Brief Announcement: A Nonblocking Set Optimized for Querying the Minimum Value

Yujie Liu and Michael Spear
Department of Computer Science and Engineering, Lehigh University
yul510@cse.lehigh.edu, spear@cse.lehigh.edu

Categories and Subject Descriptors

D.1.3 [**Parallel Techniques**]: Concurrent Programming—*Parallel Programming*

General Terms

algorithms, experimentation, measurement, performance

Keywords

shared memory, lock-free data structures, linearizability

1. INTRODUCTION

Shared memory run-time systems, such as garbage collectors (GC) and transactional memory (TM) [2], 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.

Among other examples, this characterization applies to recent problems encountered by Marathe et al. [5] and Koskinen et al. [4], where S is the set of possible start times for transactions in a TM system.

Our solution is inspired by the SNZI shared object [1]. 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 sharing 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 $1 <_t 0$. In this setting, the logical act of incrementing the counter is equivalent to moving a thread to state 1, decrementing the counter is equivalent to moving a thread to state 0, and a SNZI query returns 1 if there exists any thread in 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 register and deregister a thread's state, and $O(1)$ time to query the minimum over all thread states. The Mindicator is scalable and admits lock-free variants that are either linearizable [3] or quiesciently consistent.

2. ALGORITHM

Our lock-free Mindicator implementation is organized as a tree. In this tree, each thread is assigned a unique leaf node, and begins its `Arrive()` and `Depart()` operations at that node. `Arrive()` and `Depart()` operations propagate upward, transmitting values from children to their parents, until they reach a "turning point", after which the operations traverse downward to the leaf at which they originated. `Query()` operations access only the root node. Each leaf is read by several threads, but only modified by one thread. Mindicator nodes can have any number of children.

Figure 1 presents the lock-free Mindicator algorithm; $\infty =$ *int_max* is the default state, which is largest according to the $<_t$ relation. `Depart()` operations always reset a thread state to this value. 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. In this simple variant, the set stored in the Mindicator can be found by reading the values of all leaf nodes. Intermediate nodes store the minimum values of their respective subtrees. The root stores the minimum value of the entire set.

The Mindicator tree is composed of Node objects, where each node contains a collection of child pointers, a parent pointer, and a 64-bit CAS object. The CAS object stores a tuple, consisting of an integer summary value (`min`), a state bit (`sta`), and a version 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 steady value of ∞.

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

```
datatype NODE
  sta   : $\mathbb{B}$   ▷ state bit
  min   : $\mathbb{N}$   ▷ minimum of children
  ver   : $\mathbb{N}$   ▷ version number

initially NODE $(sta, min, ver) = (steady, \infty, 0)$

procedure $Arrive(X : \mathbb{NODE}, n : \mathbb{N})$
 1: while true do
 2:    $x \leftarrow Read(X)$
 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:    $x \leftarrow Read(X)$
 9:    if $x.min \leq n$
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 $x.min = n$
17:       $CAS(X, x, (steady, n, x.ver + 1))$

procedure $Depart(X : \mathbb{NODE}, n : \mathbb{N})$
18: $Revisit(X)$
19: $x \leftarrow Read(X)$
20: if $x.min < n$ 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:    $min \leftarrow \infty$
26:    if $x.sta = tentative$
27:       return
28:    for $C$ in $childrenof(X)$ do
29:       $c \leftarrow Read(C)$
30:       if $c.min < min$
31:          $min \leftarrow c.min$
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 $Query(X : \mathbb{NODE}) : \mathbb{N}$
37: return $Read(X).min$
```

Figure 1: The Lock-Free Mindicator Algorithm: `Arrive()` and `Depart()` should be initiated on leaves, while `Query()` should be executed on the root node.

`Arrive()` inserts a value into the set. It recursively climbs upward, starting at a leaf and ending at some "turning point." Then it regresses downward to the leaf at which it began. When climbing, the thread writes its value at nodes to lower their summary value, and marks those values as tentative. The turning point occurs either when the root is accessed, or when `Arrive()` reaches a node whose value is steady and $\leq$ the arriver's value. The thread then regresses, un-marking the tentative bits it marked in its climbing phase.

A `Depart()` begins at a leaf. It sets the leaf's `min` to the maximum value (e.g., ∞), and recurses upward. At each level, `Depart()` 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 the removal of v until it reaches either the root, or some intermediate node that holds a `steady` value $v' \leq v$.

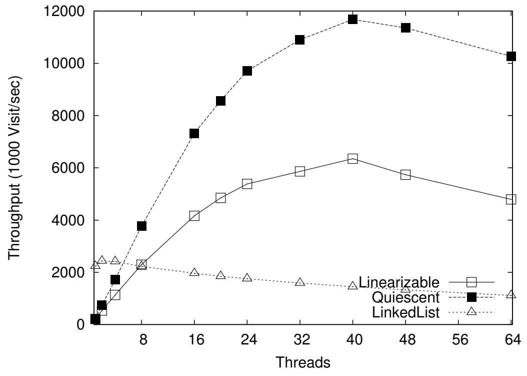

Figure 2: Preliminary Mindicator Performance.

3. EVALUATION AND CONCLUSIONS

We have implemented Mindicators that are lock-based, quiesciently consistent (QC), and lock-free (described above). We have also devised implementations that do not require `Arrive()` to begin at a leaf, and dynamically-resizing Mindicators. A proof of correctness is underway, and we have stress-tested our implementations extensively.

Figure 2 presents performance on a 64-thread Sun Niagara2 CPU. To maximize costs, all threads repeatedly `Arrive()` and `Depart()` from the Mindicator, with no Queries. `Arrive()` uses a random 10-bit value, instead of the monotonically increasing values expected in TM and GC workloads. Throughput is the average of five 5-second trials. We compare a locked doubly-linked list, a lock-free linearizable Mindicator, and a QC Mindicator. For all but the smallest thread counts, Mindicators outperform the list, and the QC algorithm, which is suitable for our target TM workloads, scales very well. We anticipate broad applicability of Mindicators to shared memory runtime systems, middleware, and operating systems.

4. REFERENCES

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.