicator tree is initialized with $min = \top$.

Inductive Step. $C > 0$. Given that the invariant holds after the first C CAS instructions are performed, we show the $(C+1)$th CAS instruction does not break the invariant.

First, a CAS^5 can never break the invariant since it modifies only the version number of a node. Second, a CAS^{11} or CAS^{33} is always safe, because they change nodes to $tentative$ state, while the invariant only constrains nodes in $steady$ state. If the $(C+1)$th CAS instruction is a CAS^{17} or CAS^{35} performed on node X, we have shown in the proof of Lemma 28 that for any ancestor O_a of O_i, $O_a.min \leq O_i.min$, and for any $steady$ descendant O_d of O_i, a CAS^{35} cannot grow $O_i.min > O_d.min$. □

Invariant 31. $\forall O_i \in \mathbf{F^t}.\ O_i.sta = steady \Rightarrow (\forall O_a \in \overline{\mathbf{T}^{\mathbf{t}}_i}.\ O_a.sta = tentative \Rightarrow O_a.min < O_i.min)$

Proof. By induction on the number C of CAS instructions that have been performed, in the scope of Mindicator tree.

Basic Step. $C = 0$. Before any CAS instruction is performed, the invariant holds trivially, since every node in the Mindicator tree is initialized with $sta = steady$ while the invariant only constrains $tentative$ nodes.

Inductive Step. $C > 0$. Given that the invariant holds after the first C CAS instructions are performed, we show the $(C+1)$th CAS instruction does not break the invariant.

According to Invariant 30, if node O is $steady$, for any ancestor O_a of O_i, $O_a.min \leq O_i.min$ holds. On the other hand, CAS^{11} and CAS^{33}, the only two CAS instructions that change nodes to $tentative$ state, *strictly* decrease the min field, which preserves the invariant. □

5.5 Behavior of Depart Operations

The following lemmas focus on the discussion of liveness, which is the primary responsibility of `Depart()` operations. For clarity, we first state some properties related to tokens, which are described informally in Section 5.1.

Observation 32 (Uniqueness). *For distinct operations $A^{\mathbf{t}}(n|\alpha)$ and $A^{\mathbf{t}'}(m|\beta)$, $\alpha \neq \beta$.*

Observation 33 (Immutability). *Once a token is generated, it is never changed.*

Observation 34. *$\langle A^{\mathbf{t}}(n|\alpha), D^{\mathbf{t}}(n|\alpha)\rangle$ take the same token $(n|\alpha)$ as parameter.*

Observation 35. *An identifier α can only appear as the id field of nodes on the full path of $A^{\mathbf{t}}(n|\alpha)$.*

Observation 36. *If $O.id = \alpha$, then $\exists A^{\mathbf{t}}(n|\alpha)$ which has performed at least one CAS instruction (i.e., the CAS^{11} on the leaf node of thread $\mathbf{t}$).*

Observation 37. *If $O.id = \alpha \wedge \exists A^{\mathbf{t}}(n|\alpha)$, then $O.min = n$.*

Observation 38. *If $O.min \neq n \wedge \exists A^{\mathbf{t}}(n|\alpha)$, then $O.id \neq \alpha$.*

Lemma 39. *Between the invocation and response points of $D^{\mathbf{t}}(n|\alpha)$: $\forall O_i \in \mathbf{F}^{\mathbf{t}}.\ O_i.sta = tentative \Rightarrow O_i.id \neq \alpha$*

Proof. By contradiction, assume at some point during $D^{\mathbf{t}}(n|\alpha)$, $\exists O_k \in \mathbf{F}^{\mathbf{t}}$ that $O_k.sta = tentative$, $O_k.id = \alpha$, and $O_k.min = n$. The only possibilities are for O_k to reach this state via a CAS^{11} or CAS^{33}.

Suppose O_k reaches the state via a CAS^{11}. The CAS must be performed by $A^{\mathbf{t}}(n|\alpha)$, the pairing operation of $D^{\mathbf{t}}(n|\alpha)$. It leads to a contradiction since the response of $A^{\mathbf{t}}(n|\alpha)$ ensures that $O_k.min < n$ if $O_k.sta = tentative$ (Corollary 29), which implies that $O_k.id \neq \alpha$.

So O_k must reach the state via a CAS^{33} performed by some `Revisit()` concurrent with $D^{\mathbf{t}}(n|\alpha)$. It follows that O_k *was* in a state where $O_k.min = n' > n$ (Observation 16). In the `Revisit()` operation that performs the CAS^{33}, the $Read^{29}$ of O_{k-1} returns $O_{k-1}.id = \alpha$ (with $O_{k-1}.min = n$). First, the $Read^{29}$ cannot return $O_{k-1}.sta = steady$: Invariant 30 establishes that in this case, $O_k.min = n'' \leq O_{k-1}.min$ at the point of the $Read^{29}$. Were this the case, $O_k.min$ would have to change from n' to n'' between the $Read^{24}$ and $Read^{29}$, and the subsequent the CAS^{33} would fail.

Thus, we know the $Read^{29}$ returns $O_{k-1}.sta = tentative$, and this, in turn, requires O_{k-1} to reach this state via a CAS^{33} concurrent with $D^{\mathbf{t}}(n|\alpha)$. Since k is finite, the deduction propagates to O_1, and requires that $O_1.sta = tentative$ and $O_1.id = \alpha$. This is impossible because $O_1.sta = steady$ after the response of $A^{\mathbf{t}}(n|\alpha)$, and holds during $D^{\mathbf{t}}(n|\alpha)$. □

Lemma 40. *The following precondition holds when invoking $D^{\mathbf{t}}_{k+1}(n|\alpha)$: $\forall O_i \in \mathbf{T}^{\mathbf{t}}_k.\ O_i.id \neq \alpha$*

Proof. By induction on the length of the trace k.

Basic Step. $k = 0$. Before invoking $D^{\mathbf{t}}_1(n)$, the trace is empty, so the precondition holds trivially.

Inductive Step. $k \geq 1$. Given that $\forall O_i \in \mathbf{T}^{\mathbf{t}}_{k-1}.\ O_i.id \neq \alpha$, we show $O_k.id \neq \alpha$ holds when line 22 is reached. More precisely, we show it holds when the `Revisit()` operation returns. A `Revisit()` operation can return from the following paths:

1. returns at line 27
2. CAS^{33} succeeds
3. CAS^{35} succeeds

In case 1, the test at line 26 indicates that $O_k.sta = tentative$. By Lemma 39, we know $O_k.id \neq \alpha$ at point of the $Read^{24}$. Since α is unique, no `Arrive()` can later set $O_i.id$ to α. Thus we must show that after the $Read^{24}$, no `Revisit()` (which we shall refer to as V) restores $O_i.id$ to α. Since `Revisit()` operations cannot modify $tentative$ nodes, the $Read^{24}$ of V must come after $O_k.sta$ becomes $steady$ (by Observation 15 this must be achieved via a CAS^{17}). At that point, $\mathcal{R}_1$ ensures that the precondition for $D^{\mathbf{t}}_k(n|\alpha)$ was established before the $Read^{24}$ of V. By the inductive hypothesis, the $Read^{29}$ of this V on O_{k-1} must return $O_{k-1}.id \neq \alpha$, so the subsequent CAS instruction cannot set $O_k.id = \alpha$

In cases 2–3, the success of CAS^{33} or CAS^{35} sets $O_k.id \neq \alpha$ and meanwhile invalidates concurrent `Revisit()` operations that try to set $O_k.id = \alpha$. Once the failed `Revisit()` operation retries, by the inductive hypothesis, $O_{k-1}.id \neq \alpha$, so its $Read^{29}$ must return $O_{k-1}.id \neq \alpha$ and the subsequent CAS^{33} or CAS^{35} instruction cannot set $O_k.id = \alpha$. □

Lemma 41. *The following postcondition holds when $D^{\mathbf{t}}_{k+1}(n|\alpha)$ returns: $\forall O_i \in \overline{\mathbf{T}^{\mathbf{t}}_k}.\ O_i.id \neq \alpha$*

Proof. Since `Depart()` operations are tail-recursive, a `Depart()` operation returns when the top level invocation $D^{\mathbf{t}}_{k+1}(n|\alpha)$ returns. In other words, we show that when $D^{\mathbf{t}}_{k+1}(n|\alpha)$ returns, $\forall O \in \mathbf{F}^{\mathbf{t}}.\ O.id \neq \alpha$. There are two cases when $D^{\mathbf{t}}_{k+1}(n|\alpha)$ returns:

1. returns at line 22 on the root node ($X = R$)
2. returns at line 21

In case 1, $D^{\mathbf{t}}_{k+1}(n|\alpha)$ returns on the root node, so the trace $\mathbf{T}^{\mathbf{t}}_k$ is the full path $\mathbf{F}^{\mathbf{t}}$. By Lemma 40, $\forall O_i \in \mathbf{T}^{\mathbf{t}}_{k+1}.\ O_i.id \neq \alpha$.

