A Lock-Free, Array-Based Priority Queue

Yujie Liu and Michael Spear

Department of Computer Science and Engineering
Lehigh University

Technical Report LU-CSE-11-004

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 several novel operations that we expect to be useful in future parallel applications, such as `extractMany`() and probabilistic `extractMin`().

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] and skip lists [18].

The ideal concurrent data structure guarantees that each operation completes in a bounded number of instructions, regardless of the activities of concurrent threads. In general, this progress property, wait freedom, introduces unacceptable overheads (see [12] for an exception). The relaxation that once a thread begins an operation on a shared data structure, *some thread* completes its operation in a bounded number of instructions is called lock freedom. While individual operations may starve, a lock-free data structure is immune to priority inversion, deadlock, livelock, and convoying. Unlike wait freedom, lock freedom has been achieved in many high-performance data structures [4, 5, 16, 17, 21].

The most intuitive and useful correctness criteria for concurrent data structures is linearizability [9], which requires that every operation appears to take effect at a single instant between when it issues its first instruction and when it issues its last instruction. The weaker property of quiescent consistency still requires each operation to appear to happen at a single instant, but only insists that an operation appear to happen between its first and last instruction *in the absence of concurrency*.

Lastly, concurrent data structures should exhibit disjoint-access parallelism [11]. That is, operations should not have overlapping memory accesses unless they are required for the correctness of the algorithm. 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. We summarize prior work below:

Heaps: There are several challenges for heap-based priority queues. First, implementations typically assume strong balance and fullness guarantees; this can result in an implementation bottleneck, as every operation modifies a pointer to the rightmost nonempty leaf. Second, `extractMin`() appears to require top-down tree traversal, while `insert`() must run bottom-up. Third, both `insert`() and `extractMin`() are expressed recursively, and involve swapping values between two nodes. Thus while the asymptotic complexity of both operations is $O(log(N))$, implementations require $2 \times log(N)$ writes. Finally, achieving atomicity appears to require atomic modification of two noncontiguous words of memory, which is beyond the ability of modern hardware.

The most popular concurrent heap, the Hunt heap [10], uses fine-grained locking, and avoids deadlock by repeatedly un-locking and re-locking in `insert`() to guarantee a deadlock-free locking order. Furthermore, while the implementation avoids contention among concurrent operations *after* they have selected their starting points, there is still a shared counter used to guarantee balance. More recently, Dragicevic and Bauer presented a linearizable heap-based priority queue that used lock-free software transactional memory (STM) [3]. Their algorithm improved performance by splitting critical sections into small atomic regions, with an underlying implementation similar to the Hunt heap, but the overhead of STM resulted in unacceptable performance.

Skiplists: The skiplist `insert`() and `removeKey`() operations have few intrinsic bottlenecks and scale well. Among the most popular implementations are the Lotan and Shavit skiplist [13], which is based on fine-grained locking, and the lock-free Fraser skiplist [4]. The key challenge in using skiplists as priority queues is in transforming `removeKey`() into `extractMin`(). Lotan and Shavit present two techniques. The first is quiescently consistent, and allows for `extractMin`() to return a value that is greater than the minimum, if both the minimum and the return value of the `extractMin`() are inserted after the `extractMin`() begins [8, Ch. 15]. The second employs a shared counter, incremented by every `insert`() and `extractMin`(). Both implementations also introduce a worst-case complexity of $O(num\_threads)$ for the `extractMin`() operation. Another skiplist-based priority queue was proposed by Sundell and Tsigas [20]. While this implementation achieves lock-freedom and linearizability without introducing a global counter, it requires reference counting, which compromises disjoint-access parallelism.

This paper introduces a new data structure suited to the construction of priority queues, called the mound. A mound is a tree of sorted lists. Like skiplists, mounds achieves 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:

- 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 ensures disjoint-access parallelism. Asymptotically, `extractMin`() is $O(log(N))$, with roughly the same overheads as the Hunt heap.

