# Mindicators: A Scalable Approach to Quiescence

Yujie Liu
*Lehigh University*
*yul510@cse.lehigh.edu*

Victor Luchangco
*Oracle Labs*
*victor.luchangco@oracle.com*

Michael Spear
*Lehigh University*
*spear@cse.lehigh.edu*

***Abstract*—We introduce the *Mindicator*, a new shared object that is optimized for querying the minimum value of a set of values proposed by several processes. A mindicator may hold at most one value per process. This interface is designed for use in shared memory runtime systems, such as garbage collectors, software transactional memory (TM), and operating system kernels.**

**We introduce linearizable and relaxed mindicator implementations, both of which are lock-free. Our algorithms employ a tree structure, where querying the minimum element takes constant time, and adding and removing elements from the set does not hinder scalability. In microbenchmarks and a synthetic TM workload, we show that both provide good scalability on the x86 and SPARC platforms.**

***Keywords*-concurrent data structures; synchronization; lock-freedom; linearizability;**

**Listing 1:** Mindicator Specification

$\mathbb{P}$ = set of all processes
$\mathbb{N}^\infty = \mathbb{N} \cup \{\infty\}$

**states**
  $val : \mathbb{P} \to \mathbb{N}^\infty$; initially $val(p) = \infty$ for all $p \in \mathbb{P}$

**procedure** ARRIVE($v : \mathbb{N}$)
  **requires** $val(p) = \infty$
  $val(p) \leftarrow v$

**procedure** DEPART()
  **requires** $val(p) \neq \infty$
  $val(p) \leftarrow \infty$

**function** QUERY() : $\mathbb{N}^\infty$
  **return** min $\{ \; val(p) \mid p \in \mathbb{P} \; \}$

## I. INTRODUCTION

*Quiescence* is a synchronization pattern used by many shared-memory run-time systems, in which a process may *quiesce* at time $t$ by blocking until every operation that started before time $t$ has finished. (To avoid deadlock, a process must not quiesce while it is executing an operation.) For example, transactional memory (TM) implementations use quiescence to provide privatization safety [11], [13], [15], and to prevent starvation [7], [18] and early memory reclamation [6], [9], Quiescence is also used in RCU synchronization [14], and sequence-lock-based critical sections [12].

In this paper, we introduce the *Mindicator* object, which can be used to implement quiescence, and we present two scalable mindicator implementations. A mindicator maintains a multiset of values, at most one per process, and supports three operations: ARRIVE, DEPART and QUERY. A process that does not have a value in the multiset invokes ARRIVE($v$) to add $v$ as its value; a process removes its value from the multiset by invoking DEPART. The QUERY operation returns the minimum value in the multiset. The sequential semantics of a mindicator is specified precisely in Listing 1.

We present a scalable, lock-free and linearizable [8] mindicator implementation, in which each process stores its value (if any) in a dedicated leaf of a rooted tree; internal nodes store the minimum of the values at their children. QUERY is implemented by a single instruction; ARRIVE and DEPART have $O(\log(n))$ time complexity, where $n$ is the number of processes. The implementation achieves good scalability using novel coordination among processes: in the common case, few operations access overlapping portions of the tree, and thus there are no slowdowns or bottlenecks due to memory conflicts.

We also show how to achieve an even more scalable implementation by relaxing the linearizability requirement, allowing a DEPART operation to return early if it conflicts with an ARRIVE operation propagating a smaller value. In this case, the DEPART operation will take effect after it returns but before the conflicting ARRIVE returns. Alternatively (and equivalently), we can consider that ARRIVE and DEPART operations take effect during their execution intervals, but a QUERY operation may return a value smaller than the minimum value currently in the multiset if it overlaps an ARRIVE operation that overlapped the DEPART operation that removed the value. This is sufficient for quiescence because it is safe for a a quiescing process to block longer than necessary.

We evaluate our mindicator implementations on Intel x86 and Oracle SPARC architectures using microbenchmarks and within a TM implementation, demonstrating their scalability. The relaxed mindicator offers partic-

ularly good performance, along with stronger progress guarantees than the current state of the art, making it suitable for using within TM runtime systems.

The remainder of this paper is organized as follows: In section II, we discuss related work and additional applications of the mindicator. We present the details of the mindicator algorithms in section III, and prove correctness and lock-freedom in section IV. In section V, we provide performance evaluation results on the Intel x86 and Oracle SPARC architectures.

## II. Related Work

The mindicator and its implementations are derived from the SNZI object [5], on which queries indicate whether the value of a counter is zero or nonzero but not the precise value when the counter is nonzero, and its tree-based implementation.

Jayanti [10] proposed the f-array, an algorithm that allows wait-free computation of an arbitrary function $f$ over the values proposed by processes. An f-array uses a tree structure, and requires that every process propagate its value from the leaf to the root, making the root a potential hot spot of contention. SNZI and mindicator can be viewed as specializations of the f-array object for specific functions $f$, which admit cheaper and more scalable implementations.

A mindicator can be implemented straightforwardly using existing data structures such as a skip list. However, lock-free skip list implementations often require garbage collection [6] or reference counting [19], making them less suitable in unmanaged environments. Also, due to frequent node allocation, skip lists typically exhibit poor locality, which hurts performance (see Section V).

Afek, Dauber and Touitou [1] provided an adaptive universal construction of shared objects in which operations propagate up a binary tree using LL/SC, and are combined to store the state of the object in the root. This implementation is adaptive: the time complexity is bounded by the number of processes that actually access the object. However, processes may still contend at the root of the binary tree, where operations are combined, and the update procedure requires the entire data structure to be copied, incurring significant overhead.

