2012 41st International Conference on Parallel Processing

Mounds: Array-Based Concurrent Priority Queues

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

***Abstract*—This paper introduces a concurrent data structure called the mound. The mound is a rooted tree of sorted lists that relies on randomization for balance. It supports O(log(log(N))) insert and O(log(N)) extractMin operations, making it suitable for use as a priority queue. We present two mound algorithms: the first achieves lock freedom via the use of a pure-software double-compare-and-swap (DCAS), and the second uses fine grained locks. Mounds perform well in practice, and support novel operations that we expect to be useful in parallel applications, such as extractMany and probabilistic extractMin.**

***Keywords*-Lock-Freedom; Linearizability; Randomization; Heap; Priority Queue; Synchronization**

I. 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, and thus most priority queue implementations are based on heaps [1, Ch. 6] or skip lists [2].

Concurent data structures are commonly evaluated in three categories: progress, correctness, and scalability. Progress refers to the ability of one thread to complete an operation when another thread is in the middle of its operation. A practical progress guarantee is lock freedom, which ensures that once a thread begins an operation on a shared data structure, *some thread* completes its operation in a bounded number of instructions. While individual operations may starve, a lock-free data structure is immune to priority inversion, deadlock, livelock, and convoying. Lock-freedom has been achieved in many high-performance data structures [3], [4].

The most intuitive and useful correctness criteria for concurrent data structures is linearizability [5], which requires that an operation appears to happen at a single instant between its invocation and response. The weaker property of quiescent consistency [6, Ch. 3] still requires each operation to appear to happen at a single instant, but only insists that an operation appear to happen between its invocation and response *in the absence of concurrency*.

Lastly, the most scalable data structures typically exhibit disjoint-access parallelism [7]. That is, their operations do not have overlapping memory accesses unless such accesses are unavoidable. This property captures the intuition that unnecessary sharing, especially write sharing, can result in scalability bottlenecks. In practice, many algorithms with artificial bottlenecks scale well up to some number of hardware threads. However, algorithms that are not disjoint-access parallel tend to exhibit surprisingly bad behavior when run on unexpected architectures, such as those with multiple chips or a large number of threads.

To date, efforts to create a priority queue that is lock-free, linearizable, and disjoint-access parallel have met with limited success. The Hunt heap [8] used fine-grained locking, and avoided deadlock by repeatedly un-locking and re-locking in `insert` to guarantee a global locking order. Dragicevic and Bauer presented a linearizable heap-based priority queue that used lock-free software transactional memory (STM) [9]. Their algorithm improved performance by splitting critical sections into small atomic regions, but the overhead of STM resulted in unacceptable performance. A quiescently consistent, skiplist-based priority queue was first proposed by Lotan and Shavit [10] using fine-grained locking, and was later made non-blocking [4]. Another skiplist-based priority queue was proposed by Sundell and Tsigas [11]. While this implementation was lock-free and linearizabile, it required reference counting, which compromises disjoint-access parallelism.

This paper introduces the "mound", a tree of sorted lists that can be used to construct linearizable, disjoint access parallel priority queues that are either lock-free or lock-based. Like skiplists, mounds achieve balance, and hence asymptotic guarantees, using randomization. However, the structure of the mound tree resembles a heap. The benefits of mounds stem from the following novel aspects of their design and implementation:

0190-3918/12 $26.00 © 2012 IEEE
DOI 10.1109/ICPP.2012.42

- While mound operations resemble heap operations, mounds employ randomization when choosing a starting leaf for an `insert`. This 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 improves disjoint-access parallelism. Asymptotically, `extractMin` is $O(log(N))$, with roughly the same overheads as the Hunt heap.
- 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 $O(log(log(N)))$.
- The mound structure enables several novel uses, such as the extraction of multiple high-priority items in a single operation, and extraction of elements that "probably" have high priority.

In Section II, we present an overview of the mound algorithm. Section III discusses the details of the lock-free algorithm. Section IV presents a fine-grained locking mound. Section V briefly discusses novel uses of the mound that reach beyond traditional priority queues. We evaluate our mound implementations on the x86 and SPARC architectures in Section VI, and present conclusions in Section VII.

II. Mound Algorithm Overview

A mound is a rooted tree of sorted lists. For simplicity of presentation, we consider an array-based implementation of a complete binary tree, and assume that the array is always large enough to hold all elements stored in the mound. The array structure allows us to locate a leaf in $O(1)$ time, and also to locate any ancestor of any node in $O(1)$ time. Other implementation alternatives are discussed in Section VI.

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 of type `T`, and $\top$ is the maximum value. We reserve $\top$ as the return value of an `extractMin` on an empty mound, to prevent the operation from blocking.

Listing 1 presents the basic data types for the mound. We define a mound as an array-based binary tree of `MNodes` (*tree*) and a *depth* field. The `MNode` type describes nodes that comprise the mound's tree. Each node consists of a pointer to a list, a boolean field, and a sequence number (unused in the locking algorithm). The list holds values of type `T`, in sorted order. We define the value of a `MNode` based on whether its list is **nil** or not. If the `MNode`'s list is **nil**, then its value is $\top$. Otherwise, the `MNode`'s value is the value stored in the first element of the list, i.e., $list.value$. The `val()` function in is shorthand for this computation.