In case 2, $D^{\mathbf{t}}_{k+1}(n|\alpha)$ returns at line 21. The $Read^{19}$ returns $O_{k+1}.min = m < n$ and $O_{k+1}.sta = steady$. Invariant 30 guarantees at the point of $Read^{19}$, $\forall O_i \in \overline{\mathbf{T}^{\mathbf{t}}_k}.\ O_i.min \leq m < n$. Thus, $\forall O_i \in \overline{\mathbf{T}^{\mathbf{t}}_k}.\ O_i.id \neq \alpha$. Lemma 40 similarly establishes that $\forall O_i \in \mathbf{T}^{\mathbf{t}}_k.\ O_i.id \neq \alpha$ holds after the `Revisit()` operation of $D^{\mathbf{t}}_k(n|\alpha)$ returns. Given the result of $Read^{19}$, it would appear that the postcondition already holds. The lone concern is if there is an in-flight `Revisit()` operating on some $O_j \in \overline{\mathbf{T}^{\mathbf{t}}_k}$ that could restore the value α.

By contradiction, assume $\exists O_j \in \overline{\mathbf{T}^{\mathbf{t}}_k}.\ O_j.id = \alpha$, which is set by some `Revisit()` operation (again, we use V to refer to this operation) after the $Read^{19}$. Then the $Read^{29}$ of O_{j-1} by V must return $O_{j-1}.id = \alpha$. So $O_{j-1}.id = \alpha$ is set by some `Revisit()` operation after the $Read^{19}$.

Repeating this reasoning for $j-1, j-2, \ldots, j-z$, we ultimately reach O_{k+1}, at which point a `Revisit()` (V') sets $O_{k+1}.id = \alpha$ after the $Read^{19}$ that returned $O_k.min = m$. This leads to a contradiction with Lemma 40, where it was shown that the precondition of $D^{\mathbf{t}}_{k+2}(n|\alpha)$ holds immediately after the `Revisit()` of $D^{\mathbf{t}}_{k+1}(n|\alpha)$ returns. □

5.6 Behavior of Query Operations

We now prove lemmas about the inter-operation behavior of the algorithm, when `Query()` operations are involved. We begin by proving simple properties for operations that are not concurrent. Beginning with Lemma 46, we begin to place constraints on when an operation's effect can be observed, even when it is concurrent with other operations. In particular, we show that if two `Query`s are both concurrent with some other operation, then the values they return can establish both the existence of some other concurrent operation, and also an order among a set of operations.

Lemma 42. *For every $Q^{\mathbf{t}}(n)$, for every $\langle A^{\mathbf{t}'}(m), D^{\mathbf{t}'}(m)\rangle$ such that $A^{\mathbf{t}'}(m) \rightarrow_{\mathcal{R}_2} Q^{\mathbf{t}}(n)$ and $Q^{\mathbf{t}}(n) \rightarrow_{\mathcal{R}_2} D^{\mathbf{t}'}(m)$, $n \leq m$.*

Proof. Since the root node R is in $\overline{\mathbf{T}_0^{\mathbf{t}'}}$, Lemma 28 ensures that $R.min \leq m$ holds until the invocation of $D^{\mathbf{t}'}(m)$, So for every $Q^{\mathbf{t}}(n)$ between $A^{\mathbf{t}'}(m)$ and $D^{\mathbf{t}'}(m)$, we have $n \leq m$. □

Lemma 43. *For every $Q^{\mathbf{t}}(n|\alpha)$, for every $\langle A^{\mathbf{t}'}(m|\beta), D^{\mathbf{t}'}(m|\beta)\rangle$ such that $Q^{\mathbf{t}}(n|\alpha) \rightarrow_{\mathcal{R}_2} A^{\mathbf{t}'}(m|\beta)$ or $D^{\mathbf{t}'}(m|\beta) \rightarrow_{\mathcal{R}_2} Q^{\mathbf{t}}(n|\alpha)$, $\alpha \neq \beta$.*

Proof. In the former case, where $Q^{\mathbf{t}}(n|\alpha)$ precedes $A^{\mathbf{t}'}(m|\beta)$, β has not been written to any node in the Mindicator tree (it will not be written anywhere until $A_1^{\mathbf{t}'}(m|\beta)$). Hence $\alpha \neq \beta$. In the latter case, where $D^{\mathbf{t}'}(m|\beta)$ precedes $Q^{\mathbf{t}'}(n|\alpha)$, the postcondition of $D_1^{\mathbf{t}'}(m|\beta)$ ensures that for the root node R, which is in $\overline{\mathbf{T}_0^{\mathbf{t}'}}$, $R.id \neq \beta$ (Lemma 41). Hence $\alpha \neq \beta$. □

Corollary 44. *For every $Q^{\mathbf{t}}(n)$, for every $\langle A^{\mathbf{t}'}(m), D^{\mathbf{t}'}(m)\rangle$ such that $m < n$, exactly one of the following claims is true:*

- *$Q^{\mathbf{t}}(n) \rightarrow_{\mathcal{R}_2} A^{\mathbf{t}'}(m)$ or $D^{\mathbf{t}'}(m) \rightarrow_{\mathcal{R}_2} Q^{\mathbf{t}}(n)$.*
- *$Q^{\mathbf{t}}(n)$ and $A^{\mathbf{t}'}(m)$ are concurrent in $\mathcal{R}_2$.*
- *$Q^{\mathbf{t}}(n)$ and $D^{\mathbf{t}'}(m)$ are concurrent in $\mathcal{R}_2$.*

Proof. Contrapositive of Lemma 42. □

Corollary 45. *For every $Q^{\mathbf{t}}(n|\alpha)$, there exists a unique $\langle A^{\mathbf{t}'}(n|\alpha), D^{\mathbf{t}'}(n|\alpha)\rangle$ and exactly one of the following claims is true:*

- *$A^{\mathbf{t}'}(n|\alpha) \rightarrow_{\mathcal{R}_2} Q^{\mathbf{t}}(n|\alpha)$ and $Q^{\mathbf{t}}(n|\alpha) \rightarrow_{\mathcal{R}_2} D^{\mathbf{t}'}(n|\alpha)$.*
- *$Q^{\mathbf{t}}(n|\alpha)$ and $A^{\mathbf{t}'}(n|\alpha)$ are concurrent in $\mathcal{R}_2$.*
- *$Q^{\mathbf{t}}(n|\alpha)$ and $D^{\mathbf{t}'}(n|\alpha)$ are concurrent in $\mathcal{R}_2$.*

Proof. First we show that for every `Query()` operation $Q^{\mathbf{t}}(n|\alpha)$, there exists a unique pair $\langle A^{\mathbf{t}'}(n|\alpha), D^{\mathbf{t}'}(n|\alpha)\rangle$. Since we require the existence of $A^{-\mathbf{1}}(\top|\phi)$ and $D^{-\mathbf{1}}(\top|\phi)$ as the first and last operations in the closed, interleaved thread history, the above claim holds trivially when $\alpha = \phi$ ($n = \top$). If $\alpha \neq \phi$, because identifiers are unique, α must be generated by some unique `Arrive()` operation $A^{\mathbf{t}'}(n|\alpha)$. Given the existence of the pair $\langle A^{\mathbf{t}'}(n|\alpha), D^{\mathbf{t}'}(n|\alpha)\rangle$, the corollary is proved by the applying the contrapositive of Lemma 43. □

Lemma 46. *For two ordered `Query()` operations $Q^{\mathbf{t}'}(n|\alpha) \rightarrow_{\mathcal{R}_2} Q^{\mathbf{t}''}(m|\beta)$, both of which are concurrent with $A^{\mathbf{t}}(n|\alpha)$ in $\mathcal{R}_2$, $m \leq n$.*

Proof. The claim holds trivially if $\alpha = \beta$, since in that case $m = n$. If $A^{\mathbf{t}}(n|\alpha)$ has "turned" (some invocation $A_k^{\mathbf{t}}(n|\alpha)$ has returned) before $Q^{\mathbf{t}'}(n|\alpha)$, Lemma 28 guarantees for the root node R, which is in the complementary trace of $A_k^{\mathbf{t}}(n|\alpha)$, $R.min \leq n$. Hence $m \leq n$.