Aspnes, Attiya and Censor [2] provided a logarithmic construction of Max-Registers, an object that supports a read operation that returns the maximum value written thus far. However, their algorithm assumes that the data structure is used only a polynomial number of times, which makes it unsuitable for the long-running problem domains we consider.

**Listing 2:** Mindicator Types and Data

```
type Node = ⟨ val : ℕ^∞, dirty : 𝔹 ⟩
data (for each process p)
  val_p          : ℕ          // p's value
  O_p[0...d_p]   : Node*[ ]   // nodes from root to p's leaf
```

**Listing 3:** A Simple Tree-based Algorithm

```
    function QUERY() : ℕ^∞
        // Read the value of root node.
        r ← READ(O_p[0])
        return r.val

    procedure ARRIVE(v : ℕ)
        val_p ← v
        for tp ← d_p ... 0 do
L1          x ← O_p[tp]
            if x.val ≤ val_p then break
S1          O_p[tp].val ← val_p

    procedure DEPART()
        for tp ← d_p ... 0 do
L2          x ← O_p[tp]
            if x.val < val_p then break
            min ← ∞
            for each child C of O_p[tp] do
                if C.val < min then min ← C.val
S2          O_p[tp].val ← min
```

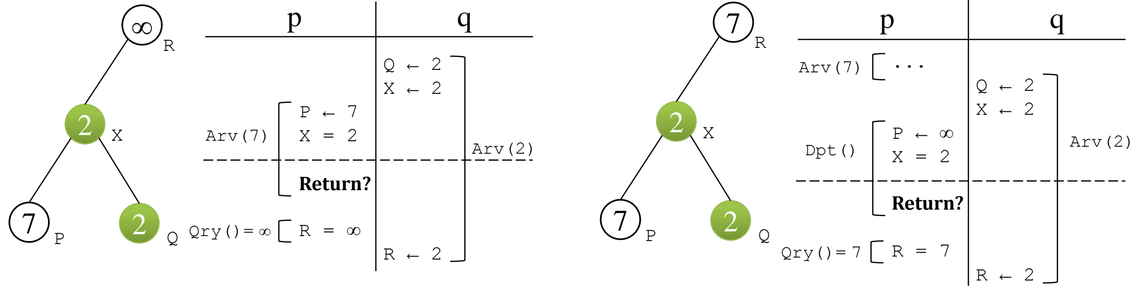
## III. Mindicator Algorithms

We implement a mindicator as a rooted tree of node objects. Each node contains a pair of fields, an integer $val$ and a boolean $dirty$, that are accessed together using load-linked/store-conditional (LL/SC). Each node is initialized to $\langle \infty, \textbf{false} \rangle$. In this paper, the structure of a mindicator tree is static; that is, the number of tree nodes and the parent/child links do not change. We could support a dynamically shaped mindicator tree, but this is not the focus of this paper.

A mindicator supports a finite (but unbounded) number of processes, equal to the number of leaves of the tree. Each process is assigned a dedicated leaf, and the assignment does not change during execution. For each process $p$, the integer $val_p$ stores the value $p$ inserted into the mindicator, and $O_p[0 \dots d_p]$ maintain pointers to the nodes on the path from root to $p$'s dedicated leaf (i.e., $O_p[0]$ is the root and $O_p[d_p]$ is $p$'s dedicated leaf).

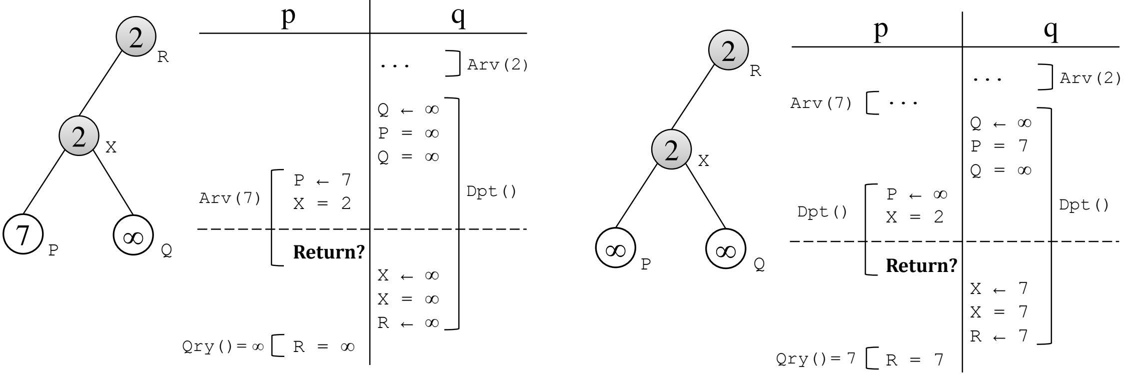
### A. A Tree-based Algorithm

We begin with a simple tree-based algorithm shown in Listing 3. For now, assume the operations do not overlap, e.g., Query, Depart and Arrive are protected by a single mutex lock. This algorithm does not require the $dirty$ field. Query is a single read on the root of the mindicator tree, returning the contents of its




**(a)** p.`Arrive(7)` encounters an unfinished q.`Arrive(2)` at X. Has 2 reached R?

**(b)** p.`Depart()` encounters an unfinished q.`Arrive(2)` at X. Has 7 been removed from R?




**(c)** p.`Arrive(7)` encounters an unfinished q.`Depart()` at X. Is 2 being removed from X?

**(d)** p.`Depart(7)` encounters an unfinished q.`Depart()` at X. Can 7 reappear at X?

Figure 1: Problematic interleavings among concurrent `Arrive` and `Depart` operations

$val$ field. Thus, the root node must keep the minimum value. To achieve this, ARRIVE and DEPART operations maintain the property that every internal node stores the minimum value stored by its children. In ARRIVE, process $p$ starts from its dedicated leaf, writing its value $val_p$ at nodes in order to lower their $val$ fields. In DEPART, the process first writes $\infty$ to its dedicated leaf. Then the process traverses upwards. For each node visited, it updates the $val$ field to the minimum $val$ of its children.