Listing 1 Simple data types and methods used by a mound of elements of type "T"

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

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

global variables
  $tree_{i\in[1,N]} \leftarrow \langle$nil, false, 0$\rangle$   : MNode     ▷ array of mound nodes
  $depth \leftarrow 1$                      : integer   ▷ depth of the mound tree
```

In a traditional min-heap, the heap invariant only holds at the boundaries of functions, and is stated in terms of the following relationship between the values of parent and child nodes:

$$\forall p, c \in [1, N] : (\lfloor c/2 \rfloor = p) \Rightarrow \mathtt{val}(tree_p) \leq \mathtt{val}(tree_c)$$

Put another way, a child's value is no less than the value of its parent. 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 the *dirty* field to express a more localized mound property:

$$\forall p, c \in [1, N] : (\lfloor c/2 \rfloor = p) \wedge (\neg tree_p.dirty) \Rightarrow \mathtt{val}(tree_p) \leq \mathtt{val}(tree_c)$$

In other words, when *dirty* is not set, a node's value is less than the value of either of its children.

A mound is initialized by setting every element in the tree to $\langle$**nil**, **false**, $0\rangle$. This indicates that every node has an empty *list*, and hence a logical `val()` of $\top$. Since all nodes have the same `val()`, the *dirty* fields can initially be marked false, as the mound property clearly holds for every parent-child pair.

A. The Insert Operation

The `insert` operation is depicted in Figure 1. When inserting a value v into the mound, the only requirement is that there exist some node index c such that $\mathtt{val}(tree_c) \geq v$ and if $c \neq 1$ (c is not the root index), then for the parent index p of c, $\mathtt{val}(tree_p) \leq v$. When such a node is identified, v can be inserted as the new head of $tree_c.list$. Inserting v as

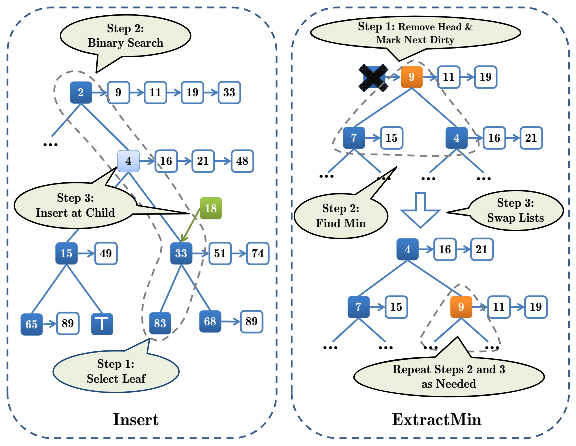


Figure 1: Steps of Insert and ExtractMin Operations

the head of $tree_c.list$ clearly cannot violate the mound property: decreasing $\texttt{val}(tree_c)$ to v does not violate the mound property between $tree_p$ and $tree_c$, since $v \geq \texttt{val}(tree_p)$. Furthermore, for every child index c' of c, it already holds that $\texttt{val}(tree_{c'}) \geq \texttt{val}(tree_c)$. Since $v \leq \texttt{val}(tree_c)$, setting $\texttt{val}(tree_c)$ to v does not violate the mound property between $tree_c$ and its children.

The $\texttt{insert}(v)$ method operates as follows: it selects a random leaf index l and compares v to $\texttt{val}(tree_l)$. If $v \leq \texttt{val}(tree_l)$, then either the parent of $tree_l$ has a $\texttt{val}()$ less than v, in which case the insertion can occur at $tree_l$, or else there must exist some node index c in the set of ancestor indices $\{\lfloor l/2 \rfloor, \lfloor l/4 \rfloor, \ldots, 1\}$, such that inserting v at $tree_c$ preserves the mound property. A binary search is employed to find this index. Note that the binary search is along an ancestor chain of logarithmic length, and thus the search introduces $O(log(log(N))$ overhead.

The leaf is ignored if $\texttt{val}(tree_l) < v$, since the mound property guarantees that every ancestor of $tree_l$ must have a $\texttt{val}() < v$, and another leaf is randomly selected. If too many unsuitable leaves are selected (indicated by a tunable *THRESHOLD* parameter), the mound is expanded by one level.

Note that `insert` is bottleneck-free. Selecting a random leaf avoids the need to maintain a pointer to the next free leaf, which would then need to be updated by every `insert` and `extractMin`. Furthermore, since each node stores a list, we do not need to modify a leaf and then swap its value upward, as in heaps. The number of writes in the operation is $O(1)$.

B. *The ExtractMin Operation*

`extractMin` is depicted in Figure 1, and resembles its analog in traditional heaps. When the minimum value is extracted from the root, the root's `val()` changes to equal the next value in its list, or $\top$ if the list becomes 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.

To restore the mound property at N, a helper function (`moundify`) is used. It analyzes the triangle consisting of a dirty node and its two children. If either child is dirty, it first calls `moundify` on the child, then restarts. When neither child is dirty, `moundify` inspects the $\texttt{val}()s$ of $tree_n$ and its children, and determines which is smallest. If $tree_n$ has the smallest value, or if it is a leaf with no children, then the mound property already holds, and the $tree_n.dirty$ field is set to false. Otherwise, swapping $tree_n$ with the child having the smallest `val()` is guaranteed to restore the mound property at $tree_n$, since $\texttt{val}(tree_n)$ becomes $\leq$ the `val()` of either of its children. 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. In this case, `moundify` is called recursively on the child. Just as in a traditional heap, $O(log(N))$ calls suffice to "push" the violation downward until the mound property is restored.

III. The Lock-Free Mound

An appealing property of mounds is their amenity to a lock-free implementation. In this section, we present a lock-free, linearizable mound that can be implemented on modern x86 and SPARC CPUs. Pseudocode for the lock-free algorithm appears in Listing 2.

A. *Preliminaries*

As is common when building lock-free algorithms, we require that every shared memory location be read via an atomic `READ` instruction. We perform updates to shared memory locations using compare-and-swap (`CAS`), double-compare-and-swap (`DCAS`), and optionally double-compare-single-swap (`DCSS`) operations. We assume that these operations atomically read/modify/write one or two locations, and that they return a boolean indicating if they succeeded. These instructions can be simulated with modest overhead on modern hardware using known techniques [12].

To avoid the ABA problem, every mutable shared location (e.g., each `MNode`) is augmented with a counter (c). The counter is incremented on every

Listing 2 The Lock-free Mound Algorithm