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
```

- 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 2, we present a sequential mound algorithm. Section 3 introduces a linearizable lock-free mound based on the double-compare-and-swap (`DCAS`) and double-compare-single-swap (`DCSS`) instructions. Section 4 presents a fine-grained locking mound. Section 5 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 6, and present conclusions in Section 7.

2 The Sequential Mound Algorithm

A mound is a rooted tree of sorted lists. For simplicity of presentation, we use 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. A more practical implementation is discussed in Section 6.

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 values of type T, where $<$ defines a total order on elements of T, and $\top$ is the maximum value. We reserve $\top$ as the return value of an `extractMin`() on an empty mound.

The `MNode` type (Listing 1) describes nodes that comprise the mound's tree. Each node consists of a pointer to a list and a boolean field. The list holds a value and a next pointer. 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 Listing 2 is shorthand for this computation.

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:

$$\forall_{p,c:tree} : (parentof(c) = p) \rightarrow \texttt{val}(p) \leq \texttt{val}(c)$$

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 a $dirty$ field to express this "mound property":

$$\forall_{p,c:tree} : (parentof(c) = p) \wedge (\neg p.dirty) \rightarrow \texttt{val}(p) \leq \texttt{val}(c)$$

Listing 2 presents a sequential mound implementation. We define a mound as an array-based binary tree of `MNodes` ($tree$), and a $depth$ field. A mound is initialized by setting every element in the tree to $\langle \textbf{nil}, \textbf{false} \rangle$. 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.

2.1 The insert Operation

When inserting a value v into the mound, the only requirement is that there exist some node index c such that `val`$(tree_c) \geq v$ and if $c \neq 1$ (c is not the root index), then for the parent index p of c, `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 the head of $tree_c.list$ clearly cannot violate the mound property: decreasing `val`$(tree_c)$ to v does not violate the mound property between $tree_p$ and $tree_c$, since $v \geq$ `val`$(tree_p)$. Furthermore, for every child index c' of c, it already holds that `val`$(tree_{c'}) \geq$ `val`$(tree_c)$. Since $v \leq$ `val`$(tree_c)$, setting `val`$(tree_c)$ to v does not violate the mound property between $tree_c$ and its children.

The `insert`(v) method operates as follows: it selects a random leaf index l and compares v to `val`$(tree_l)$. If $v \leq$ `val`$(tree_l)$, then either the parent of $tree_l$ has a `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 $\{l/2, l/4, \ldots, 1\}$, 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 depth, and thus the search introduces $O(log(log(N))$ overhead.

The leaf is ignored if `val`$(tree_l) < v$, since the mound property guarantees that every ancestor of $tree_l$ must have a `val`$() < v$, and another leaf is randomly selected. If too many unsuitable leaves are selected (indicated by *THRESHOLD*), the mound is expanded by one level. After expansion, every leaf is guaranteed to be available (`val`$() =$ **nil**), and thus any random leaf is a suitable starting point for the binary search.

2.2 The extractMin Operation

`extractMin`()is similar to 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.

`moundify`() restores the mound property throughout the tree. In the sequential case, at most one node in the tree has its $dirty$ field set to true. Thus when `moundify`() is called on a node n, the children of n have $dirty$ set to false, and each child is itself a mound whose minimum value is stored in the head of the child's list. The `moundify`() operation inspects the `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 `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. The same invariants apply to the subsequent recursive call on the child: the parameter of the call is a $dirty$ node, the children of that node are not $dirty$, and each child is, itself, a mound. Thus just as in a traditional heap, $O(log(N))$ calls suffice to "push" the violation downward until the mound property is restored.

Listing 2 The Sequential Mound Algorithm

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

func val($n : MNode$) : $T$
S1:  if $n.list = $ nil return $\top$
S2:  return $n.list.value$

func randLeaf() : $\mathbb{N}$
S3:  return $random\ i \in [2^{depth-1}, 2^{depth} - 1]$

proc insert($v : T$)
S4:  $c \leftarrow$ findInsertPoint($v$)
S5:  $tree_c.list \leftarrow$ new $LNode(v, tree_c.list)$

func findInsertPoint($v : \mathbb{N}$) : $\mathbb{N}$
S6:  for $attempts \leftarrow 1 \ldots THRESHOLD$
S7:     $leaf \leftarrow$ randLeaf()
S8:     if val($tree_{leaf}$) $\geq v$  ▷ Leaf is a valid starting point
S9:        return binarySearch($leaf, 1, val$)
S10: $depth \leftarrow depth + 1$ ▷ Grow mound one level
S11: return binarySearch(randLeaf(), $1, val$)

func extractMin() : $T$
S12: if $tree_1.list = $ nil return $\top$ ▷ Handle empty mound
S13: $oldlist \leftarrow tree_1.list$
S14: $retval \leftarrow tree_1.list.value$
S15: $tree_1 \leftarrow \langle tree_1.list.next,$ true $\rangle$ ▷ Remove head of root's list and mark root dirty
S16: moundify(1)
S17: delete($oldlist$)
S18: return $retval$

proc moundify($n : \mathbb{N}$)
S19: if $n \in [2^{depth-1}, 2^{depth} - 1]$  ▷ check if $n$ is a leaf
S20:    $tree_n.dirty \leftarrow$ false
S21:    return
S22: $l \leftarrow 2n$
S23: $r \leftarrow 2n + 1$
     ▷Check if left or right child has smaller value than $n$
S24: if val($tree_l$) $\leq$ val($tree_r$) and val($tree_l$) $<$ val($tree_n$)
S25:    swap($tree_n, tree_l$) ▷ Propagate to left child
S26:    moundify($l$)
S27: elif val($tree_r$) $<$ val($tree_l$) and val($tree_r$) $<$ val($tree_n$)
S28:    swap($tree_n, tree_r$) ▷ Propagate to right child
S29:    moundify($r$)
S30: else
S31:    $tree_n.dirty \leftarrow$ false ▷ Resolve problem locally
```