Both ARRIVE and DEPART can often return without visiting the root. An ARRIVE operation immediately returns if the process meets a node with $val \leq val_p$. In that case, every ancestor of that node has $val \leq val_p$ and hence, there is no need to lower their $val$ fields. A DEPART operation returns if it meets a node with $val < val_p$, since every ancestor of that node also has $val < val_p$. That is, none of the ancestors has $val = val_p$. It is easy to verify that at operation boundaries the value of each node is the minimum of its children, and the value of the root node is the minimum of the leaf nodes.

The tree-based algorithm benefits scalability in several ways: First, both ARRIVE and DEPART operations change only a fraction of the tree, which reduces unnecessary sharing. Second, in the common case these operations return without touching the root node. Third, a process that frequently performs QUERY will keep the root node in its local cache until the minimum value is changed (i.e., by the ARRIVE of a new minimum or the DEPART of the old minimum).

### B. Unsafe Interleavings

Unsurprisingly, the simple tree-based algorithm fails if the steps of operations can interleave, even if lines L1 and L2 use LL instructions, and lines S1 and S2 use SC instructions. Two problems arise with this naïve approach when a ARRIVE or DEPART returns without accessing the root node:

**Problem 1: Trusting an Unfinished Arrive** In Figure 1a, an ARRIVE operation incorrectly trusts the value of another unfinished ARRIVE. Reading $X.val = 2$ causes $p$'s ARRIVE operation to return immediately on node $X$. However, the ARRIVE operation by process

$q$ has not written its value 2 to the root yet. Thus $p$'s subsequent QUERY returns $\infty$, violating the correctness conditions.

Similarly, in Figure 1b, a DEPART operation trusts the value of an unfinished ARRIVE. The DEPART operation by process $p$ is removing its value 7 from the mindicator. This DEPART returns immediately after reading $X.val = 2$, which is written by an unfinished ARRIVE of process $q$, and thus neglects to remove 7 from the root node $R$.

**Problem 2: Delayed Update of Depart** Trusting the value of an unfinished DEPART can also lead to incorrect behavior. In Figure 1c, process $q$ invokes DEPART to remove its value 2 from the mindicator, but its write to node $X$ is delayed. Then process $p$'s ARRIVE(7) reads $X.val = 2$ and returns immediately, relying on the fact that 2 is already in the mindicator. Since $q$ is unaware of $p$'s ARRIVE, it will write $\infty$ at nodes $X$ and $R$, causing $p$'s subsequent QUERY to fail to see the value 2.

In Figure 1d, processes $p$ and $q$ are removing their values from the mindicator. Process $q$ reads $P.val = 7$ before $p$ sets it to $\infty$. Process $q$ then attempts to update $X.val$ to 7, but is delayed. $p$ returns immediately after reading $X.val = 2$, assuming that its value 7 has been removed. Later on, $q$'s delayed write will restore the value 7 at node $X$, and then at the root node $R$, making $p$'s DEPART operation invalid.

In comparing SNZI and f-array, we see that f-array solves these problems by requiring every operation to traverse up to the root (no early termination), whereas SNZI handles the problem by requiring some operations to traverse up from a leaf to some node, and then back down from that node to the originating leaf. We now present a linearizable, lock-free algorithm based on the latter technique.

### C. *The Lock-free Mindicator Algorithm*

We present our lock-free mindicator algorithm in Listings 4 and 5. It avoids Problem 1 above by using the *dirty* bit in each node, maintaining the following invariant: if a node $X$ is not *dirty* then $A.val \leq X.val$ for every ancestor $A$ of $X$. It avoids Problem 2 above by requiring that an ARRIVE or DEPART operation that completes without reaching the root of the tree (i.e., returns **true** from PROPAGATE or SUMMARIZE) to write the highest node reached (lines 3 and 8), causing a conflict with any DEPART operation concurrently updating that node.

In more detail, ARRIVE consists of two stages, the *propagate stage* and the *clean stage*. In the propagate stage, the process traverses from its dedicated leaf towards the root, invoking PROPAGATE on each node visited. According to the return value of PROPAGATE, the propagate stage may stop at some internal node known as the *turning point*. Then in the clean stage, CLEAN invokes on each node from the turning point back down to the dedicated leaf.

**Listing 4:** Lock-free Mindicator: Operations

```
function QUERY() : ℕ^∞
0 |  r ← READ(O_p[0])
  |  return r.val

procedure ARRIVE(v : ℕ)
  // Record the inserted value.
  val_p ← v
  // Propagate Stage
  tp ← d_p
  done ← false
  while tp ≥ 0 and ¬done do
     done ← PROPAGATE(O_p[tp])
     tp ← tp − 1
  // Clean Stage
  i ← tp + 1
  while i ≤ d_p do
     CLEAN(O_p[i])
     i ← i + 1

procedure DEPART()
  tp ← d_p
  done ← false
  while tp ≥ 0 and ¬done do
     done ← SUMMARIZE(O_p[tp])
     tp ← tp − 1
```

In DEPART, the process starts from the dedicated leaf, invoking SUMMARIZE along the path towards the root. Like the propagate stage of ARRIVE, DEPART may stop at some internal node without traversing to the root, according to the returned boolean of SUMMARIZE.