```
func val($N$ : MNode) : T
L1:  if $N.list$ = nil return $\top$
L2:  else return nonFaultingLoad($N.list.value$)

func randLeaf($d$ : integer) : integer
L3:  return $random\ i \in [2^{d-1}, 2^d - 1]$

proc insert($v$ : T)
L4:  while true
L5:      $c \leftarrow$ findInsertPoint($v$)
L6:      $C \leftarrow$ READ($tree_c$)
L7:      if val($C$) $\geq v$
L8:          $C' \leftarrow \langle$new LNode($v, C.list$), $C.dirty, C.c + 1\rangle$
L9:          if $c = 1$
L10:             if CAS($tree_c, C, C'$) return
L11:         else
L12:             $P \leftarrow$ READ($tree_{c/2}$)
L13:             if val($P$) $\leq v$
L14:                 if DCSS($tree_c, C, C', tree_{c/2}, P$) return
L15:         delete($C'.list$)

func findInsertPoint($v$ : T) : integer
L16: while true
L17:     $d \leftarrow$ READ($depth$)
L18:     for $attempts \leftarrow 1 \ldots THRESHOLD$
L19:         $leaf \leftarrow$ randLeaf($d$)
L20:         if val($leaf$) $\geq v$ return binarySearch($leaf, 1, v$)
L21:     CAS($depth, d, d + 1$)

func extractMin() : T
L22: while true
L23:     $R \leftarrow$ READ($tree_1$)
L24:     if $R.dirty$
L25:         moundify(1)
L26:         continue
L27:     if $R.list$ = nil return $\top$
L28:     if CAS($tree_1, R, \langle R.list.next$, true, $R.c + 1\rangle$)
L29:         $retval \leftarrow R.list.value$
L30:         delete($R.list$)
L31:         moundify(1)
L32:         return $retval$