3 The Concurrent Mound Algorithm

The sequential mound algorithm avoids global bottlenecks by using a randomized search instead of maintaining a strict count of the number of unused leaves. Furthermore, by using a list of values at each node,

Listing 3 Extended `MNode` type for use in concurrent mounds. All fields are read in a single atomic instruction, and every update increments the counter field (c).

```
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
```

the algorithm avoids the need to repeatedly swap a value upward from a leaf; instead `insert`() need only find a suitable parent/child pair and then insert the value as the new head of the child's list.

We now describe a lock-free mound implementation, which requires compare-and-swap (`CAS`) and double-compare-and-swap (`DCAS`) operations, and which benefits from the availability of a double-compare-single-swap (`DCSS`) operation. 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 acceptable overhead on modern hardware using known techniques [7, 14].

3.1 Preliminaries

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 will be performed using `CAS`, `DCAS`, or `DCSS`. Furthermore, any 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. In practice, this is easily achieved on 32-bit x86 and SPARC architectures. Since `LNodes` are never modified in the sequential mound algorithm, they are not augmented with a 32-bit counter. Listing 3 presents the new concurrent `MNode` type (`CMNODE`).

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 will be based on the observation that failure in one thread to make forward progress must be due to another thread making forward progress. By forbidding "backward" progress, livelock will not be a concern.

Since the CMNodes 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 introduce a new helping method, `val'`(). When given a cached copy of a `CMNode`, `val'`() uses a nonfaulting load to dereference the list pointer. Garbage collection, object pools, or Hazard Pointers [15] would avoid the need for a nonfaulting load.