Otherwise, we say $A^{\mathbf{t}}(n|\alpha)$ is still "climbing" (i.e., no invocation has returned yet) at the time when $Q^{\mathbf{t}'}(n|\alpha)$ occurs. Suppose $A_{k+1}^{\mathbf{t}}(n|\alpha)$ is current invocation when the `Query()` occurs. We argue that after $Q^{\mathbf{t}'}(n|\alpha)$ and before the response of $A^{\mathbf{t}}(n|\alpha)$, no node in $\mathbf{F}^{\mathbf{t}}$ can be changed with $min > n$. By Lemma 27, $\forall O_i \in \mathbf{T}_k^{\mathbf{t}}.\ O_i.min \leq n$. Thus we only need to prove $\forall O_i \in \overline{\mathbf{T}_k^{\mathbf{t}}}.\ O_i.min \leq n$.

Intuitively, since $O_k \neq R$, there must be a concurrent `Depart()` that is using CAS^{33} or CAS^{35} to "pull" the token $(n|\alpha)$ up to the root node, where it can be observed by the `Query()` ($Q^{\mathbf{t}'}(n|\alpha)$).

At the point of $Q^{\mathbf{t}'}(n|\alpha)$, the invocation $A_1^{\mathbf{t}}(n|\alpha)$ must have performed the CAS^{11} on the leaf node of $\mathbf{t}$. Otherwise no node in the Mindicator tree can have $id = \alpha$.

Between the CAS^{11} of $A_1^{\mathbf{t}}(n|\alpha)$ and $Q^{\mathbf{t}'}(n|\alpha)$, every $O_i \in \overline{\mathbf{T}_k^{\mathbf{t}}}$ must reach a state where $O_i.id = \alpha$. By contradiction, assume some $O_{i'}.id$ never equals α. Then $O_{i'+1}.id$ can never be α, since $O_{i'+1}$ can be

updated only by threads other than $\mathbf{t}$, via either `Revisit()` operations (which can set $O_{i'+1}.id = \alpha$ only if $O_{i'}.id = \alpha$) or `Arrive()` operations (which can never set $O_{i'+1}.id = \alpha$ if the performer is not thread $\mathbf{t}$). It follows that $R.id \neq \alpha$, which contradicts with $Q^{\mathbf{t}'}(n|\alpha)$. Moreover, the above argument implies that $\forall O_i \in \overline{\mathbf{T}^{\mathbf{t}}_k}$, O_i reaches the state $O_i.id = \alpha$ *before* O_{i+1} reaches the state $O_{i+1}.id = \alpha$, since the `Revisit()` operation that sets $O_{i+1}.id = \alpha$ must first read $O_i.id = \alpha$.

Now we argue that $\forall O_i \in \overline{\mathbf{T}^{\mathbf{t}}_k}$, once $O_i.id = \alpha$ (with $O_i.min = n$), $O_i.min \leq n$ holds until the response of $A^{\mathbf{t}}(n|\alpha)$. By contradiction, assume $\exists O_{i'} \in \overline{\mathbf{T}^{\mathbf{t}}_k}.\ O_{i'}.min > n$. Since `Arrive()` never increases the min field, there must exist a `Revisit()` operation that grows $O_{i'}.min > n$, and its $Read^{24}$ must return $O_{i'-1}.min > n$. It follows that some `Revisit()` operation grows $O_{i'-1}.min > n$ after $O_{i'-1}.id = \alpha$. As in the proof of Lemma 41, such a situation requires us to repeat the reasoning on $\{O_{i'-1}, O_{i'-2}, \ldots\}$, and we ultimately arrive at the situation that there exists some `Revisit()` operation that grows $O_{k+1}.min > n$ after $O_{k+1}.id = \alpha$. Here we have two cases:

1. $O_{k+1}.id = \alpha$ was set by the CAS^{11} of $A^{\mathbf{t}}_{k+1}(n|\alpha)$
2. $O_{k+1}.id = \alpha$ was set by a CAS^{33} or CAS^{35} of some `Revisit()` operation

The former case instantly leads to a contradiction with Lemma 27. In the latter case, it follows that $O_k.id$ *was* α and $O_k.min$ is changed greater than n thereafter, which again splits to the above two cases. Therefore, the deduction either leads to a contradiction at some $O_{i''} \in \mathbf{T}^{\mathbf{t}}_k$ ($i'' < k$) through the former case, or descends to that $O_1.min$ is changed greater than n after $A^{\mathbf{t}}_1(n|\alpha)$ performs the CAS^{11} on O_1 (the leaf node), which contradicts with Lemma 27. □

Lemma 47. *For two ordered* `Query()` *operations* $Q^{\mathbf{t}'}(n|\alpha) \rightarrow_{\mathcal{R}_2} Q^{\mathbf{t}''}(m|\beta)$*, both of which are concurrent with* $D^{\mathbf{t}}(m|\beta)$ *in* $\mathcal{R}_2$*,* $n \leq m$*.*

Proof. By contradiction, assume $n > m$. Given this assumption, at the point of $Q^{\mathbf{t}'}(n|\alpha)$, we first show that in the full path $\mathbf{F}^{\mathbf{t}}$, $\forall O_i \in \mathbf{F}^{\mathbf{t}}.\ O_i.sta = steady \Rightarrow O_i.min > m$. Since R is an ancestor of every node O in the Mindicator tree, Invariant 30 requires $\forall O_i \in \mathbf{F}^{\mathbf{t}}.\ O_i.sta = steady \Rightarrow R.min \leq O_i.min$. Since the query returns $R.min = n$, we know that when the query occurs, $n \leq O_i.min$. Since we assume that $n > m$, it follows that $\forall O_i \in \mathbf{F}^{\mathbf{t}}.\ m < O_i.min$. Since β is a unique value generated to accompany value m, $\forall O_i \in \mathbf{F}^{\mathbf{t}}.\ O_i.id \neq \beta$.

Recall that Lemma 39 ensures that between the invocation and response of $D^{\mathbf{t}}(m|\beta)$, $\forall O_i \in \mathbf{F}^{\mathbf{t}}.\ O_i.sta = tentative \Rightarrow O_i.id \neq \beta$. Combining the Lemma with the above consequence of our assumption that $n > m$, at the point of $Q^{\mathbf{t}'}(n|\alpha)$, we know that $\forall O_i \in \mathbf{F}^{\mathbf{t}}.\ O_i.id \neq \beta$.

Given the assumption that $n > m$, we now we show that after $Q^{\mathbf{t}'}(n|\alpha)$, β cannot appear anywhere in $\mathbf{F}^{\mathbf{t}}$ until after the response of $D^{\mathbf{t}}(m|\beta)$. This will form a contradiction since $Q^{\mathbf{t}''}(m|\beta)$ is concurrent with $D^{\mathbf{t}}(m|\beta)$.