proc moundify($n$ : integer)
L33: while true
L34:     $N \leftarrow$ READ($tree_n$)
L35:     $d \leftarrow$ READ($depth$)
L36:     if $\neg N.dirty$ return
L37:     if $n \in [2^{d-1}, 2^d - 1]$
L38:         if $\neg$CAS($tree_n, N, \langle N.list$, false, $N.c + 1\rangle$)
L39:             continue
L40:     $L \leftarrow$ READ($tree_{2n}$)
L41:     $R \leftarrow$ READ($tree_{2n+1}$)
L42:     if $L.dirty$
L43:         moundify($2n$)
L44:         continue
L45:     if $R.dirty$
L46:         moundify($2n + 1$)
L47:         continue
L48:     if val($L$) $\leq$ val($R$) and val($L$) $<$ val($N$)
L49:         if DCAS($tree_n, N, \langle L.list$, false, $N.c + 1\rangle$,
           $tree_{2n}, L, \langle N.list$, true, $L.c + 1\rangle$)
L50:             moundify($2n$)
L51:             return
L52:     elif val($R$) $<$ val($L$) and val($R$) $<$ val($N$)
L53:         if DCAS($tree_n, N, \langle R.list$, false, $N.c + 1\rangle$,
           $tree_{2n+1}, R, \langle N.list$, true, $R.c + 1\rangle$)
L54:             moundify($2n + 1$)
L55:             return
L56:     else ▷ solve problem locally
L57:         if CAS($tree_n, N, \langle N.list$, false, $N.c + 1\rangle$)
L58:             return
```

CAS/DCAS/DCSS, and is read atomically as part of the READ operation. In practice, this is easily achieved on 32-bit x86 and SPARC architectures. Note that LNodes are immutable, and thus do not require a counter.

We assume that CAS, DCAS, and DCSS do not fail spuriously. We also assume that the implementations of these operations are at least lock-free. Given these assumptions, the lock-free progress guarantee for our algorithm is based on the observation that failure in one thread to make forward progress must be due to another thread making forward progress.

Since MNodes are statically allocated in a mound that never shrinks, the load performed by a READ will not fault. However, if a thread has READ some node $tree_n$ as N, and wishes to dereference $N.list$, the dereference could fault: a concurrent thread could excise and free the head of $tree_n.list$ as part of an extractMin, leading to $N.list$ being invalid. In our pseudocode, we employ a non-faulting load. Garbage collection or object pools would avoid the need for a non-faulting load.

B. Lock-Free Moundify

If no node in a mound is marked $dirty$, then every node satisfies the mound property. In order for $tree_n$ to become $dirty$, either (a) $tree_n$ must be the root, and an extractMin must be performed on it, or else (b) $tree_n$ must be the child of a dirty node, and a moundify operation must swap lists between $tree_n$ and its parent in the process of making the parent's $dirty$ field false.

Since there is no other means for a node to become $dirty$, the algorithm provides a strong property: in a mound subtree rooted at n, if n is not $dirty$, then $\mathtt{val}(tree_n)$ is at least as small as every value stored in every list of every node of the subtree. This in turn leads to the following guarantee: for any node $tree_p$ with children $tree_l$ and $tree_r$, if $tree_p$ is dirty and both $tree_l$ and $tree_r$ are not $dirty$, then executing $\mathtt{moundify}(p)$ will restore the mound property at $tree_p$.

In the lock-free algorithm, this guarantee enables the separation of the extraction of the root's value from the restoration of the mound property, and also enables the restoration of the mound property to be performed independently at each level, rather than through a large atomic section. This, in turn, allows the recursive cleaning moundify of one extractMin to run concurrently with another extractMin.

The lock-free moundify operation retains the obligation to clear any $dirty$ bit that it sets. However, since the operation is performed at one level at a time, it is possible for two operations to reach the same $dirty$ node. Thus, $\mathtt{moundify}(n)$ must be able to *help* clean the *dirty* field of the children of $tree_n$, and must also detect if it has been helped (in which case $tree_n$ will not be *dirty*).

The simplest case is when the operation has been helped. In this case, the READ on line L34 discovers that the parameter is a node that is no longer *dirty*. The next simplest case is when moundify is called on a leaf: a CAS is used to clear the *dirty* bit.

The third and fourth cases are symmetric, and handled on lines L48–L55. In these cases, the children $tree_r$ and $tree_l$ of $tree_n$ are READ and found not to be *dirty*. Furthermore, a swap (by DCAS) is needed between $tree_p$ and one of its children, in order to restore the mound property. Note that a more expensive "triple compare double swap" involving $tree_n$ and both its children is not required. Consider the case where $tree_r$ is not involved in the DCAS: for the DCAS to succeed, $tree_n$ must not have changed since line L34, and thus any modification to $tree_r$ between lines L41 and L49 can only lower $\texttt{val}(tree_r)$ to some value $\geq \texttt{val}(tree_n)$.

In the final case, $tree_n$ is dirty, but neither of its children has a smaller val(). A simple CAS can clear the *dirty* field of $tree_n$. This is correct because, as in the above cases, while the children of $tree_n$ can be selected for insert, the inserted values must remain $\geq \texttt{val}(tree_n)$ or else $tree_n$ would have changed.

C. The Lock-Free ExtractMin Operation

The lock-free extractMin operation begins by reading the root node of the mound. If the node is *dirty*, then there must be an in-flight moundify operation, and it cannot be guaranteed that the val() of the root is the minimum value in the mound. In this case, the operation helps perform moundify, and then restarts.

There are two ways in which extractMin can complete. In the first, the read on line L23 finds that the node's *list* is **nil** and not *dirty*. In this case, at the time when the root was read, the mound was empty, and thus $\top$ is returned. The linearization point is the READ on line L23.

In the second case, extractMin uses CAS to atomically extract the head of the list. The operation can only succeed if the root does not change between the read and the CAS, and it always sets the root to *dirty*. The CAS is the linearization point for the extractMin: at the time of its success, the value extracted was necessarily the minimum value in the mound.

Note that the call to moundify on line L31 is not strictly necessary: extractMin could simply return, leaving the root node *dirty*. A subsequent extractMin would inherit the obligation to restore the mound property before performing its own CAS on the root. Similarly, recursive calls to moundify on lines L50 and L54 could be skipped.

After an extractMin calls moundify on the root, it may need to make several recursive moundify calls at lower levels of the mound. However, once the root is not *dirty*, another extractMin can remove the new minimum value of the root.

D. The Lock-Free Insert Operation