3.2 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 sequential algorithm provides a strong property: in a mound subtree rooted at n, if n is not $dirty$, then `val`($tree_n$) is at least as small as every

Listing 4 The Lock-Free `moundify`()Operation

```
func val'($N$ : CMNode) : $T$
L1:  if $N.list$ = nil return $\top$
L2:  return $nonfaultingLoad(N.list.value)$

proc moundify($n : \mathbb{N}$)
L3:  while true
L4:     $N \leftarrow$ READ($tree_n$)
L5:     if $\neg N.dirty$  $\triangleright$ Did another thread clean $n$?
L6:        return
L7:     $d \leftarrow depth$
L8:     if $n \in [2^{d-1}, 2^d - 1]$  $\triangleright$ Is $n$ a leaf?
L9:        if CAS($tree_n, N, \langle N.list,$ false$, N.c + 1\rangle$)
L10:          return
L11:       continue
L12:    $L \leftarrow$ READ($tree_{2n}$)
L13:    $R \leftarrow$ READ($tree_{2n+1}$)
L14:    if $L.dirty$  $\triangleright$ Ensure left not $dirty$
L15:       moundify($2n$)
L16:       continue
L17:    if $R.dirty$  $\triangleright$ Ensure right not $dirty$
L18:       moundify($2n + 1$)
L19:       continue
L20:    if val'($L$) $\leq$ val'($R$) and val'($L$) $<$ val'($N$)  $\triangleright$ Push problem to left?
L21:       if DCAS($tree_n, N, \langle L.list,$ false$, N.c + 1\rangle, tree_{2n}, L, \langle N.list,$ true$, L.c + 1\rangle$)
L22:          moundify($2n$)
L23:          return
L24:    elif val'($R$) $<$ val'($L$) and val'($R$) $<$ val'($N$)  $\triangleright$ Push problem to right?
L25:       if DCAS($tree_n, N, \langle R.list,$ false$, N.c + 1\rangle, tree_{2n+1}, R, \langle N.list,$ true$, R.c + 1\rangle$)
L26:          moundify($2n + 1$)
L27:          return
L28:    else $\triangleright$ Solve problem locally
L29:       if $CAS(tree_n, N, \langle N.list,$ false$, N.c + 1\rangle)$
L30:          return
```

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 `moundify`(p) will restore the mound property at $tree_p$.

In the concurrent algorithm, this guarantee enables the complete 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`(). Listing 4 presents code for achieving this goal in a lock-free manner.

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, `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 $dirty$). These cases correspond to Lines L14–L19 and L5–L6, respectively.

There are five means through which `moundify`() can linearize. The simplest is when the operation has been helped. In this case, the operation occurs at the point of its `READ` on Line L4, which discovers that the parameter is a node that is no longer $dirty$. The next simplest case is when `moundify`() is called on a leaf: Following a `READ` of the `CMNode`, the function checks the depth of the mound tree. Since we do not shrink mounds, if $tree_n$ is a leaf at the time of this read of the $depth$, then the operation can be linearized at the `READ` on Line L4, since every leaf trivially supports the mound property, even when marked $dirty$.

Though not strictly necessary, we use a CAS on Line L9 to unset the leaf's $dirty$ field. While the `CAS` could occur after the node ceases to be a leaf (i.e., after a concurrent `insert`() increments $depth$), the following invariant ensures correctness: For any $tree_p$ that is the parent of $tree_c$, any `insert`() that selects $tree_c$ as its insertion point can only decrease `val`$(tree_c)$ to some value $\geq$ `val`$(tree_p)$. Thus even if $tree_p$ is $dirty$ when $tree_c$ is created, a successful `CAS` indicates that $tree_p$ did not change since before $tree_c$ was created, and thus `val`$(tree_c)$ can only monotonically decrease from $\top$ to some $v \geq$ `val`$(tree_p)$. Thus if the `CAS` succeeds, the operation linearized at the read on Line L4.

The third and fourth cases are symmetric, and handled on Lines L20–L27. In these cases, the children $tree_r$ and $tree_l$ of $tree_n$ are `READ` and found not to be $dirty$. Furthermore, a swap is needed between $tree_p$ and one of its children, in order to restore the mound property. The `moundify`() can linearize by successfully performing the swap, using a DCAS. 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 L4, and thus any modification to $tree_r$ between Lines L13 and L21 can only lower `val`$(tree_r)$ to some value $\geq$ `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 $tree_n$, and serves as the linearization point. 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 $\geq$ `val`$(tree_n)$ or else $tree_n$ would have changed.