By contradiction, assume $\exists O'_i \in \mathbf{F}^{\mathbf{t}}.\ O_i.id = \beta$. $O_i = \beta$ must be set by some `Revisit()` that occurs after after $Q^{\mathbf{t}'}(n|\alpha)$. The `Revisit()` operation cannot set $O_i.id = \beta$ via a CAS^{33}, otherwise, Lemma 39 is violated since the CAS^{33} also sets $O_i.sta = tentative$. So the `Revisit()` operation must perform a CAS^{35}, which grows $O_i.min = m$ from some value $m' \leq m$. In this `Revisit()`, the $Read^{29}$ returns $O_{i-1}.id = \beta$, so it must precede $Q^{\mathbf{t}'}(n|\alpha)$. It follows that the $Read^{24}$ (which returns $O_i.min = m' \leq m$) in this `Revisit()` also precedes $Q^{\mathbf{t}'}(n|\alpha)$. On the other hand, we have shown above that at the point of $Q^{\mathbf{t}'}(n|\alpha)$, $O_i.sta = steady \Rightarrow O_i.min = m'' > m$. Therefore, such a CAS^{35} must fail due to fact that a concurrent modification to O_i updates $O_i.ver$. That is, at the point of $Q^{\mathbf{t}'}(n|\alpha)$, either $O_i.sta$ is changed (from $steady$ to $tentative$) or $O_i.min$ is changed (from $m' < m$ to some $m'' > m$). □

Corollary 48. *For two ordered* `Query()` *operations* $Q^{\mathbf{t}'}(n|\alpha) \rightarrow_{\mathcal{R}_2} Q^{\mathbf{t}''}(m|\beta)$*, both of which are concurrent with* $A^{\mathbf{t}}(p|\gamma)$ *in* $\mathcal{R}_2$*, if* $m > p$*, then* $\alpha \neq \gamma$*.*

Proof. Contrapositive of Lemma 46. □

Corollary 49. *For two ordered* `Query()` *operations* $Q^{\mathbf{t}'}(n|\alpha) \rightarrow_{\mathcal{R}_2} Q^{\mathbf{t}''}(m|\beta)$*, both of which are concurrent with* $D^{\mathbf{t}}(p|\gamma)$ *in* $\mathcal{R}_2$*, if* $n > p$*, then* $\beta \neq \gamma$*.*

Proof. Contrapositive of Lemma 47. □

5.7 Proof of Linearizability

The construction of increasingly strong Lemmas now allows us to prove that every interleaved history of operations performed in a Mindicator corresponds to a legal sequential history. An important simplifying factor deals with what can be observed: when `Arrive()` and `Depart()` operations are concurrent, their relative order does not matter unless there is a concurrent `Query()`, by which that order can be observed. For example, in the non-trivial case of Lemma 46 (where the two `Query`s return different results), there was an anonymous `Arrive()`, concurrent with the `Arrive()`that is explicitly named in the lemma, and we showed that the presence of two `Query`s returning different values induced bounds on when the two `Arrive`s could happen. In summary:

- Corollary 44 establishes a *weak safety* condition, where in the absence of concurrent operations, a `Query()` will always return the smallest value in the Mindicator.
- Corollary 45 establishes a *weak liveness* condition, where in the absence of concurrent operations, a `Query()` will never return a value that is not represented in the Mindicator.
- Lemma 46 sets a latest bound on when an `Arrive()` can happen, by showing that a concurrent `Depart()` can cause an `Arrive()` to take effect (e.g., be visible to a `Query()`) as early as its first CAS.
- Lemma 47 sets an earliest bound on when a `Depart()` (D) can happen, by showing that a concurrent `Depart()` can make D's value visible to a `Query()` even after D has performed several CAS operations.
- Corollary 48 sets an earliest bound on when an `Arrive()` can happen.
- Corollary 49 sets a latest bound on when a `Depart()` can happen.

We now give the construction procedure of relation $\mathcal{R}_2^*$ on all $\{$`Arrive()`, `Depart()`, `Query()`$\}$ operations (shown in Figure 4). $\mathcal{R}_2^*$ is iteratively elaborated from $\mathcal{R}_2$. Initially, $\mathcal{R}_2^*$ is set equal to $\mathcal{R}_2$. In each iteration, the procedure looks for a pair of concurrent operations that matches one of the specified patterns. An ordering is assigned between such two operations and added to $\mathcal{R}_2^*$. The construction procedure terminates when no more matched pairs can be found. We show that for concurrent operations, $\mathcal{R}_2^*$ introduces an order that preserves the aforementioned bounds on when operations can take effect. Upon termination, $\mathcal{R}_2^*$ establishes strong safety and liveness conditions, in that any total order induced among the remaining unordered operations in $\mathcal{R}_2^*$ results in a history for which there exists an equivalent legal sequential history.

Lemma 50. *Corollaries 44 and 45, Lemmas 46, 47, and Corollaries 48, and 49 (all with respect to* $\mathcal{R}_2^*$*) hold as invariants for the loop in the construction procedure.*

Proof. By induction on the number of iterations k.

Basic Step. $k = 0$. $\mathcal{R}_2^*$ is initialized as $\mathcal{R}_2$, so all the invariants hold before the loop is executed.

Inductive Step. $k > 0$. Given that all the invariants hold for the first k iterations, we show they continue to hold after the $(k+1)$th iteration.

Once an ordering pair $\{op_1 \rightarrow_{\mathcal{R}_2^*} op_2\}$ is added to $\mathcal{R}_2^*$, Lemmas 46 and 47, and Corollaries 48 and 49 trivially hold. Since these lemmas apply to concurrent operations, if one of these lemmas held, removing

procedure $Construct\mathcal{R}_2^*\ From\mathcal{R}_2()$
  $\mathcal{R}_2^* \leftarrow \mathcal{R}_2$
  repeat
    if exists $Q^{\mathbf{t}}(n|\alpha)$ and $A^{\mathbf{t}'}(n|\alpha)$ that are concurrent in both $\mathcal{R}_2$ and $\mathcal{R}_2^*$
      $\mathcal{R}_2^* \leftarrow \mathcal{R}_2^* \cup \{A^{\mathbf{t}'}(n|\alpha) \rightarrow_{\mathcal{R}_2^*} Q^{\mathbf{t}}(n|\alpha)\}$
    elif exists $Q^{\mathbf{t}}(n|\alpha)$ and $D^{\mathbf{t}'}(n|\alpha)$ that are concurrent in both $\mathcal{R}_2$ and $\mathcal{R}_2^*$
      $\mathcal{R}_2^* \leftarrow \mathcal{R}_2^* \cup \{Q^{\mathbf{t}}(n|\alpha) \rightarrow_{\mathcal{R}_2^*} D^{\mathbf{t}'}(n|\alpha)\}$
    elif exists $Q^{\mathbf{t}}(n)$ and $A^{\mathbf{t}'}(m)$ that are concurrent in both $\mathcal{R}_2$ and $\mathcal{R}_2^*$, and $n > m$
      $\mathcal{R}_2^* \leftarrow \mathcal{R}_2^* \cup \{Q^{\mathbf{t}}(n) \rightarrow_{\mathcal{R}_2^*} A^{\mathbf{t}'}(m)\}$
    elif exists $Q^{\mathbf{t}}(n)$ and $D^{\mathbf{t}'}(m)$ that are concurrent in both $\mathcal{R}_2$ and $\mathcal{R}_2^*$, and $n > m$
      $\mathcal{R}_2^* \leftarrow \mathcal{R}_2^* \cup \{D^{\mathbf{t}'}(m) \rightarrow_{\mathcal{R}_2^*} Q^{\mathbf{t}}(n)\}$
  until no new pairs are added to $\mathcal{R}_2^*$

Figure 4: The Construction Procedure of Relation $\mathcal{R}_2^*$

the concurrent relationship makes it hold trivially. Thus, we need only prove that Corollaries 44 and 45 are not violated.

An ordering relation $\{op_1 \rightarrow_{\mathcal{R}_2^*} op_2\}$ added to $\mathcal{R}_2^*$ in the $(k+1)$th iteration must take one of the following forms:

1. $\{A^{\mathbf{t}'}(n|\alpha) \rightarrow_{\mathcal{R}_2^*} Q^{\mathbf{t}}(n|\alpha)\}$
2. $\{Q^{\mathbf{t}}(n|\alpha) \rightarrow_{\mathcal{R}_2^*} D^{\mathbf{t}'}(n|\alpha)\}$
3. $\{Q^{\mathbf{t}}(n|\alpha) \rightarrow_{\mathcal{R}_2^*} A^{\mathbf{t}'}(m|\beta)\}$ and $n > m$
4. $\{D^{\mathbf{t}'}(m|\beta) \rightarrow_{\mathcal{R}_2^*} Q^{\mathbf{t}}(n|\alpha)\}$ and $n > m$

In case 1, $\{A^{\mathbf{t}'}(n|\alpha) \rightarrow_{\mathcal{R}_2^*} Q^{\mathbf{t}}(n|\alpha)\}$ is added to $\mathcal{R}_2^*$ through the first branch in the loop of the construction procedure. Corollary 45 continues to hold for $Q^{\mathbf{t}}(n|\alpha)$, where its second claim becomes false while the first claim becomes true. Corollary 44 requires $n > m$ in its premise, which does not apply in this case, hence it continues to hold.