The simplest technique for making insert lock-free is to use a k-Compare-Single-Swap operation (k-CSS), in which the entire set of nodes that are read in the binary search are kept constant during the insertion. However, the correctness of insert depends only on the insertion point $tree_c$ and its parent node $tree_p$.

First, we note that expansion only occurs after several attempts to find a suitable leaf fail: In insert, the randLeaf and findInsertPoint functions read the *depth* field once per set of attempts to find a suitable node, and thus *THRESHOLD* leaves are guaranteed to all be from the same level of the tree, though it may not be the leaf level at any point after line L17. The CAS on line L21 ensures expansion only occurs if the random nodes were, indeed, all leaves.

Furthermore, neither the findInsertPoint nor binarySearch method needs to ensure atomicity among its reads: after a leaf is read and found to be a valid starting point, it may change. In this case, the binary search will return a node that is not a good insertion point. This is indistinguishable from when binary search finds a good node, only to have that node change between its return and the return from findInsertPoint. To handle these cases, insert double-checks node values on lines L6 and L12, and then ensures the node remains unchanged by updating with a CAS or DCSS.

There are two cases for insert: when an insert is performed at the root, and the default case.

First, suppose that v is being inserted into a mound, v is smaller than the root value ($\texttt{val}(tree_1)$), and the root is not *dirty*. In this case, the insertion must occur at the root. Furthermore, any changes to other nodes of the mound do not affect the correctness of the insertion, since they cannot introduce values $< \texttt{val}(tree_1)$. A CAS suffices to atomically add to the root, and serves as the linearization point (line L10). Even if the root is *dirty*, it is acceptable to insert at the root with a CAS, since the insertion does not increase the root's value. The insertion will conflict with any concurrent moundify, but without preventing lock-free progress. Additionally, if the root is dirty and a moundify(1) operation is concurrent, then either inserting v at the root will decrease val($tree_1$) enough that the moundify can use the low-overhead code path on line L57, or else it will be immaterial to the fact that line L49 or L53 is required to swap the root with a child.

This brings us to the default case. Suppose that $tree_c$ is not the root. In this case, $tree_c$ is a valid insertion point if and only if val($tree_c$) $\geq v$, and for $tree_c$'s parent $tree_p$, val($tree_p$) $\leq v$. Thus it does not matter if the insertion is atomic with respect to all of the nodes accessed in the binary search. In fact, both $tree_p$ and $tree_c$ can change after findInsertPoint returns. All that matters is that the insertion is atomic with respect to some READs that support $tree_c$'s selection as the insertion point. This is achieved through READs on lines L6 and L12, and thus the reads performed by findInsertPoint are immaterial to the correctness of the insertion. The DCSS on line L14 suffices to linearize the insert.

Note that the $dirty$ fields of $tree_p$ and $tree_c$ do not affect correctness. Suppose $tree_c$ is $dirty$. Decreasing the value at $tree_c$ does not affect the mound property between $tree_c$ and its children, since the mound property does not apply to nodes that are $dirty$, and cannot affect the mound property between $tree_p$ and $tree_c$, or else findInsertPoint would not return c. Next, suppose $tree_p$ is $dirty$. In this case, for line L14 to be reached, it must hold that val($tree_p$) $\leq v \leq$ val($tree_c$). Thus the mound property holds between $tree_p$ and $tree_c$, and inserting at $tree_c$ will preserve the mound property. The $dirty$ field in $tree_p$ is either due to a propagation of the $dirty$ field that will ultimately be resolved by a simple CAS (e.g., val($tree_p$) is $\leq$ the val() of either of $tree_p$'s children), or else the $dirty$ field will be resolved by swapping $tree_p$ with $tree_c$'s sibling.

IV. A Fine-Grained Locking Mound

We now present a mound based on fine-grained locking. To minimize the number of lock acquisitions, we employ a hand-over-hand locking strategy for restoring the mound property following an extractMin. In this manner, it is no longer necessary to manage the $dirty$ field and sequence counter c in each mound node: unlocked nodes are never dirty. We reuse the $dirty$ field as the lock bit.

We use the same findInsertPoint function as in the lock-free algorithm, and thus allow for an inserting thread to read a node that is locked due to a concurrent insert or extractMin. This necessitates that locations be checked before modification.

The other noteworthy changes to the algorithm deal with how and when locks are acquired and released. Since moundify now uses hand-over-hand locking during a downward traversal of the tree, it always locks the parent before the child. To ensure compatibility, insert must lock parents before children. To avoid cumbersome lock reacquisition, we forgo the optimization for inserting v at a node whose val() $= v$, and also explicitly lock the parent of the insertion point. Similarly, in moundify, we lock a parent and its children before determining the appropriate action. The resulting code appears in Listing 3. Note that the resulting code is both deadlock and livelock-free.

In comparison to the lock-free mound, we expect much lower latency, but without tolerance for preemption. The expectation of lower latency stems from the reduction in the cost of atomic operations: even though we must lock some nodes that would not be modified by CAS in the lock-free algorithm, we are immune to ABA problems and thus only need 32-bit CAS instructions. Furthermore, a critical section corresponding to a DCAS in the lock-free algorithm requires at most three CAS instructions in the locking algorithm. In contrast, lock-free DCAS implementations require 5 CAS instructions [12]. A series of such DCAS instructions offers additional savings, since locks are not released and reacquired: a moundify that would require J DCAS instructions (costing $5J$ CAS instructions) in the lock-free algorithm requires only $2J + 1$ CAS instructions in the locking algorithm.

V. Additional Features of the Mound

Our presentation focused on the use of mounds as the underlying data structure for a priority queue. We now discuss additional uses for the mound.

Probabilistic ExtractMin: Since the mound uses a fixed tree as its underlying data structure, it is amenable to two nontraditional uses. The first, probabilistic extractMin, is also available in a heap: since any MNode 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 MNode$s$ instead of the root, extractMin can lower contention by probabilistically guaranteeing the result to be close to the minimum value.

ExtractMany: It is possible to execute an extractMany, which returns several elements from the mound. In the common case, most MNodes 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

Listing 3 The Fine-Grained Locking Mound Algorithm

