A Lock-Free, Array-Based Priority Queue *

Yujie Liu and Michael Spear
Lehigh University
{yul510, spear}@cse.lehigh.edu

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel Programming

General Terms Algorithms, Design, Performance

Keywords Lock-Free, Linearizability, Randomization, Heap, Priority Queue, Synchronization, Mound

1. Introduction

Priority queues are useful in scheduling, discrete event simulation, networking (e.g., routing and real-time bandwidth management), graph algorithms (e.g., Dijkstra's algorithm), and artificial intelligence (e.g., A^* search). In these and other applications, not only is it crucial for priority queues to have low latency, but they must also offer good scalability and guarantee progress. Furthermore, the `insert` and `extractMin` operations are expected to have no worse than $O(log(N))$ complexity. In practice, this has focused implementation on heaps [1, Ch. 6] [4] and skip lists [6].

This paper introduces a new lock-free, linearizable [3] priority queue, called the mound. A mound is a tree of sorted lists. Mounds employ randomization when choosing a starting leaf for an `insert`, which avoids the need for insertions to contend for a mound-wide counter, but introduces the possibility that a mound will have "empty" nodes in non-leaf positions. The use of sorted lists avoids the need to swap a leaf into the root position during `extractMin`. Combined with the use of randomization, this ensures disjoint-access parallelism. Asymptotically, `extractMin` is $O(log(N))$. The sorted list also obviates the use of swapping to propagate a new value to its final destination in the mound `insert` operation. Instead, `insert` uses a binary search along a path in the tree to identify an insertion point, and then uses a single writing operation to insert a value. The `insert` complexity is $O(log(log(N)))$. Our lock-free mound employs a software `DCAS` [2] to implement multiword atomic operations.

2. The Mound Algorithm

We focus on the operations needed to implement a lock-free priority queue with a mound, namely `extractMin` and `insert`. We permit the mound to store arbitrary non-unique, totally-ordered values. We reserve $\top$ as the return value of an `extractMin` on an empty mound.

As is common when building lock-free algorithms, we require that every shared memory location be read via a single atomic `READ` operation, which stores its result in a local variable. All updates of shared memory are performed using `CAS`, `DCAS`, or `DCSS` [2]. Furthermore, every mutable shared location is augmented with a counter (c). The counter is incremented on every update, and is read atomically as part of the `READ` operation.

A mound is a rooted tree of sorted lists. The notation $\mathtt{val}(n)$ denotes the value of the first element in the list stored at node n (namely $n.list$). If $n.list$ is empty, $\mathtt{val}(n)$ returns $\top$.

In a traditional min-heap, the heap invariant only holds at the boundaries of functions, ensuring that the value of each node is no greater than the value of any child. This property is also the correctness property for a mound when there are no in-progress operations. When an operation is between its invocation and response, we employ a $dirty$ field to express this "mound property": for every node c and its parent p, $(\neg p.dirty) \Rightarrow \mathtt{val}(p) \leq \mathtt{val}(c)$.

When inserting a value v into the mound, the only requirement is that there exist some node c such that $\mathtt{val}(c) \geq v$ and if c is not the root, for the parent p of c, $\mathtt{val}(p) \leq v$. When such a node is identified, v can be inserted as the new head of c's list. Inserting v as the head of c's list clearly cannot violate the mound property: decreasing $\mathtt{val}(c)$ to v does not violate the mound property between p and c, since $v \geq \mathtt{val}(p)$. Furthermore, for every child c' of c, it already holds that $\mathtt{val}(c') \geq \mathtt{val}(c)$. Since $v \leq \mathtt{val}(c)$, setting $\mathtt{val}(c)$ to v does not violate the mound property between c and its children.

The $\mathtt{insert}(v)$ method operates as follows: it selects a random leaf l and compares v to $\mathtt{val}(l)$. If $v \leq \mathtt{val}(l)$, then either the parent of l has a $\mathtt{val}()$ less than v, in which case the insertion can occur at l, or else there must exist some ancestor c such that inserting v at $c.list$ preserves the mound property. A binary search is employed to find this ancestor. Note that the binary search is along an ancestor chain of logarithmic depth, and thus the search introduces $O(log(log(N))$ overhead. The leaf is ignored if $\mathtt{val}(l) < v$, since the mound property guarantees that every ancestor a of l must have a $\mathtt{val}(a) < v$, and another leaf is randomly selected. If too many unsuitable leaves are selected (bounded by *THRESHOLD*), the mound is expanded by one level. After expansion, every leaf l is guaranteed to be available ($\mathtt{val}(l) = \top > v$), and thus any random leaf is a suitable starting point for the binary search.

`extractMin` is similar to its analog in traditional heaps. When the minimum value is extracted from the root, we return (and remove) the first element of the root's list as the result, or $\top$ if the list is empty. This behavior is equivalent to the traditional heap behavior of moving some leaf node's value into the root. At this point, the mound property may not be preserved between the root and its children, so the root's $dirty$ field is set true.

`moundify` restores the mound property throughout the tree. When `moundify` is called on a node n, it first ensures the children

* This research was sponsored in part by National Science Foundation Grant CNS-1016828

PPoPP'12, February 25–29, 2012, New Orleans, Louisiana, USA.
ACM 978-1-4503-1160-1/12/02.