3.3 The Lock-Free extractMin Operation

The lock-free `extractMin`() operation appears in Listing 5. The 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 L32 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 L32.

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 L41 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 L22 and L26 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.

Listing 5 The Lock-Free `extractMin`()Operation

```
func extractMin() : $T$
L31: while true
L32:     $R \leftarrow$ READ($tree_1$)
L33:     if $R.dirty$  ▷ root must not be dirty
L34:         moundify(1)
L35:         continue
L36:     if $R.list =$ nil  ▷ check for empty mound
L37:         return $\top$
L38:     if CAS($tree_1, R, \langle R.list.next,$ true$, R.c + 1\rangle$)  ▷ remove head of root's list
L39:         $retval = R.list.value$
L40:         delete($R.list$)
L41:         moundify(1)
L42:         return $retval$
```

3.4 The Lock-Free insert Operation

The most straightforward 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, we previously argued that the correctness of `insert`() depends only on the insertion point $tree_c$ and its parent node $tree_p$. We now argue the correctness of this approach for the lock-free `insert`().

In Listing 6, the `randLeaf`() and `findInsertPoint`() functions are now more precise with their access of the $depth$ field. The field is read 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 L59. Furthermore, expansion only occurs if the random nodes were, indeed, all leaves. This is ensured by the `CAS` on Line L64.

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 L45 and L53, and then ensures the node remains unchanged by updating with a `CAS` or `DCAS`.

There are three cases for `insert`(): when an insert is performed at the root, when an insert is performed at a node whose `val`() is equal to the value being inserted, and the default case.

First, suppose that v is being inserted into a mound, v is smaller than the root value (`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 $<$ `val`($tree_1$). A `CAS` suffices to atomically add to the root, and serves as the linearization point (Line L50). 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 L29, or else it will be immaterial to the fact that Line L21 or L25 is required to swap the root with a child.

Second, suppose that some node index c is selected as the insertion point, and `val`($tree_c$) $= v$. In this

Listing 6 The Lock-Free `insert()`Operation

```
proc insert($v : T$)
L43: while true
L44:     $c \leftarrow$ findInsertPoint($v$)
L45:     $C \leftarrow$ READ($tree_c$)
L46:     if val'($C$) $< v$  ▷ ensure insertion point still valid
L47:         continue
L48:     $C' \leftarrow \langle$new $LNode(v, C.list), C.dirty, C.c + 1\rangle$
L49:     if $c = 1$ or val'($C$) $= v$  ▷ can we insert with CAS?
L50:         if CAS($tree_c, C, C'$)
L51:             return
L52:     else ▷ must use DCSS
L53:         $P \leftarrow$ READ($tree_{c/2}$)
L54:         if val'($P$) $\leq v$  ▷ ensure appropriate parent value
L55:             if DCSS($tree_c, C, C', tree_{c/2}, P$)
L56:                 return
L57:     delete($C'.list$)

func findInsertPoint($v : \mathbb{N}$) : $\mathbb{N}$
L58: while true
L59:     $d \leftarrow$ READ($depth$)
L60:     for $attempts \leftarrow 1 \ldots$ THRESHOLD
L61:         $leaf \leftarrow$ randLeaf($d$)
L62:         if val($leaf$) $\geq v$
L63:             return binarySearch($leaf, 1, v$)
L64:     CAS($depth, d, d + 1$)

func randLeaf($d : \mathbb{N}$) : $\mathbb{N}$
L65: return $random\ i \in [2^{d-1}, 2^d - 1]$
```

case, too, the value of any node in the tree other than $tree_c$ cannot affect the correctness of the operation, and neither can the fact of $tree_c$ being $dirty$. Inserting v at $tree_c$ has no logical affect on `val`($tree_c$), and thus a simple `CAS` can again be used (Line L50). The `CAS` serves as the linearization point of the `insert`().