```
func setLock($i$ : integer) : MNode
F1:  while true
F2:      $N \leftarrow$ READ($tree_i$)
F3:      if $\neg N.dirty$ and CAS($tree_i, N, \langle N.list, \textbf{true}\rangle$)
F4:          return $N$

func extractMin() : T
F5:  $R \leftarrow$ setLock(1)
F6:  if $R.list =$ nil   ▷ check for empty mound
F7:      $tree_1 = \langle R.list, \textbf{false}\rangle$ ▷ unlock the node
F8:      return $\top$
F9:  $tree_1 \leftarrow \langle R.list.next, \textbf{true}\rangle$ ▷ remove list head, keep node locked
F10: $retval = R.list.value$
F11: delete($R.list$)
F12: moundify(1)
F13: return $retval$

proc moundify($n$ : integer)
F14: while true
F15:     $N \leftarrow$ READ($tree_n$)
F16:     $d \leftarrow depth$
F17:     if $n \in [2^{d-1}, 2^d - 1]$   ▷ Is $n$ a leaf?
F18:         $tree_n \leftarrow \langle tree_n.list, \textbf{false}\rangle$
F19:         return
F20:     $L \leftarrow$ setLock($2n$)
F21:     $R \leftarrow$ setLock($2n+1$)
F22:     if val($L$) $\leq$ val($R$) and val($L$) $<$ val($N$)
F23:         $tree_{2n+1} \leftarrow \langle R.list, \textbf{false}\rangle$ ▷ unlock right child
F24:         $tree_n \leftarrow \langle L.list, \textbf{false}\rangle$ ▷ update and unlock parent
F25:         $tree_{2n} \leftarrow \langle N.list, \textbf{true}\rangle$ ▷ keep left locked after update
F26:         moundify($2n$)
F27:     elif val($R$) $<$ val($L$) and val($R$) $<$ val($N$)
F28:         $tree_{2n} \leftarrow \langle L.list, \textbf{false}\rangle$ ▷ unlock left child
F29:         $tree_n \leftarrow \langle R.list, \textbf{false}\rangle$ ▷ update and unlock parent
F30:         $tree_{2n+1} \leftarrow \langle N.list, \textbf{true}\rangle$ ▷ keep right locked after update
F31:         moundify($2n+1$)
F32:     else ▷ Solve problem locally by unlocking $tree_n$ and its children
F33:         $tree_n \leftarrow \langle N.list, \textbf{false}\rangle$
F34:         $tree_{2n} \leftarrow \langle L.list, \textbf{false}\rangle$
F35:         $tree_{2n+1} \leftarrow \langle R.list, \textbf{false}\rangle$