The real work of the algorithm is encapsulated in the PROPAGATE, CLEAN and SUMMARIZE operations. Each takes a node object ($X$) as its parameter. All share a similar code pattern: the process first performs a LL on $X$, then performs some computation and then updates $X$ with a SC.

A PROPAGATE by process $p$ ensures $X.val \leq val_p$. In the first case, if $p$ reads $X.val > val_p$, $p$ decreases $X.val$ to $val_p$ and sets $X.dirty$ to true. Here, $p$ must propagate its value to the ancestors. In the second case, if $p$ reads $X.val \leq val_p$ and $X.dirty$, then $p$ leaves $X$ intact, because some process $q$ (possibly $p$ itself, if there is a concurrent DEPART) must be in the midst of an ARRIVE, and has propagated $val_q$ to $X$, where $val_q \leq val_p$. However, $p$ cannot ascertain whether $q$'s ARRIVE has reached the ancestors of $X$, so $p$ must propagate its value to the parent of $X$. In the last case, $p$ reads $X.val \leq val_p$ and $\neg X.dirty$. In this case, $p$ can stop its propagation at $X$ by succeeding in a SC that does not change any field of $X$: the SC serves to

**Listing 5:** Lock-free Mindicator: Internals

```
  function Propagate(X : Node*) : 𝔹
      while true do
1         x ← LL(X)
          // Case 1: Continue propagation.
          if x.val > val_p then
2             if SC(X, ⟨val_p, true⟩) then return false
          // Case 2: Continue propagation.
          else if x.dirty then return false
          // Case 3: Stop Propagation.
3         else if SC(X, x) then return true

  procedure Clean(X : Node*)
4     x ← LL(X)
      if x.val = val_p and x.dirty then
5         SC(X, ⟨val_p, false⟩)

  function Summarize(X : Node*) : 𝔹
      while true do
6         x ← LL(X)
          // Traverse to the parent if X.dirty.
          if x.dirty then return false
          // Find the child the with minimum val.
          min ← ∞
          for each child C of X do
7             c ← READ(C)
              if c.val < min then min ← c.val
          // If we help another process propagate by decreasing
             X.val, we must set X.dirty.
8         if SC(X, ⟨min, (min < x.val)⟩) then
              // If the old value of X is smaller than val_p, the
                 whole Depart can return.
              return x.val < val_p
```

interrupt concurrent SUMMARIZE operations that might increase $X.val$.

In CLEAN, if process $p$ reads $X.val = val_p$, then $p$ attempts to set $X.dirty$ to **false**, so that any other process $q$ with $val_q \geq val_p$ can stop its propagate stage or DEPART at $X$ without traversing to the ancestors. Note that we need not retry the updating SC if it fails.

A SUMMARIZE by process $p$ ensures that $p$'s value is removed from node $X$. If $p$ finds $X$ dirty, it leaves $X$ intact and proceeds to $X$'s parent (if any). In that case, $X.val$ cannot be $p$'s value. Otherwise, $p$ computes the minimum over the *val* fields of $X$'s children and attempts to update $X.val$ using SC. This SC may reduce $X.val$ because by propagating the value of another process to $X$, in which case, it sets $X.dirty$ to **true**; otherwise, it leaves $X$ clean. If the SC fails, the operation is retried. If $X$ is clean and its previous value is smaller than $p$'s value, then SUMMARIZE returns **true**,

**Listing 6:** Relaxed Mindicator: SUMMARIZE

```
  function Summarize(X : Node*) : 𝔹
      while true do
          x ← LL(X)
*         if x.dirty then return x.val < val_p
          min ← ∞
          for each child C of X do
              c ← READ(C)
              if c.val < min then min ← c.val
*         if min < x.val and min < val_p then return true
          if SC(X, ⟨min, (min < x.val)⟩) then
              return x.val < val_p
```

allowing DEPART to stop early.

### D. Relaxed Mindicators

As discussed in the introduction, we can improve the scalability of the implementation by relaxing the linearizability requirement, allowing DEPART to return early if it conflicts with an ARRIVE operation propagating a smaller value. Our relaxed implementation guarantees that such a DEPART operation will take effect before the conflicting ARRIVE returns. Although this implementation is not linearizable, it suffices for our intended applications (i.e., quiescence synchronization).

In more detail, we add two conditions in which a process $p$ executing a SUMMARIZE($X$) operation may return **true**: if $X$ is dirty and $X.val < val_p$, or if $min < X.val$ and $min < val_p$. (The new code for SUMMARIZE appears in Listing 6. Modified lines are marked with *.) In either case, some process $q$ must be in the midst of an ARRIVE($v$) operation with $v < val_p$, and that operation will not complete until the value at every node from $X$ to the root is at most $v$ (and thus not $val_p$). Thus, although $p$'s DEPART operation might return before it has taken effect (i.e., while a QUERY operation might still return $p$'s value), it will take effect before that ARRIVE operation completes.