In case 2, $\{Q^{\mathbf{t}}(n|\alpha) \rightarrow_{\mathcal{R}_2^*} D^{\mathbf{t}'}(n|\alpha)\}$ is added to $\mathcal{R}_2^*$ through the second branch in the loop of the construction procedure. Corollary 45 continues to hold for $Q^{\mathbf{t}}(n|\alpha)$, where its last claim becomes false while the first claim becomes true. Corollary 44 requires $n > m$ in its premise, which does not apply in this case, hence it continues to hold.

In case 3, $\{Q^{\mathbf{t}}(n|\alpha) \rightarrow_{\mathcal{R}_2^*} A^{\mathbf{t}'}(m|\beta)\}$ ($n > m$) is added to $\mathcal{R}_2^*$ through the third branch in the loop of the construction procedure. Since $n > m$, $\alpha \neq \beta$. Corollary 44 continues to hold for $Q^{\mathbf{t}}(n)$, where its second claim becomes false while the first claim becomes true. Corollary 45 requires $\alpha = \beta$ in its premise, which does not apply in this case, hence it continues to hold.

In case 4, if $\{D^{\mathbf{t}'}(m|\beta) \rightarrow_{\mathcal{R}_2^*} Q^{\mathbf{t}}(n|\alpha)\}$ is added to $\mathcal{R}_2^*$ through the fourth branch in the loop of the construction procedure. Since $n > m$, $\alpha \neq \beta$. Corollary 44 continues to hold for $Q^{\mathbf{t}}(n|\alpha)$, where its last claim becomes false while the first claim becomes true. Corollary 45 requires $\alpha = \beta$ in its premise, which does not apply in this case, hence it continues to hold. □

Lemma 51. *$\mathcal{R}_2^*$ is a superset of $\mathcal{R}_2$, the initial partial order on all operations.*

Proof. The construction procedure only adds new pairs to $\mathcal{R}_2^*$, which initially equals to $\mathcal{R}_2$. □

Lemma 52. *$\mathcal{R}_2^*$ forms a directed acyclic graph (DAG), in which the set of vertexes includes all operations, and for two operations op_1 and op_2, there is a directed edge from op_1 to op_2 iff. $op_1 \rightarrow_{\mathcal{R}_2^*} op_2$.*

Proof. By induction on the number of iterations k.

Basic Step. $k = 0$. $\mathcal{R}_2^*$ is initialized as $\mathcal{R}_2$. The partial order imposed by $\mathcal{R}_2$ creates a DAG.

Inductive Step. $k > 0$. Given that no cycle is formed in the first k iterations, we the $(k+1)$th iteration does not form a cycle.

An ordering relation added to $\mathcal{R}_2^*$ in the $(k+1)$th iteration must take one of the following forms:

1. $\{A^{\mathbf{t}'}(n|\alpha) \rightarrow_{\mathcal{R}_2^*} Q^{\mathbf{t}}(n|\alpha)\}$
2. $\{Q^{\mathbf{t}}(n|\alpha) \rightarrow_{\mathcal{R}_2^*} D^{\mathbf{t}'}(n|\alpha)\}$
3. $\{Q^{\mathbf{t}}(n|\alpha) \rightarrow_{\mathcal{R}_2^*} A^{\mathbf{t}'}(m|\beta)\}$ and $n > m$
4. $\{D^{\mathbf{t}'}(m|\beta) \rightarrow_{\mathcal{R}_2^*} Q^{\mathbf{t}}(n|\alpha)\}$ and $n > m$

In case 1, $\{A^{\mathbf{t}'}(n|\alpha) \rightarrow_{\mathcal{R}_2^*} Q^{\mathbf{t}}(n|\alpha)\}$ is added to $\mathcal{R}_2^*$ through the first branch in the loop of the construction procedure. By contradiction, suppose adding this edge forms a cycle. Then there exists a chain of operations $Q^{\mathbf{t}}(n|\alpha) \rightarrow_{\mathcal{R}_2^*} op_1 \rightarrow_{\mathcal{R}_2^*} op_2 \rightarrow_{\mathcal{R}_2^*} \ldots \rightarrow_{\mathcal{R}_2^*} op_n \rightarrow_{\mathcal{R}_2^*} A^{\mathbf{t}'}(n|\alpha)$ created by previous iterations.

First note that the chain is not empty, as otherwise $Q^{\mathbf{t}}(n|\alpha) \rightarrow_{\mathcal{R}_2^*} A^{\mathbf{t}'}(n|\alpha)$ was added in a previous iteration. This is impossible, because no pair matching such a pattern can be added by the construction procedure, and since $Q^{\mathbf{t}}(n|\alpha)$ is concurrent with $A^{\mathbf{t}'}(n|\alpha)$, the pair is not initially in $\mathcal{R}_2$.

We claim that for every op_i in the chain, op_i is concurrent with $A^{\mathbf{t}'}(n|\alpha)$ in $\mathcal{R}_2$. If either $op_i \rightarrow_{\mathcal{R}_2} A^{\mathbf{t}'}(n|\alpha)$, or $A^{\mathbf{t}'}(n|\alpha) \rightarrow_{\mathcal{R}_2} op_i$, there is a contradiction with the induction hypothesis, in that a cycle must have been formed in a previous iteration. In the former case, $op_i.res \rightarrow_{\mathcal{R}_1} A^{\mathbf{t}'}(n|\alpha).inv \rightarrow_{\mathcal{R}_1} Q^{\mathbf{t}}(n|\alpha)$, so we have $op_i \rightarrow_{\mathcal{R}_2} Q^{\mathbf{t}}(n|\alpha)$ by definition (also, $op_i \rightarrow_{\mathcal{R}_2^*} Q^{\mathbf{t}}(n|\alpha)$ since $\mathcal{R}_2 \subset \mathcal{R}_2^*$). Taken together with the chain, this forms a cycle that must have been added previously. In the latter case, $Q^{\mathbf{t}}(n|\alpha) \rightarrow_{\mathcal{R}_1} A^{\mathbf{t}'}(n|\alpha).res \rightarrow_{\mathcal{R}_1} op_i.inv$, and it follows that $A^{\mathbf{t}'}(n|\alpha) \rightarrow_{\mathcal{R}_2} op_i$. When combined with the chain, this also forms a cycle that must have been added previously.

Therefore, we know that for the chain to exist, edge $op_n \rightarrow_{\mathcal{R}_2^*} A^{\mathbf{t}'}(n|\alpha)$ must have been added to $\mathcal{R}_2^*$ by the third branch of the construction procedure, in some previous iteration. Thus, op_n must be a $Q^{\mathbf{t}''}(p|\gamma)$ where $p > n$. On the other hand, since all `Query`s are totally ordered in $\mathcal{R}_2$, it must hold that $Q^{\mathbf{t}}(n|\alpha) \rightarrow_{\mathcal{R}_2} op_n$: the alternative, $op_n \rightarrow_{\mathcal{R}_2} Q^{\mathbf{t}}(n|\alpha)$, would require a cycle in the assumed chain of operations. Given that $Q^{\mathbf{t}}(n|\alpha) \rightarrow_{\mathcal{R}_2} Q^{\mathbf{t}''}(p|\gamma)$, with both queries concurrent with $A^{\mathbf{t}'}(n|\alpha)$, this creates a contradiction with Lemma 46, which requires $p \leq n$. Thus the chain could not exist, and adding this edge will not form a cycle.

Similarly, in case 2, we assume by contradiction that a chain $D^{\mathbf{t}'}(n|\alpha) \rightarrow_{\mathcal{R}_2^*} op_1 \rightarrow_{\mathcal{R}_2^*} op_2 \rightarrow_{\mathcal{R}_2^*} \ldots \rightarrow_{\mathcal{R}_2^*} op_n \rightarrow_{\mathcal{R}_2^*} Q^{\mathbf{t}}(n|\alpha)$ was created by previous iterations. Using the same logic, we would have to conclude that $D^{\mathbf{t}'}(n|\alpha) \rightarrow_{\mathcal{R}_2^*} op_1$ was previously added through the fourth branch of the construction procedure. Thus, op_1 must be a $Q^{\mathbf{t}''}(p|\gamma)$ where $p > n$. Combining this deduction with the assumption that $op_1 \rightarrow_{\mathcal{R}_2} Q^{\mathbf{t}}(n|\alpha)$ creates a contradiction with Lemma 47, which requires $p \leq n$.