This brings us to the third and final 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 `READ`s that support $tree_c$'s selection as the insertion point. This is achieved through `READ`s on Lines L45 and L53, and thus the reads performed by `findInsertPoint`() are immaterial to the correctness of the insertion. The `DCSS` on Line L55 suffices to linearize the `insert`().

In the third case, the $dirty$ fields of $tree_p$ and $tree_c$ do not affect correctness. Suppose that $tree_c$ 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 that $tree_p$ is $dirty$. In this case, for Line L55 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 $tree_p$ either is 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.

4 A Fine-Grained Locking Mound

We now present a mound based on fine-grained locking. Our implementation is a transformation from the lock-free algorithm. To realize this algorithm, we add a lock bit to each mound node. Furthermore, 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 an explicit $dirty$ field in each mound node.

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 Listings 7 and 8. 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 primarily 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 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 [7]. A series of such `DCAS` instructions offers additional savings, since locks are not released and reacquired: a `moundify`() that would require J `DCAS`es (costing $5J$ `CAS`es) in the lock-free algorithm requires only $2J + 1$ `CAS`es in the locking algorithm. A critical section corresponding to a `DCSS` in the lock-free algorithm requires 2 `CAS`es in the locking algorithm, which matches the Luchangco `k-CSS` for $k = 2$.

5 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.

5.1 Mounds as a Replacement for Heaps

As with traditional heaps, if the mound is implemented with a linked data structure, then merging two mounds $M1$ and $M2$ can be done in $O(log(N))$ time. The `merge`() operation need only (a) create a new root node R with value $\top$ and $dirty =$ **true**, (b) atomically link the roots of $M1$ and $M2$ as the children of R, and then (c) call `moundify`() on R. The cost of this approach is that the tree implementation may not be amenable to binary search along a leaf-to-root path, resulting in an increase to $O(log(N))$ `insert`() complexity. Achieving atomicity for `merge`() is likely to require a more powerful atomic instruction than `DCAS` in the lock-free case. However, since merging is likely to be infrequent, as long as the instruction

Listing 7 The Fine-Grained Locking Mound Algorithm (1/2)

```
type $LMNode$
  $LNode*$   $list$   $\triangleright$ sorted list of values stored at this node
  $boolean$  $lock$   $\triangleright$ true if node locked

func setLock($i : \mathbb{N}$) : $LMNode$
F1:  while true
F2:    $N \leftarrow$ READ($tree_i$)
F3:    if $\neg N.lock$ and CAS($tree_i, N, \langle N.list,$ true$\rangle$)
F4:      return $N$

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

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

composes cleanly with existing DCAS and DCSS implementations, the impact will be minimal. For a fine-grained locking mound, the implementation is straightforward.

In order to support Dijkstra's algorithm, a priority queue must implement an efficient increaseKey() operation. In a mound, this can be achieved by adding a boolean $moved$ field to each $LNode$. Logically,

Listing 8 The Fine-Grained Locking Mound Algorithm (2/2)

```
type $LMNode$
  $LNode*$   $list$   ▷ sorted list of values stored at this node
  $boolean$  $lock$   ▷ true if node locked