proc insert($v$ : T)
F36: while true
F37:     $c \leftarrow$ findInsertPoint($v$)
F38:     if $c = 1$   ▷ insert at root?
F39:         $C \leftarrow$ setLock($c$)
F40:         if val($C$) $\geq v$   ▷ double-check node
F41:             $tree_c \leftarrow \langle$new LNode($v, C.list$), false$\rangle$
F42:             return
F43:         $tree_c \leftarrow \langle C.list, \textbf{false}\rangle$ ▷ unlock root and start over
F44:         continue
F45:     $P \leftarrow$ setLock($c/2$)
F46:     $C \leftarrow$ setLock($c$)
F47:     if val($C$) $\geq v$ and val($P$) $\leq v$   ▷ check insertion point
F48:         $tree_c \leftarrow \langle$new LNode($v, C.list$), false$\rangle$
F49:         $tree_{c/2} \leftarrow \langle P.list, \textbf{false}\rangle$
F50:         return
F51:     else ▷ unlock $tree_c$ and $tree_{c/2}$, then try again
F52:         $tree_{c/2} \leftarrow \langle P.list, \textbf{false}\rangle$
F53:         $tree_c \leftarrow \langle C.list, \textbf{false}\rangle$
```

setting the $list$ pointer to **nil** and $dirty$ to true, and then calling `moundify`. This technique can be used to implement prioritized work stealing.

VI. Evaluation

In this section, we evaluate the performance of mounds using targeted microbenchmarks. Experiments labeled “Niagara2” were collected on a 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 Intel Xeon X5650 processor with six cores, each two-way multithreaded, running Linux 2.6.32. The x86 code was compiled using gcc 4.4.3, with –O3 optimizations. On both machines, the largest level of the cache hierarchy is shared among all threads. The Niagara2 cores are substantially simpler than the x86 cores, and have one less level of private cache.

A. Implementation Details

We implemented `DCAS` using a modified version of the technique proposed by Harris et al [12]. The resulting implementation resembles an inlined nonblocking software transactional memory [13]. We chose to implement `DCSS` using a `DCAS`. Rather than using a flat array, we implemented the mound as a 32-element array of arrays, where the n^{th} second-level array holds 2^n elements. We did not pad `MNode` types to a cache line. This implementation ensures minimal space overhead for small mounds, and we believe it to be the most realistic for real-world applications, since it can support extremely large mounds. We set the *THRESHOLD* constant to 8. Changing this value did not affect performance, though we do not claim optimality.

Since the x86 does not offer non-faulting loads, we used a per-thread object pool to recycle `LNode`s without risking their return to the operating system. To enable atomic 64-bit reads on 32-bit x86, we used a lightweight atomic snapshot algorithm, as 64-bit atomic loads can otherwise only be achieved via high-latency floating point instructions.

B. Effects of Randomization

Unlike heaps, mounds do not guarantee balance, instead relying on randomization. To measure the effect of this randomization on overall mound depth, we ran a sequential experiment where 2^{20} `inserts` were performed, followed by $2^{19} + 2^{18}$ `extractMins`. We measured the fullness of every mound level after the insertion phase and during the remove phase. We also measured the fullness whenever the depth of the mound increased. We varied the order of insertions, using either randomly selected keys, keys that always increased, or keys that always decreased. These correspond to the average, worst, and best cases for mound depth. Lastly, we measured the impact of repeated insertions and removals on mound depth, by initializing a mound with 2^8, 2^{16}, or 2^{20} elements, and then performing 2^{20}

Insert Order	% Fullness of Non-Full Levels
Increasing	99.96% (17), 97.75% (18), 76.04% (19), 12.54% (20)
Random	99.99% (16), 96.78% (17), 19.83% (18)

Table I: Incomplete mound levels after 2^{20} insertions. Incompleteness at the largest level is expected.

Initialization	Ops	Non-Full Levels
Increasing	524288	99.9% (16), 94.6% (17), 61.4% (18), 17.6% (19), 1.54% (20)
Increasing	786432	99.9% (15), 93.7% (16), 59.3% (17), 17.6% (18), 2.0% (19), 0.1% (20)
Random	524288	99.7% (16), 83.4% (17), 14.7% (18)
Random	786432	99.7% (15), 87.8% (16), 38.9% (17), 3.6% (18)

Table II: Incomplete mound levels after many `extractMin`s. Mounds were initialized with 2^{20} elements, using the same insertion orders as in Table I.

randomly selected operations (an equal mix of `insert` and `extractMin`).

Table I describes the levels of a mound that have nodes with empty lists after 2^{20} insertions. For all but the last of these levels, incompleteness is a consequence of the use of randomization. Each value inserted was chosen according to one of three policies. When each value is larger than all previous values ("Increasing"), the worst case occurs. Here, every list has exactly one element, and every insertion occurs at a leaf. This leads to a larger depth (20 levels), and to several levels being incomplete. However, note that the mound is still only one level deeper than a corresponding heap would be in order to store as many elements. [1]

When "Random" values are inserted, we see the depth of the mound drop by two levels. This is due to the average list holding more than one element. Only 56K elements were stored in leaves (level 18), and 282K elements were stored in the 17th level, where lists averaged 2 elements. 179K elements were stored in the 16th level, where lists averaged 4 elements. The longest average list (14 elements) was at level 10. The longest list (30) was at level 7. These results suggest that mounds should produce more space-efficient data structures than either heaps or skiplists, and also confirm that randomization is an effective strategy.

We next measured the impact of `extractMin` on the depth of mounds. In Table II, we see that randomization leads to levels remaining partly filled for much longer than in heaps. After 75% of the elements have been removed, the deepest level remains nonempty. Furthermore, we found that the repeated `extractMin` operations decreased the average list size significantly. After 786K removals, the largest list in the mound had only 8 elements.

[1]The other extreme occurs when elements are inserted in decreasing order, where the mound organizes itself as a sorted list at the root.

Initial Size	Incomplete Levels
2^{20}	99.9% (16), 99.4% (17), 74.3% (18)
2^{16}	99.7% (13), 86.1% (14)
2^{8}	95.3% (6), 68.8% (7)

Table III: Incomplete mound levels after 2^{20} random operations, for mounds of varying sizes. Random initialization order was used.

To simulate real-world use, we pre-populated a mound, and executed 2^{20} operations (an equal mix of `insert` and `extractMin`), using randomly selected keys for insertions. The result in Table III shows that this usage does not lead to greater imbalance or to unnecessary mound growth. However, the incidence of removals did reduce the average list size. After the largest experiment, the average list size was only 3.

C. Insert Performance

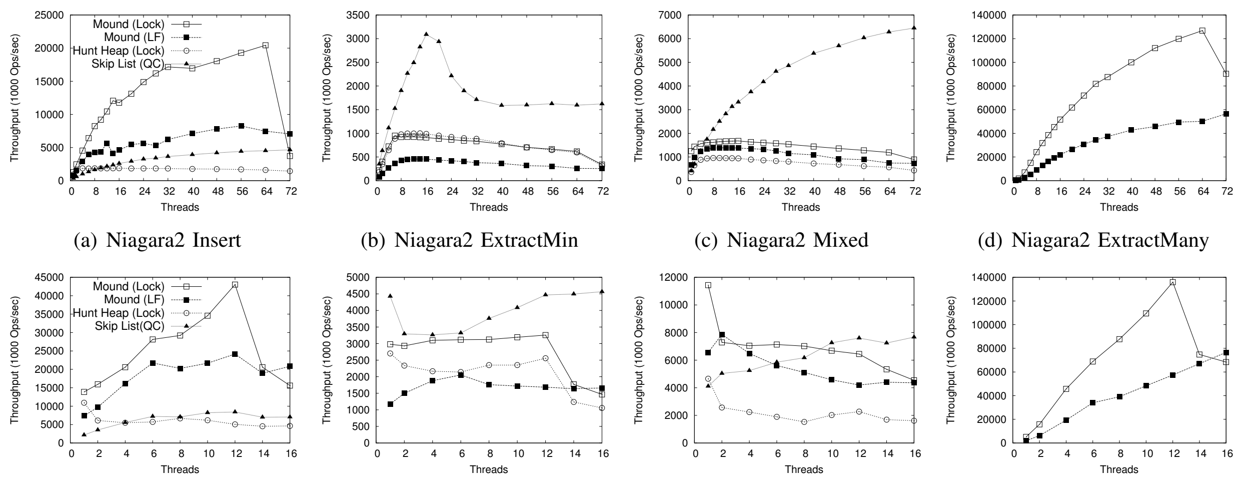
Next, we evaluate the latency and throughput of `insert` operations. As comparison points, we include the Hunt heap [8], which uses fine-grained locking, and a quiescently consistent, skiplist-based priority queue [6], [10, Ch. 3][2]. Each experiment is the average of three trials, and each trial performs a fixed number of operations per thread. We conducted additional experiments with the priority queues initialized to a variety of sizes, ranging from hundreds to millions of entries. We present only the most significant trends.

Figure 2 (a) and (e) present `insert` throughput. The extremely strong performance of the fine-grained locking mound is due both to its asymptotic superiority, and its low-overhead implementation using simple spinlocks. In contrast, while the lock-free mounds scale well, they have much higher latency. On the Niagara2, `CAS` is implemented in the L2 cache; thus there is a hardware bottleneck after 8 threads, and high overhead due to our implementation of `DCAS` with multiple `CAS`es. On the x86, both 64-bit atomic loads and `DCAS` contribute to the increased latency. As previously reported by Lotan and Shavit, insertions are costly for skip lists. The hunt heap has low single-thread overhead, but the need to "trickle up" causes `insert`s to contend with each other, which hinders scalability.