In case 3, we assume by contradiction that a chain $A^{\mathbf{t}'}(m|\beta) \rightarrow_{\mathcal{R}_2^*} op_1 \rightarrow_{\mathcal{R}_2^*} op_2 \rightarrow_{\mathcal{R}_2^*} \ldots \rightarrow_{\mathcal{R}_2^*} op_n \rightarrow_{\mathcal{R}_2^*} Q^{\mathbf{t}}(n|\alpha)$ ($n > m$) was created by previous iterations. We similarly conclude $A^{\mathbf{t}'}(m|\beta) \rightarrow_{\mathcal{R}_2^*} op_1$ was added through the first branch of the construction procedure. Thus, op_1 must be a $Q^{\mathbf{t}''}(m|\gamma)$ ($\gamma = \beta$). Combining this deduction with the assumption that $op_1 \rightarrow_{\mathcal{R}_2} Q^{\mathbf{t}}(n|\alpha)$ creates a contradiction with Corollary 48, which requires $\gamma \neq \beta$.

In case 4, we assume by contradiction that a chain $Q^{\mathbf{t}}(n|\alpha) \rightarrow_{\mathcal{R}_2^*} op_1 \rightarrow_{\mathcal{R}_2^*} op_2 \rightarrow_{\mathcal{R}_2^*} \ldots \rightarrow_{\mathcal{R}_2^*} op_n \rightarrow_{\mathcal{R}_2^*} D^{\mathbf{t}'}(m|\beta)$ ($n > m$) was created by previous iterations. We similarly conclude $op_n \rightarrow_{\mathcal{R}_2^*} D^{\mathbf{t}'}(m|\beta)$ is added through the second branch of the construction procedure. Thus, op_1 must be a $Q^{\mathbf{t}''}(m|\gamma)$ ($\gamma = \beta$). Combining this deduction with the assumption that $Q^{\mathbf{t}}(n|\alpha) \rightarrow_{\mathcal{R}_2} op_n$ creates a contradiction with Corollary 49, which requires $\gamma \neq \beta$. □

Lemma 53. *Every topological order of $\mathcal{R}_2^*$ is a legal sequential history.*

Proof. As shown in Lemma 50, Corollaries 44 and 45 hold as invariants for the loop of the construction procedure. When the loop terminates, we know that only the first claim applies in each of the two Corollaries. By contradiction, suppose the second or the third claim still held. In that case, the loop would find such a concurrent operation and add an ordering pair to $\mathcal{R}_2^*$ before terminating.

Furthermore, we know that if a `Query()` operation $Q^{\mathbf{t}}(n|\alpha)$ is concurrent with an `Arrive()` operation $A^{\mathbf{t}}(m|\beta)$ or a `Depart()` operation $D^{\mathbf{t}}(m|\beta)$, then $n \leq m$ and $\alpha \neq \beta$. Otherwise, such concurrent operations would be found and added to $\mathcal{R}_2^*$ through one of the branches before the loop terminates. Assigning any ordering between two such operations is always safe for Corollary 44, which requires $n > m$ in its premise. The validity of Corollary 45's first claim is not affected by ordering these operations in $\mathcal{R}_2^*$, since they have different ids. Thus, every topological order of $\mathcal{R}_2^*$ conforms to the sequential specification in Section 2. □

Theorem 54. *The Mindicator algorithm is linearizable.*

Proof. Lemma 51 and Lemma 53 indicate the augmented Mindicator algorithm is a linearizable implementation according to the specification in Section 2, which implies the correctness of the original algorithm. □

6 Relaxed Mindicators

In many applications, the progress guarantees that Mindicators offer are stronger than needed. We discuss optimizations to the Mindicator algorithm based on relaxing either nonblocking progress, or linearizability.

6.1 Lock-Based Mindicators

To implement a Mindicator using fine-grained locking, we replace the version numbers and state bits with a lock. Since `Arrive()` and `Depart()` operations begin at a leaf and traverse upward, there is no risk of deadlock: each node can be locked on first access, up to and including the root. Furthermore, both strict two-phase locking and hand-over-hand fine-grained locking are acceptable methodologies.

Clearly, if all queries locked the root, the Mindicator would not scale. To ensure scalability, we note that so long as the `min` field of the root is an atomic register object supporting reads and writes, query operations need not acquire a lock: they may simply read the root node's value field.

This lock-based Mindicator maintains the property that every `Arrive()` and `Depart()` operation that does not reach the root must touch an intermediate node I without changing its `min` value (that is, it must lock the node; analogously, in the nonblocking algorithm it would have updated only the node's version number). Given this condition, and the ability to perform multi-word updates to an internal node when it is locked, we introduce a Mindicator optimized for locality. To a non-leaf internal node with W children, we add an array of W integers. Each array entry is assigned to one of the node's children, and caches the `min` value of that child. In this manner, depart operations no longer need to look at sibling nodes when computing the new minimum value of a parent; rather, they need only scan through the summary array embedded in a parent node.

To maintain the values in the array, we require that an arriver at Node N who is propagating an arrive of value v from child C update $N.W[C]$ to hold v after locking N. Similarly, a departer at Node N must set $N.W[C]$ to $\top$. Note that this technique is orthogonal to the locking methodology (two-phase or hand-over-hand).

Obviously, this technique has no impact on the asymptotic complexity of the algorithm. However, as it reduces the number of nodes (and cache lines) accessed, It should have low overhead in the common case, particularly when the entire node, including $W[]$, fits in a single cache line.

6.2 Mindicators with Quiescent Consistency

In our initial specification, we presented a regular expression describing the valid order in which operations may occur within a single thread. While we treated all query operations issued by a thread ($Q^{\mathbf{t}}$) as having the same semantics, this is not strictly necessary. More specifically, we can separate these queries into $Q_a^{\mathbf{t}}$ and $Q_d^{\mathbf{t}}$, corresponding to queries in which the last non-query operation was an arrive, and those in which the last non-query operation was a depart (or for which there were no previous non-query operations). We may then restate our regular expression as follows:

$$H_i = (Q_d^{\mathbf{t}})^*(A^{\mathbf{t}}(Q_a^{\mathbf{t}})^* D^{\mathbf{t}}(Q_d^{\mathbf{t}})^*)^*(A^{\mathbf{t}})^?(Q_a^{\mathbf{t}})^*$$

We can use this expression to exploit the fact that some applications do not require queries after departure to be linearizable. This relaxes the liveness condition, so that the value returned by a query operation must correspond to some value in the live set only if there are no pending operations from other threads.

Note that we do not relax the safety condition: a Mindicator Query must return a value that is less than or equal to the smallest value in the Mindicator. We are only relaxing the set of values that a query may return when there are pending operations. This results in a Quiescently Consistent implementation [5, Ch 3.3].

In effect, this relaxation means that if a thread $\mathbf{t}$ departs, and thus removes some value $v^{\mathbf{t}}$, then if $v^{\mathbf{t}}$ was the smallest value in the Mindicator prior to $\mathbf{t}$'s departure, the Mindicator may continue to return $v^{\mathbf{t}}$ in response to queries until some point when there are no pending operations.

Figure 5 presents the `Depart()` and `Revisit()` operations for a Quiescently Consistent Mindicator. The `Arrive()` and `Query()` operations are unchanged, and omitted. The essence of the change to `Depart()` is that `Depart()` on node N only recurses upward if (a) N is stable and (b) N's value increases as a result of the `Depart()` calling `Revisit()`. In this case, the departing thread $\mathbf{t}$ must assume that no concurrent thread is performing an `Arrive()` or `Depart()` on N that will eventually propagate $\mathbf{t}$'s departure upward.