### E. Alternative Mindicator Interface

For some applications, a slightly different object type, specified in Listing 7, may be simpler and more convenient. In this variant, every process has exactly one value in the multiset, and the ARRIVE and DEPART operations are replaced by a single CHANGE operation.

We can easily adapt the algorithms above to implement this alternative specification by extending the mindicator tree so that the dedicated leaf of each process $p$ has two "ghost" children and a bit indicating which ghost leaf is *current*. The ghost leaves and the current bit are entirely local to $p$: no other process reads or writes them. To change its value, $p$ flips the current bit, invokes ARRIVE with its new value, starting from its

**Listing 7:** Alternative Mindicator Specification

```
ℙ = set of all processes

states
  val : ℙ → ℕ; initially 0 for all p ∈ ℙ

procedure CHANGE(v : ℕ)
  val(p) ← v

function QUERY() : ℕ
  return min { val(p) | p ∈ ℙ }
```

newly current leaf and then invokes DEPART starting from its previous current leaf. Note that one of these operations will be trivial (i.e., it will modify only local state), depending on whether the value is increasing or decreasing, so the cost of an operation is just the cost of the nontrivial operation.

## IV. CORRECTNESS

In this section, we sketch a proof that the algorithm in Listings 4 and 5 implements a lock-free linearizable mindicator object. In particular, we state the main invariants and sketch their proof and how they are used.

### A. Invariants

As usual for such proofs, we assume that local computation is done atomically with the immediately preceding step (i.e., invocation of QUERY, ARRIVE or DEPART, or access to a node). Points between these atomic steps are indicated by numbers in the left margin of Listing 5. We write $p@1$, for example, to denote that process $p$ is about to execute line 1. We denote local variables of $p$ by subscripting them with $p$ (e.g., $x_p$).

We partition the execution of the algorithm into *phases* corresponding to calls to the internal routines. Specifically, we say that process $p$ is in phase Prop($k$), and write $p$@Prop($k$), if it is executing the body of a call to PROPAGATE($O_p[k]$) (i.e., if $p@\{1, 2, 3\}$), and similarly for $p$@Cln($k$) and $p$@Sum($k$). We say that $p$ is *absent* if it has never invoked ARRIVE, or it has not invoked ARRIVE since it last returned from DEPART; we say $p$ is *present* if it has returned from ARRIVE and has not since invoked DEPART.

Much of the subtlety in the algorithm has to do with processes updating nodes with the SC at line 8. We say that a process $p$ *contends* for a node $X$ if it is at line 7 or 8 in a call to SUMMARIZE($X$) and no process has executed a successful SC on $X$ since $p$'s most recent LL. For such a process, we use $\mathit{unchecked}_p$ for the set of children of $X$ that $p$ has not read since its most recent LL.

We can think of each subtree of the mindicator tree as a mindicator object of its own, implicitly maintaining the set of processes that have arrived and not since departed at that node. When a process arrives or departs, it updates its dedicated leaf and then propagates this information up the mindicator tree, using either PROPAGATE, if it is arriving, or SUMMARIZE, if it is departing, helping other processes that it encounters along the way. The algorithm guarantees that the value at a node is the minimum of the values associated with processes that have arrived (and not subsequently departed) at that node.

To state this invariant, we need to define precisely when a process "has arrived and not since departed" at a node. We do this by adding an auxiliary variable $X.\mathit{set}$ for each node $X$. Updating this variable is complicated by two facts: First, a process may "terminate early" when it finds a clean node whose previous value is less than the process's value. In this case, the set associated with each ancestor must also be updated. Second, a process may "help" propagate other processes' information up the tree during its summarize phase, specifically, by a successful SC at line 8. To handle this case, we introduce another auxiliary variable $\mathit{set}_p$ for a process $p@\{7, 8\}$ to record to processes in the nodes that $p$ has read when $p$ read them.

Note that the value of $\mathit{set}_p$ is relevant only if $p$ contends for $X_p$ (i.e., $p@\{7, 8\}$ and there has been no successful SC on $X_p$ since $p$'s most recent LL). Also, a process $q$ can be in $\mathit{set}_p$ for such $p$ only if $p$ has read the child of $X_p$ associated with $q$ since its most recent LL. We use $\mathit{contend}(q, k)$ to denote the set of such processes (i.e., $\mathit{contend}(q, k) = \{p : p \text{ contends for } O_q[k] \land O_q[k+1] \notin \mathit{unchecked}_q\}$).

We now define $X.\mathit{set}$ and $\mathit{set}_p$ precisely. These sets are all empty in the initial state, and they are updated as follows:

- The last step of $p$'s Prop($k$) phase adds $p$ to $O_p[k].\mathit{set}$.
- If Prop($k$) is $p$'s final Prop phase (i.e., before it "turns around" and begins its first Cln phase) then its last step also adds $p$ to $O_p[j].\mathit{set}$ for all $j < k$ (i.e., all the ancestors of $O_p[k]$) and to $\mathit{set}_q$ for all $q \in \mathit{contend}(p, j)$ for $j < k$ (i.e., all the contenders for ancestors of $O_p[k]$ that have already read the child associated with $p$).
- If the last step of $p$'s Sum($k$) phase is an LL of $O_p[k]$, finding it dirty (i.e., SUMMARIZE returns from line 6), then it removes $p$ from $O_p[k].\mathit{set}$.
- If the last step of $p$'s Sum($k$) phase is a successful SC at line 8, then it changes $O_p[k].\mathit{set}$ to $\mathit{set}_p$.
- If Sum($k$) is $p$'s final Sum phase (i.e., the last step of DEPART) then its last step also removes $p$ from $O_p[j].\mathit{set}$ for all $j < k$ (i.e., all the ancestors of $O_p[k]$) and from $\mathit{set}_q$ for all $q \in \mathit{contend}(p, j)$ for $j < k$ (i.e., all the contenders for ancestors of $O_p[k]$

that have already read the child associated with $p$).