Listing 1 The Lock-free Mound Algorithm

```
type LNode
  T           value     ▷ value stored in this list node
  LNode*      next      ▷ next element in list

type CMNode
  LNode*      list      ▷ sorted list of values stored at this node
  boolean     dirty     ▷ true if mound property does not hold
  int         c         ▷ counter – incremented on every update

global variables
  tree_{i∈[1,N]} ← ⟨nil, false, 0⟩   : CMNode   ▷ array of mound nodes
  depth ← 1                          : ℕ        ▷ depth of the mound tree

func val(N : CMNode) : T
  if N.list = nil return ⊤ else return N.list.value

proc insert(v : T)
  while true
    c ← findInsertPoint(v)
    C ← READ(tree_c)
    if val(C) ≥ v
      C′ ← ⟨new LNode(v, C.list), C.dirty, C.c + 1⟩
      if c = 1
        if CAS(tree_c, C, C′) return
      else
        P ← READ(tree_{c/2})
        if val(P) ≤ v
          if DCSS(tree_c, C, C′, tree_{c/2}, P) return
      delete(C′.list)

func findInsertPoint(v : ℕ) : ℕ
  while true
    d ← READ(depth)
    for attempts ← 1 . . . THRESHOLD
      leaf ← randLeaf(d)
      if val(leaf) ≥ v return binarySearch(leaf, 1, v)
    CAS(depth, d, d + 1)

func randLeaf(d : ℕ) : ℕ
  return random i ∈ [2^{d−1}, 2^d − 1]

func extractMin() : T
  while true
    R ← READ(tree_1)
    if R.dirty
      moundify(1)
      continue
    if R.list = nil return ⊤
    if CAS(tree_1, R, ⟨R.list.next, true, R.c + 1⟩)
      retval ← R.list.value
      delete(R.list)
      moundify(1)
      return retval

proc moundify(n : ℕ)
  while true
    N ← READ(tree_n)
    d ← READ(depth)
    if ¬N.dirty return
    if n ∈ [2^{d−1}, 2^d − 1] return
    L ← READ(tree_{2n})
    R ← READ(tree_{2n+1})
    if L.dirty
      moundify(2n)
      continue
    if R.dirty
      moundify(2n + 1)
      continue
    if val(L) ≤ val(R) and val(L) < val(N)
      if DCAS(tree_n, N, ⟨L.list, false, N.c + 1⟩,
              tree_{2n}, L, ⟨N.list, true, L.c + 1⟩)
        moundify(2n)
        return
    elif val(R) < val(L) and val(R) < val(N)
      if DCAS(tree_n, N, ⟨R.list, false, N.c + 1⟩,
              tree_{2n+1}, R, ⟨N.list, true, R.c + 1⟩)
        moundify(2n + 1)
        return
    else ▷ Solve problem locally
      if CAS(tree_n, N, ⟨N.list, false, N.c + 1⟩) return
```

of n have $dirty$ set to false, by recursively invoking `moundify` on any $dirty$ children. `moundify` then inspects the `val`() of n and each child, and determines which is smallest. If n has the smallest value, or if n is a leaf, then the mound property already holds, and the operation completes. Otherwise, swapping n with the child having the smallest `val`() restores the mound property at n. However, the child involved in the swap now may not satisfy the mound property with its children, and thus its $dirty$ field is set true. Thus just as in a traditional heap, $O(log(N))$ calls suffice to restore the mound property.

3. Discussion

In a companion technical report, we present sequential and a fine-grained locking based mound algorithms [5]. In our evaluation, we found mound performance to exceed that of the lock-based Hunt priority queue, and to rival that of skiplist-based priority queues.

We also identified nontraditional uses for mounds. The first, probabilistic `extractMin`, is also available in a heap: since any $CMNode$ that is not dirty is, itself, the root of a mound, `extractMin` can be executed on any such node to select a random element from the priority queue. By selecting with some probability shallow, nonempty, non-root $CMNodes$, `extractMin` can lower contention by probabilistically guaranteeing the result to be close to the minimum value. Secondly, it is possible to execute an `extractMany`, which returns several elements from the mound. In the common case, most $CMNode$s in the mound will be expected to hold lists with a modest number of elements. Rather than remove a single element, `extractMany` returns the entire list from a node, by setting the $list$ pointer to **nil** and $dirty$ to true, and then calling `moundify`. This technique can be used to implement lock-free prioritized work stealing.

References

[1] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms, 2nd edition*. MIT Press and McGraw-Hill Book Company, 2001.

[2] T. Harris, K. Fraser, and I. Pratt. A Practical Multi-word Compare-and-Swap Operation. In *Proceedings of the 16th International Conference on Distributed Computing*, Toulouse, France, Oct. 2002.

[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] G. Hunt, M. Michael, S. Parthasarathy, and M. Scott. An Efficient Algorithm for Concurrent Priority Queue Heaps. *Information Processing Letters*, 60:151–157, Nov. 1996.

[5] Y. Liu and M. Spear. A Lock-Free, Array-Based Priority Queue. Technical Report LU-CSE-11-004, Lehigh University, 2011.

[6] I. Lotan and N. Shavit. Skiplist-Based Concurrent Priority Queues. In *Proceedings of the 14th International Parallel and Distributed Processing Symposium*, Cancun, Mexico, May 2000.