7 Evaluation

We evaluate Mindicators using a targeted microbenchmark and a synthetic STM workload. Experiments labeled "Niagara2" were collected on a 1.165 GHz, 64-way Sun UltraSPARC T2 with 32 GB of RAM, running Solaris 10. The Niagara2 has eight cores, each eight-way multithreaded. On the Niagara2, code was compiled using gcc 4.3.2 with –O3 optimizations. Experiments labeled "x86" were collected on a 12-way HP z600 with 6GB RAM and a 2.66GHz Intel Xeon X5650 processor with six cores, each two-way multithreaded, running Linux kernel 2.6.37. The x86 code was compiled using gcc 4.4.5, with –O3 optimizations. On both machines, the lowest level of the cache hierarchy is shared among all threads. However, the Niagara2 cores are substantially simpler than the x86 cores, resulting in different instrumentation overheads.

```
datatype NODE
  sta   : B   ▷ state bit
  min   : N   ▷ minimum of children
  ver   : N   ▷ version number

initially NODE (sta, min, ver)
  = (steady, ⊤, 0)

function QCQuery(X : NODE) : N
  ▷ ... same with linearizable Query ...

procedure QCArrive(X : NODE, n : N)
  ▷ ... same with linearizable Arrive ...

procedure QCDepart(X : NODE, n : N)
49: if QCRevisit(X)
50:     QCDepart(parentof(X), n)

function QCRevisit(X : NODE) : B
51: while true do
52:     x ← Read(X)
53:     min ← ⊤
54:     if x.sta = tentative
55:         return false
56:     for C in childrenof(X) do
57:         c ← Read(C)
58:         if c.min < min
59:             min ← c.min
60:     if min ≤ x.min
61:         return false
62:     elif CAS(X, x, (steady, min, x.ver + 1))
63:         return true
```

Figure 5: The Quiescently Consistent Mindicator Algorithm. The datatypes, `Arrive()` and `Query()` functions are unchanged from Figure 2

7.1 Baseline List-Based Set

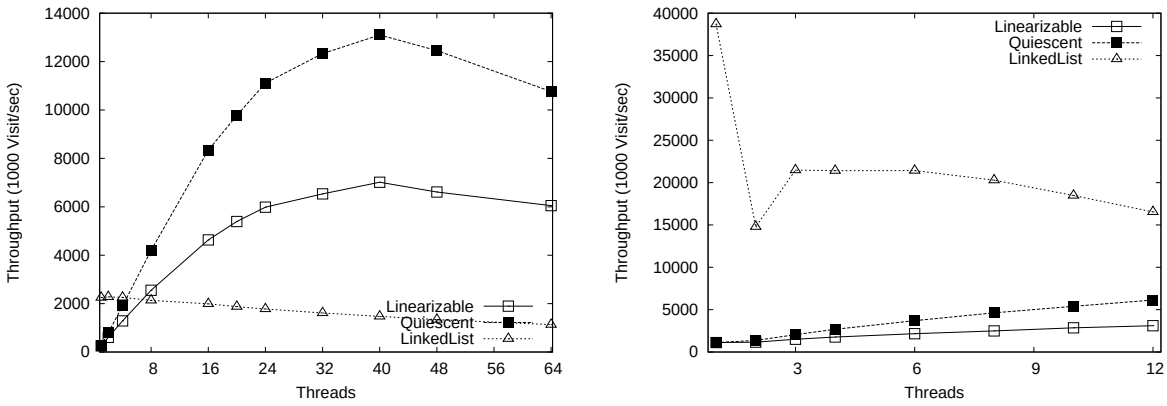
As a baseline, we use a sorted doubly-linked list. The list is protected by a single coarse-grained lock, which minimizes latency. On the x86, the single lock is very efficient, since the relatively small core count and the aggressive implementation of CAS operations in the private L1 cache lead to low overhead. On the Niagara2, where CAS is implemented in the shared L2, fine-grained locking scales well, but has too much latency to provide a good baseline. Using the list, `Arrive()` costs $O(|T|)$. Since the list has back-edges, we save a pointer to the inserted node, and `Depart()` costs $O(1)$. Furthermore, since the list is protected by a single lock, we save a snapshot of the minimum value separate from the head pointer. In this manner, an $O(1)$ query can read the snapshot without acquiring the lock.

7.2 Mindicator Implementation Details

We evaluate the lock-free Mindicator (Section 3), and the quiescently consistent (QC) Mindicator (Section 6.2). We configured the Mindicators to have out-degree of 2, and 64 leaves. In Section 8, we present a technique that shows how these constraints can be easily overcome. We expect our performance at low thread counts to improve significantly through application of the suggested techniques. To support 32-bit values, ver is a 31-bit counter, and min is a 32-bit integer, so the entire Node can be modified with a 64-bit CAS. We optimized the implementation by (a) eliminating recursion in the tail-recursive `Depart()` operation; (b) employing 32-bit CAS operations for those instructions that only change the version number; and (c) providing a fast-path so leaf node updates use an ordered store instead of a CAS on the Niagara2.

7.3 Stress Test Microbenchmark

We stress test the Mindicator through a targeted microbenchmark where all threads repeatedly arrive and depart with randomly generated values. This test emphasizes the cost of `Arrive()` and `Depart()`, since there are no `Query`s. Furthermore, since there is no program code apart from `Arrive()` and `Depart()`, this over-emphasizes the cost of atomic operations. In particular, on the x86, where CAS is performed in the L1, the baseline (list) algorithm allows a single thread to rapidly execute many successive `Arrive()`



(a) Niagara2

(b) x86

Figure 6: Stress-test microbenchmark: all threads repeatedly Arrive and Depart with random values.

and `Depart()` operations, while other threads starve. This results in an inflated throughput curve in Figure 6(b). On the Niagara2 (Figure 6(a)), the lack of processor optimizations for uncontended lock acquisition results in a fundamentally different behavior: the list hardly scales, and despite higher latency the Mindicator scales to a throughput $6\times$ that of the list. Both charts also demonstrate the cost of linearizability, with the QC Mindicator significantly outperforming the linearizable Mindicator.

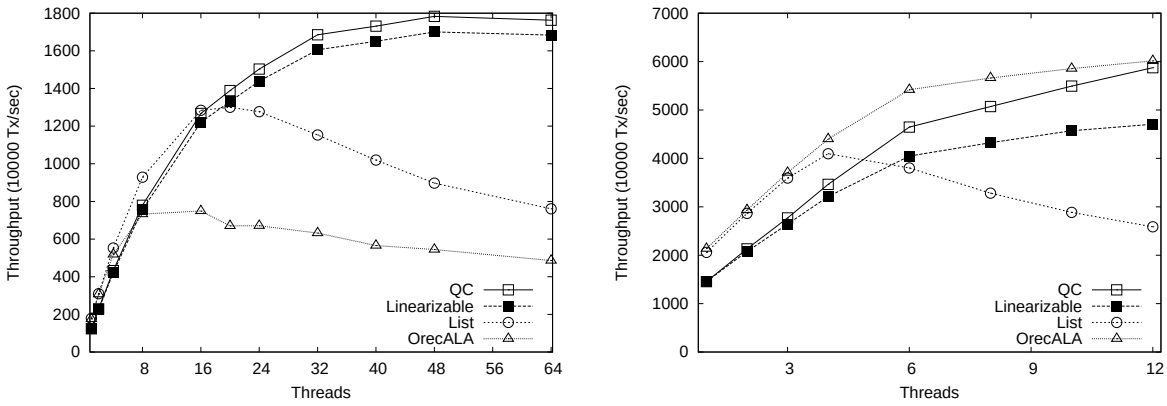
In further experiments, we found that the addition of a `Query()` thread had little impact on the Niagara2 result, but that on the x86, adding `Query()` threads decreased the performance gap between the list and Mindicators. In short, since our baseline list implementation has an optimized `Query()` operation, a single thread repeatedly issuing `Query()` operations creates coherence traffic and reduces the likelihood that the snapshot remain in a single core's cache for consecutive `Arrive()` and `Depart()` operations.

7.4 Synthetic TM Workload

While the previous experiment demonstrates the ability of the Mindicator to scale, it is too artificial to draw strong conclusions. In particular, the test does not give insight into a workload where threads announce their state, perform unrelated work, and then clear their state. In such a case, the application working set will contain more than just the Mindicator.

To evaluate this more realistic setting, we created a new TM algorithm, based on the work of Menon et al. [15]. In previous work, it was shown that Menon's algorithm for providing strong semantics in an STM was comparable in performance to a mechanism that employs a ticket lock to serialize transaction commit [17]. We extended the latest version of RSTM [12, 16] with a Mindicator-based version of Menon's algorithm. RSTM refers to the variant with a Ticket lock as OrecALA; we label the other curves in Figure 7 based on whether a list, a QC Mindicator, or a linearizable Mindicator is used with Menon's algorithm. To test the STM implementations, we use a simple red-black tree benchmark. Threads perform an equal mix of insert, remove, and lookup operations, using 8-bit keys.