- An LL at line 6 that makes $p$ a contender initializes $set_p$ to $\emptyset$.
- A read by $p$ at line 7 of node $X$ (a child of the node $p$ is contending for) adds the processes in $X.set$ to $set_p$.

The invariant above is then:

**Invariant 1.** $X.val = \min(\{val_q : q \in X.set\})$ *for all* $X$*. If* $p@\{7, 8\}$ *and then* $min_p = \min(\{val_q : q \in set_p\})$.

We prove this invariant using several subsidiary ones. The first says that $p$ says that $p$ is not in the set of any node when it is absent; it is in the set of every ancestor of its dedicated leaf when it is present or in the clean phase of its ARRIVE operation; and when it is in the Prop($k$) phase of its ARRIVE operation or the Sum($k$) phase of its DEPART operation, its arrival or departure has propagated monotonically up the tree at least as far as level $k$. To prove this invariant inductively, we need to strengthen it to cover the sets of processes contending for nodes associated with $p$. To that end, we define the following predicates: $S(p, k) = (p \in O_p[k].set \wedge \forall q \in contend(p, k), p \in set_q)$ and $\overline{S}(p, k) = (p \notin O_p[k].set \wedge \forall q \in contend(p, k), p \notin set_q)$.

**Invariant 2.** *For any process* $p$*:*

- *If* $p$ *is absent then* $\overline{S}(p, i)$ *for all* $i$*.*
- *If* $p@Prop(k)$ *for some* $k$ *then either* $S(p, i)$ *for all* $i$ *or there exists* $k' \leq k$ *such that* $p \notin O_p[k'].set$ *and* $\overline{S}(p, j)$ *for all* $j < k'$ *and* $S(p, i)$ *for all* $i > k'$*.*
- *If* $p@Cln(k)$ *for some* $k$ *or* $p$ *is present then* $S(p, i)$ *for all* $i$*.*
- *If* $p@Sum(k)$ *for some* $k$ *then either* $\overline{S}(p, i)$ *for all* $i$ *or there exists* $k' \leq k$ *such that* $p \in O_p[k'].set$ *and* $S(p, j)$ *for all* $j < k'$ *and* $\overline{S}(p, i)$ *for all* $i > k'$*.*

Another important invariant is that when a node is clean, its value is greater than or equal to the value of any of its ancestors, and furthermore, any ancestor with the same value must also be clean. This invariant allows a process to stop early when it encounters a clean node whose value is less than the process's value. To handle contending processes, we define $V(p, k) = \max(\{O_p[k].val\} \cup \{min_q : q \in contend(p, k)\})$, and to prove it inductively, we bound the values as the propagate up the tree.

**Invariant 3.** *For any process* $p$ *and value* $v$*:*

- *If* $p@Prop(k)$ *then* $V(p, i) \leq val_p$ *for all* $i > k$*.*
- *If* $p@Cln(k)$ *for some* $k$ *then* $V(p, i) \leq val_p$ *for all* $i$*.*
- *If* $p@Cln(k)$ *for some* $k$ *or* $O_p[k] = \langle val_p, \textbf{\textit{false}} \rangle$ *then* $V(p, j) \leq v$ *and* $O_p[j] \neq \langle v, \textbf{\textit{true}} \rangle$ *for all* $j < k$*.*
- *If* $p$ *is present then* $V(p, k) \leq val_p$ *and* $O_p[k] \neq \langle v, \textbf{\textit{true}} \rangle$ *for all* $k$*.*

### B. Lock-freedom

The key to prove lock-freedom is to define a measure of *making progress*. We can think of each function/procedure as an operation. That is, consider a Node object has PROPAGATE, CLEAN and SUMMARIZE as its operations. It is easy to see that each of these operations is lock-free: CLEAN is wait-free (no loops), and PROPAGATE and SUMMARIZE only go around the loop if they do an SC on the node that fails, which can happen only if another operation does a successful SC. Since any operation that does a successful SC terminates immediately afterwards, that means that operations on Nodes are lock-free. Since Arrive does at most $d$ PROPAGATE and $d$ CLEAN, and DEPART does at most $d$ SUMMARIZE, so the higher-level object is also lock-free.

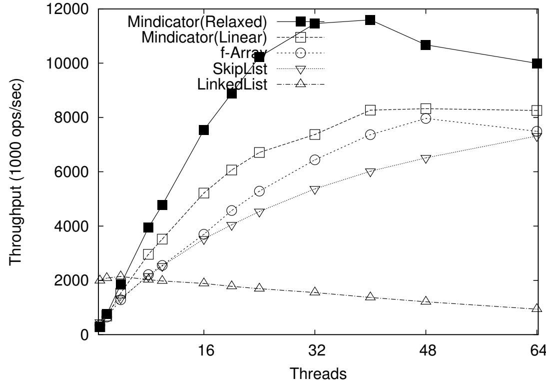
We define $S$, the remaining number of SC operations that is needed to complete an ARRIVE or DEPART, as the measure of progress. The maximum number of SC operations is bounded in each ARRIVE and DEPART operation. (i.e., $S \leq 2d$ for ARRIVE and $S \leq d$ for DEPART). For every $k$ steps of an ARRIVE or DEPART, where $k$ is the maximum number of steps before the operation invokes SC (or decides it won't have to do the SC), $S$ is decreased by 1 for at least one ARRIVE or DEPART operation. Therefore, for $n$ processes, if any ARRIVE or DEPART operation takes $2ndk$ steps, some ARRIVE or DEPART operation must complete in that interval.