D. ExtractMin Performance

In Figure 2 (b) and (f), each thread performs 2^{16} `extractMin` operations on a priority queue that is pre-populated with exactly enough elements that the last of these operations will leave the data structure

[2]We extended Vincent Gramoli's open-source skiplist.



(a) Niagara2 Insert (b) Niagara2 ExtractMin (c) Niagara2 Mixed (d) Niagara2 ExtractMany

(e) x86 Insert (f) x86 ExtractMin (g) x86 Mixed (h) x86 ExtractMany

Figure 2: In `insert` test (a and e), each thread inserts 2^{16} randomly selected values. In `extractMin` test (b and f), each thread performs 2^{16} `extractMin` operations to make the priority queue empty. In mixed operation test (c and g), equal mix of random `insert` and `extractMin` operations are performed on a queue initialized with 2^{16} random elements. In `extractMany` test (d and h), the mound is initialized with 2^{20} elements, and then threads repeatedly call `extractMany` until the mound is empty.

empty. The skip list implementation is almost perfectly disjoint-access parallel, and thus on the Niagara2, it scales well. To extract the minimum, threads attempt to mark (`CAS`) the first "undeleted" node as "deleted" in the bottom level list, and keeps searching if the marking failed. On successfully marking a node as "deleted", the thread performs a subsequent physical removal of the marked node, which mitigates further contention between operations. On the x86, the deeper cache hierarchy results in a slowdown for the skiplist from 1–6 threads, after which the use of multithreading decreases cache misses and results in slight speedup.

The algorithms of the locking mound and the Hunt queue are similar, and their performance curves match closely. Slight differences on the x86 are largely due to the shallower tree of the mound. However, in both cases performance is substantially worse than for skiplists. As in the `insert` experiment, the lock free mound pays additional overhead due to its use of `DCAS`. Since there are $O(log(N))$ `DCAS`es, instead of the single `DCAS` in `insert`, the overhead of the lock free mound is significantly higher than the locking mound.

E. Scalability of Mixed Workloads

The behavior of a concurrent priority queue is expected to be workload dependent. While it is unlikely that any workload would consist of repeated calls to `insert` and `extractMin` with no work between calls, we present such a stress test microbenchmark in Figure 2 (c) and (g) as a more realistic evaluation than the previous single-operation experiments.

In the mixed workload, we observe the mounds provide better performance at lower thread counts. On the x86, the locking mound provides the best performance until 10 threads, but suffers under preemption. The lock-free mounds outperform skiplists until 6 threads. As in the `extractMin` test, once the point of hardware multithreading is reached, the large number of `CAS`es becomes a significant overhead.

F. ExtractMany Performance

One of the advantages of the mound is that it stores a collection of elements at each tree node. As discussed in Section V, implementing `extractMany` entails only a simple change to the `extractMin` operation. However, its effect is pronounced. As Figure 2 (d) and (h) show, `extractMany` scales well.

This scaling supports our expectation that mounds will be a good fit for applications that employ prioritized or probabilistic work stealing. However, there is a risk that the quality of data in each list is poor. For example, if the second element in the root list is extremely large, then using `extractMany` will not provide a set of high-priority elements. Table IV presents the average list size and average value of elements in a mound after 2^{20} insertions of random values. As desired, extracted

Level	List Size	Avg. Value	Level	List Size	Avg. Value
0	12	52.5M	9	15.46	367M
1	15.5	179M	10	13.81	414M
2	21.75	215M	11	12.33	472M
3	21.75	228M	12	10.57	538M
4	21.18	225M	13	8.80	622M
5	20.78	263M	14	7.22	763M
6	19.53	294M	15	5.47	933M
7	18.98	297M	16	3.67	1.14B
8	17.30	339M	17	2.14	1.45B

Table IV: Average list size and list value of mound nodes after 2^{20} random insertions.

lists are large, and have an average value that increases with tree depth. Similar experiments using values from smaller ranges are even more pronounced.

VII. Conclusions

In this paper we presented the mound, a new data structure for use in concurrent priority queues. The mound combines a number of novel techniques to achieve its performance and progress guarantees. Chief among these are the use of randomization and the employment of a structure based on a tree of sorted lists. Linearizable mounds can be implemented in a highly concurrent manner using either pure-software `DCAS` or fine-grained locking. Their structure also allows several new uses. We believe that prioritized work stealing is particularly interesting.

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. The performance tradeoffs are nuanced, and will certainly depend on workload and architecture. Workloads that can employ `extractMany` or that benefit from fast `insert` will benefit from the mound. The difference in performance between the x86 and Niagara2 suggests that deep cache hierarchies favor mounds.

The lock-free mound is a practical algorithm despite its reliance on software `DCAS`. We believe this makes it an ideal data structure for designers of future hardware. In particular, the question of what new concurrency primitives (such as `DCAS` and `DCSS`, best-effort hardware transactional memory [14], or even unbounded transactional memory) should be added to next-generation architectures will be easier to address given algorithms like the mound, which can serve as microbenchmarks and demonstrate the benefit of faster hardware multiword atomic operations.

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] W. Pugh, "Skip Lists: A Probabilistic Alternative to Balanced Trees," *Communications of the ACM*, vol. 33, pp. 668–676, June 1990.

[3] M. M. Michael and M. L. Scott, "Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms," in *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing*, May 1996.

[4] K. Fraser, "Practical Lock-Freedom," Ph.D. dissertation, King's College, University of Cambridge, Sep. 2003.

[5] M. P. Herlihy and J. M. Wing, "Linearizability: a Correctness Condition for Concurrent Objects," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 3, pp. 463–492, 1990.

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

[7] A. Israeli and L. Rappoport, "Disjoint-Access-Parallel Implementations of Strong Shared Memory Primitives," in *Proceedings of the 13th ACM Symposium on Principles of Distributed Computing*, 1994.

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

[9] K. Dragicevic and D. Bauer, "Optimization Techniques for Concurrent STM-Based Implementations: A Concurrent Binary Heap as a Case Study," in *Proceedings of the 23rd International Symposium on Parallel and Distributed Processing*, Rome, Italy, May 2009.

[10] 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.

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

[12] 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.

[13] T. Harris and K. Fraser, "Language Support for Lightweight Transactions," in *Proceedings of the 18th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Oct. 2003.

[14] D. Dice, Y. Lev, M. Moir, and D. Nussbaum, "Early Experience with a Commercial Hardware Transactional Memory Implementation," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, Washington, DC, Mar. 2009.