Again, we find that the QC algorithm provides excellent scalability, but the gap between QC and linearizability is much less. Furthermore, the list-based implementations cease to offer satisfactory performance, with performance degrading on both machines at a low thread count. On the Niagara2, the Mindicator-based



(a) Niagara2

(b) x86

Figure 7: RBTree performance for STM implementations that provide strong semantics using Mindicators.

algorithms continue to scale well beyond the point at which OrecALA plateaus. On the x86, performance is comparable. We suspect that applying a rebalancing strategy (Section 8) would recover the remaining gap on the x86. Furthermore, there is a direct relationship between the length of transactions and the benefit of Mindicators. In additional experiments, we found that for tiny transactions (e.g., Hashtable updates) Mindicators offered less benefit (especially on the x86), whereas when transaction durations increased, or write set sizes grew, the benefit of the Mindicator increased. In essence, as transactions grow more complex, the superior scalability of the Mindicator-based algorithms provides an increasing advantage over OrecALA and list-based implementations of Menon's algorithm.

8 Unbounded Mindicators

Thus far, we have required that every thread be assigned a unique leaf node of the Mindicator. Such an approach would seem to require static knowledge of the maximum number of threads that might ever use the Mindicator. Additionally, such static estimation is likely to be conservative, resulting in added latency when few threads are using a Mindicator intended to support a large thread count. We briefly discuss two approaches for supporting unbounded Mindicators. Both approaches are lock-free.

8.1 Adding Children to Leaves

The simplest approach is to increase the out-degree of each node in the Mindicator tree, and designate one child of each Node (e.g., the first child) as a distinguished childless node (DCN). Each thread is then assigned a DCN at which it must `Arrive()` and `Depart()`, rather than being assigned a leaf. Note that such a change does not affect the correctness of the algorithm in Figure 2.

Once there is a DCN, threads need not arrive and depart at leaves. Thus, when all current nodes in the Mindicator have been assigned to threads, a new thread need only create a new node and atomically install it as a child of some leaf node. Note that this leaf node will still have a DCN. In practice, the thread will actually install W children, where W is the pre-set out-degree of Nodes.

As a simple but powerful optimization, the fields of the DCN can be inlined directly into its parent. Additionally, matters of priority can be addressed by assigning higher-priority threads to nodes that are closer to the root. Since even the root node has a DCN, a single thread can be guaranteed minimal CAS operations and no recursion, if desired.

8.2 Adding a Parent to the Root

An alternative approach, which preserves the property of all threads arriving at leaves of identical depth, is to dynamically change the root. When all leaves of a Mindicator have been assigned to threads, a newly created thread could first create $W - 1$ empty, uninitialized Mindicator trees of equal depth to the existing Mindicator, where W is the out-degree of nodes in the Mindicator. It then creates a new Node (R'), which will become the root, and sets these trees as children of R'. The new thread then sets the original Mindicator as a child of R', but does not yet set R' as the parent of the original Mindicator's root.

The new thread then claims a node in the new Mindicator, and performs an `Arrive()` with the minimum possible value (this step is necessary for safety). Once this is done, the new thread can atomically install R' as the parent of the original Mindicator's root node. At this point, the global pointer to the Mindicator must be atomically updated, via an operation that any thread can perform on behalf of the new thread. Finally, the new thread performs a `Depart()` to remove its artificial minimal value. In this manner, all invariants are preserved in the Mindicator at all times.

8.3 Dynamic Rebalancing

When the number of threads varies during program execution, there may be times where the Mindicator is far deeper than necessary, resulting in unnecessary latency. A simple, orthogonal mechanism suffices to address this case. When there are DCNs, we need only re-assign threads to DCNs that are closer to the root. This effectively "pushes up" the arrival point of threads, to make them closer to the root.

When Mindicators are resized by adding a parent to the root, we first reassign threads to the leftmost set of leaves in the tree. We then set the global root pointer to the lowest node in the tree that is an ancestor of all these leaves, and then we mark that common ancestor's parent pointer as unusable. This effectively "pushes down" the root, to make it closer to the arrival point of threads.

9 Conclusions and Future Work

In this paper we introduced the Mindicator, a set-like object optimized for shared-memory runtime systems. The Mindicator is lock-free, scalable, and offers lower asymptotic overhead than the state-of-the-art. We proved the lock-free Mindicator implementation correct, and demonstrated through stress tests and STM-based microbenchmarks that the Mindicator is able to achieve its performance goals.

In addition to the existence of the Mindicator, which we expect to have impact on a variety of shared-memory runtime systems, we offer two additional artifacts. First, our proof technique is nuanced, on account of the linearization point for operations not being clearly defined. While not the first algorithm with this property, we hope the structure of our proof can help in the eventual discovery of a general proof strategy for this class of algorithms. Second, our experimental results add to the mounting evidence that shared memory microbenchmark performance on the x86 is not an indicator of the performance of systems. The variety of optimizations that the x86 employs (such as performing CAS at the private L1) necessitate that new runtime systems be evaluated in an environment with realistic cache behaviors.

As future work, we intend to explore how QC Mindicators and unbounded Mindicators can improve performance. We also plan to explore the ways in which a Mindicator can improve runtime systems, such as OS components, TM, and GC. We are also exploring the use of hardware TM to accelerate Mindicators, using techniques similar to those proposed by Dice et al. [1].

References

[1] D. Dice, Y. Lev, V. Marathe, M. Moir, M. Olszewski, and D. Nussbaum. Simplifying Concurrent Algorithms by Exploiting Hardware TM. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, Santorini, Greece, June 2010.

[2] F. Ellen, Y. Lev, V. Luchangco, and M. Moir. SNZI: Scalable NonZero Indicators. In *Proceedings of the Twenty-Sixth ACM Symposium on Principles of Distributed Computing*, Portland, OR, Aug. 2007.

[3] K. Fraser. *Practical Lock-Freedom*. PhD thesis, King's College, University of Cambridge, Sept. 2003.

[4] R. Guerraoui, M. Herlihy, and B. Pochon. Toward a Theory of Transactional Contention Managers. In *Proceedings of the 24th ACM Symposium on Principles of Distributed Computing*, Las Vegas, NV, July 2005.

[5] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[6] M. P. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th International Symposium on Computer Architecture*, San Diego, CA, May 1993.

[7] M. P. Herlihy and J. M. Wing. Linearizability: a Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[8] R. L. Hudson, B. Saha, A.-R. Adl-Tabatabai, and B. Hertzberg. A Scalable Transactional Memory Allocator. In *Proceedings of the International Symposium on Memory Management*, Ottawa, ON, Canada, June 2006.

[9] E. Koskinen and M. Herlihy. Concurrent Non-commutative Boosted Transactions. In *Proceedings of the 4th ACM SIGPLAN Workshop on Transactional Computing*, Raleigh, NC, Feb. 2009.

[10] C. Lameter. Effective Synchronization on Linux/NUMA Systems. In *Proceedings of the May 2005 Gelato Federation Meeting*, San Jose, CA, May 2005.

[11] Y. Lev, V. Luchangco, V. Marathe, M. Moir, D. Nussbaum, and M. Olszewski. Anatomy of a Scalable Software Transactional Memory. In *Proceedings of the 4th ACM SIGPLAN Workshop on Transactional Computing*, Raleigh, NC, Feb. 2009.

[12] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M. L. Scott. Lowering the Overhead of Nonblocking Software Transactional Memory. In *Proceedings of the 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, Ottawa, ON, Canada, June 2006.

[13] V. J. Marathe, M. F. Spear, and M. L. Scott. Scalable Techniques for Transparent Privatization in Software Transactional Memory. In *Proceedings of the 37th International Conference on Parallel Processing*, Portland, OR, Sept. 2008.

[14] P. E. McKenney. *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels*. PhD thesis, OGI School of Science and Engineering at Oregon Health and Sciences University, 2004.

[15] V. Menon, S. Balensiefer, T. Shpeisman, A.-R. Adl-Tabatabai, R. Hudson, B. Saha, and A. Welc. Practical Weak-Atomicity Semantics for Java STM. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, Munich, Germany, June 2008.

[16] M. Spear. Lightweight, Robust Adaptivity for Software Transactional Memory. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, Santorini, Greece, June 2010.

[17] M. F. Spear, L. Dalessandro, V. J. Marathe, and M. L. Scott. Ordering-Based Semantics for Software Transactional Memory. In *Proceedings of the 12th International Conference On Principles Of DIstributed Systems*, Luxor, Egypt, Dec. 2008.

[18] M. F. Spear, M. M. Michael, and C. von Praun. RingSTM: Scalable Transactions with a Single Atomic Instruction. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, Munich, Germany, June 2008.