## V. EVALUATION

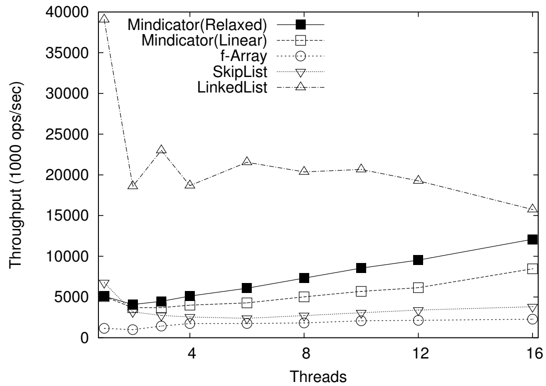
We evaluate mindicators using a microbenchmark and a synthetic TM workload. Experiments labeled "Niagara2" were run on a 1.165 GHz, 64-way Sun UltraSPARC T2 with 32 GB of RAM, running Solaris 10. The Niagara2 has eight cores, each eight-way multithreaded. On the Niagara2, code was compiled using gcc 4.3.2 –O3. Experiments labeled "x86" were run 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 4.7.0 –O3. Results are the average of 5 trials. Our tests run in a 32-bit environment. To simulate LL/SC, we append a counter to each node, and use 64-bit compare-and-swap (CAS) instructions. The counter uses 31 bits.

We compare the following implementations:

*Mindicators:* We configured our mindicators to have 64 leaves. We tested various tree structures: the tree nodes may have an out-degree of 2, 4 or 8 (corresponding to a depth of 7, 4 and 2 respectively).




(a) Niagara2




(b) x86

Figure 2: Stress Test Microbenchmark

*f-Array:* This implementation was achieved by specializing the f-Array object so that the function $f$ at each node computes the minimum of its children's values.

*Skip List:* An implementation using a lock-free skip list [6]. ARRIVE and DEPART are implemented by insert and remove operations on the skip list, and QUERY returns the value of the first element of the bottom level list.

*Linked List:* As a baseline, we use a sorted doubly-linked list. The list is protected by a single coarse-grained lock to minimize latency. Using the list, ARRIVE takes linear time. Since the list has back-edges, we save a pointer to the inserted node, and DEPART costs $O(1)$. We save a snapshot of the minimum value separate from the head pointer, so that an $O(1)$ query can read the snapshot without acquiring the lock.

### A. Stress Test Microbenchmark

We stress test the mindicator through a 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 operations. Furthermore, since there is no program code apart from ARRIVE and DEPART, this over-emphasizes implementation artifacts of CAS. In particular, on the x86, the CAS implementation is unfair, and a single thread will rapidly execute many successive ARRIVE and DEPART operations. This results in an inflated throughput curve for the List in Figure 2(b). On the Niagara2 (Figure 2(a)), the simpler CAS implementation results in a fundamentally different behavior: the list hardly scales, and despite higher latency the mindicator scales to a throughput $13\times$ that of the list. Both charts also demonstrate the cost of linearizability, with the relaxed mindicator scaling best.

While the f-array is also implemented as a tree, its wait-free property impedes scalability relative to the lock-free mindicator. The f-array hardly scales on x86, while on Niagara2 it scales worse than our mindicators. There are two causes: First, write contention on the root of the f-array causes frequent cache invalidations, which incur a steep penalty on the deep cache hierarchy of the x86. Second, in f-array, both ARRIVE and DEPART must read every child of every node while traversing from leaf to root, whereas mindicators only read extra children during DEPART. This difference results in a larger working set and higher cache miss rate for the f-array.

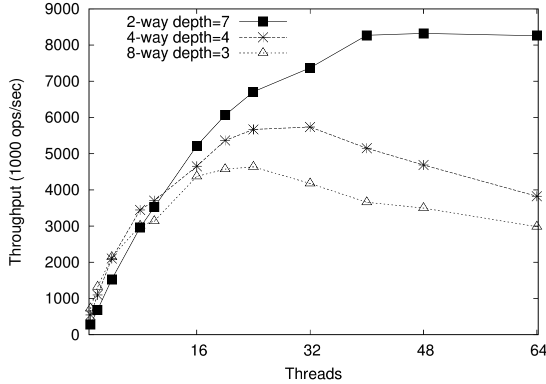
The skip list exhibits poor scalability and high overhead on both platforms. This is due to the maintenance of the dynamic data structure: repeated allocation/deallocation[1] of nodes creates much larger memory footprints, and thus, degrades locality. On the other hand, the size of the skip list tends to be small (i.e. less than 64), and causes frequent cache misses as the linked structure frequently changes.

In further experiments, the addition of a QUERY thread had little impact on the Niagara2 result, but decreased the performance gap between the list and mindicators on x86, by decreasing the performance of the baseline list. This was despite the baseline list's non-blocking QUERY operation: a single thread repeatedly issuing QUERY operations creates coherence traffic that slows down other cores. In contrast, the performance of mindicators are barely impacted by extra QUERY threads added to the workload.

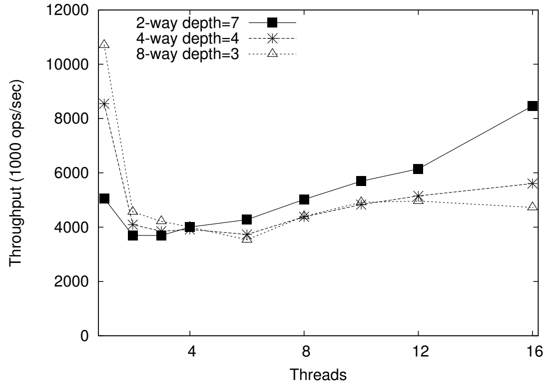
### B. Varying the Mindicator Structures

The tree structure of a mindicator can be varied to optimize for the workload and architecture. Mindicator trees need not be symmetric or balanced, and can be shaped to match a NUMA architecture. In Figure 3, we adjusted the out-degree of nodes from 2 to 8,

[1]Per-thread allocators are used to avoid bottleneck.

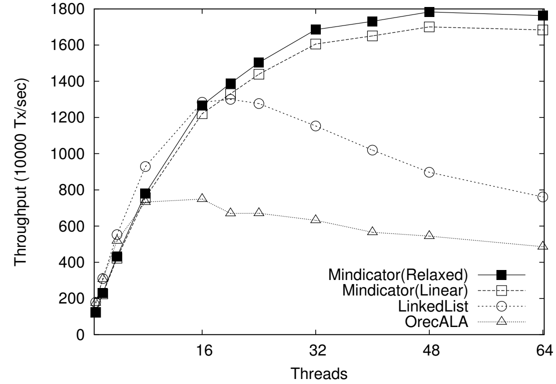



(a) Niagara2

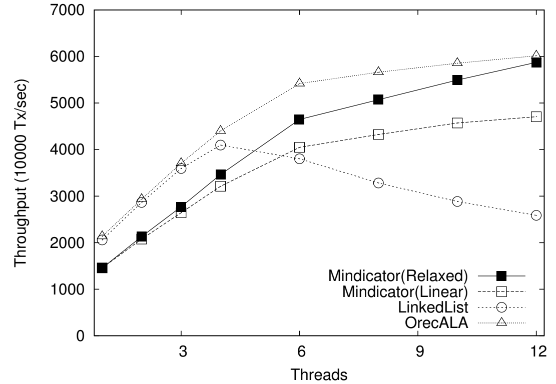



(b) x86

Figure 3: Microbenchmark on Various Tree Structures




(a) Niagara2




(b) x86

Figure 4: STM RBTree Benchmark

making a 64-leaf mindicator more shallow. As depth decreases, there is less overhead at low thread counts, primarily due to fewer steps in the ARRIVE and DEPART operations. However, at higher thread counts, deeper trees with a fewer children per node scale better, since the total number of locations accessed by DEPART decreases.

### C. *Synthetic TM Workload*

While the microbenchmark demonstrates the ability of the mindicator to scale, it is artificial, since the mindicator comprises the entire working set of the experiment. To test a more realistic setting, we employed a mindicator as the quiescence mechanism for the "ALA" TM algorithm from Menon et al. [15]. We compare it to the RSTM [16] equivalent of the Menon technique, "OrecALA", which which employs a pair of counters for the same purpose [4], [17]. Variants of the algorithm appear in Figure 4, with labels derived from whether a list, a relaxed mindicator, or a linearizable mindicator is used. 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. These transactions are small enough that mindicator overheads matter, but large enough that the main overhead is the TM, not the mindicator.

Again, we find that the relaxed algorithm provides excellent scalability, but the gap between relaxed 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 algorithms continue to scale well beyond the point at which OrecALA peaks. On the x86, performance is comparable. 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.

## VI. Conclusions and Future Work

This paper introduced the mindicator, a concurrent data structure that provides a scalable query operation which returns the minimum of values proposed by processes. Our mindicator implementations are lock-free and scalable.

Our future work is in three directions: First, we seek to develop a better understanding of workloads for which the relaxed mindicator is sufficient, and to develop low-overhead mechanisms for rebalancing mindicators. Second, our experiments reveal that 64-bit CAS operations are expensive on the x86, and we are exploring alternatives that might reduce overhead (one particularly appealing option is to accelerate mindicators with hardware TM [3]). Finally, we are investigating additional uses of the mindicator within and beyond TM. In this regard, mindicators can implement a variety of commutative functions, opening up the possibility of wide applicability in applications as well as run-time systems.

## References


[1] Y. Afek, D. Dauber, and D. Touitou. Wait-free made fast. In *Proceedings of the Twenty-seventh Annual ACM Symposium on Theory of Computing*, Las Vegas, Nevada, United States, 1995.

[2] J. Aspnes, H. Attiya, and K. Censor. Max registers, counters, and monotone circuits. In *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing*, Calgary, AB, Canada, 2009.

[3] 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.

[4] D. Dice and N. Shavit. TLRW: Return of the Read-Write Lock. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, Santorini, Greece, June 2010.

[5] 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.

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

[7] 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.

[8] 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.

[9] 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.

[10] P. Jayanti. f-arrays: Implementation and applications. In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing*, Monterey, California, July 2002.

[11] 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.

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

[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.

[19] H. Sundell and P. Tsigas. Fast and Lock-Free Concurrent Priority Queues for Multi-Thread Systems. *Journal of Parallel and Distributed Computing*, 65:609–627, May 2005.