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:     $tree_c \leftarrow \langle C.list,$ 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:     $tree_{c/2} \leftarrow \langle P.list,$ false$\rangle$
F50:     return
F51:   else ▷ unlock $tree_c$ and $tree_{c/2}$, then try again
F52:     $tree_{c/2} \leftarrow \langle P.list,$ false$\rangle$
F53:     $tree_c \leftarrow \langle C.list,$ false$\rangle$
```

if a node's $moved$ field is $true$, then the node does not exist. If a READ detects a $moved$ $LNode$, it is responsible for removing the node via an atomic CAS. Given such a field, increaseKey() is simple: the operation must atomically (a) set an LNode's $moved$ field to true and (b) insert() the new value of the node. Special care is needed to prevent a value from being removed from the mound concurrently with the increaseKey() which is likely to require a "4-compare-2-swap" in the lock-free case, with a trivial locking implementation. The overhead is $O(log(log(N))$, an improvement over binary heaps. As in heaps, the requirement for an increaseKey() operation necessitates a mechanism for managing handles to queue entries. In a mound, managing handles will be simple since $LNode$s are already reached by indirection.

5.2 New Opportunities for Mounds

Since the mound uses a fixed tree as its underlying data structure, it is amenable to three nontraditional uses. 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.

Third, mounds can be used in place of bag data structures, by executing extractMin() or extractMany()

on any randomly selected non-null node. While lock-free bag algorithms already exist [19], this use demonstrates the versatility of mounds.

6 Evaluation

In this section, we evaluate the performance of mounds using targeted microbenchmarks. Experiments labeled “Niagara2” were collected on a 1.165 GHz, 64-way Sun UltraSPARC T2 with 32 GB of RAM, running Solaris 10. The Niagara2 has eight cores, each eight-way multithreaded. On the Niagara2, code was compiled using gcc 4.3.2 with –O3 optimizations. Experiments labeled “x86” were collected on a 12-way HP z600 with 6GB RAM and a 2.66 GHz Intel Xeon X5650 processor with six cores, each two-way multithreaded, running Linux kernel 2.6.32. The x86 code was compiled using gcc 4.4.3, with –O3 optimizations. On both machines, the lowest 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.

6.1 Implementation Details

We implemented `DCAS` using a modified version of the technique proposed by Harris et al [7]. The resulting implementation resembles an inlined nonblocking software transactional memory [6]. We chose to implement `DCSS` using a `DCAS`. Initial experiments using Luchangco’s k-CSS [14] did not offer a noticeable advantage, since the `DCSS` did not account for a significant fraction of execution time.

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 $CMNode$ 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 nonfaulting loads, we used a per-thread object pool to recycle $LNodes$ 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.

We compare two lock-free mound implementations. The first adheres to the presentation in Section 3. The second employs a small amount of laziness. Specifically, after `moundify`() restores the mound invariant at a given node, it only recurses to a dirty child (Lines L22 and L26) if the child is of depth 12 or less. This ensures that “hot” nodes at the top of the tree are kept clean, but allows for “cold” nodes to remain dirty indefinitely.

6.2 Effect 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} `insert`()s were performed, followed by $2^{19} + 2^{18}$ `extractMin`()s. 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} randomly selected operations (an equal mix of `insert`() and `extractMin`()).

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)
Decreasing	N/A

Table 1: 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)
Decreasing	524288	N/A
Decreasing	786432	N/A

Table 2: Incomplete mound levels after many `extractMin`() operations. Mounds were initialized with 2^{20} elements, using the same insertion orders as in Table 1.

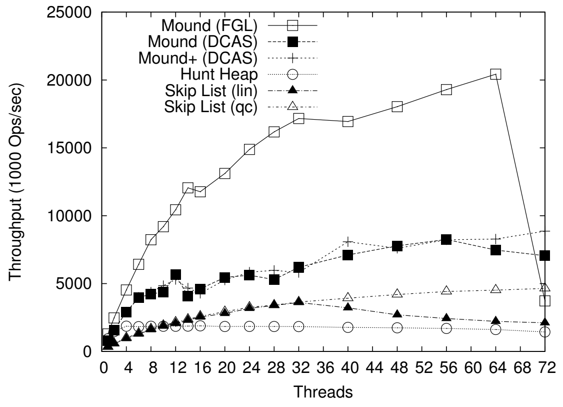
Table 1 describes the levels of a mound that have nodes with empty lists after 2^{20} insertions. For all but the last of these levels, incompleteness is a consequence of the use of randomization. Each value inserted was chosen according to one of three policies. When each value is larger than all previous 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. The other extreme occurs when every element inserted is smaller than all previous elements. In this “Decreasing” case, the mound organizes itself as a sorted list stored at the root.

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.

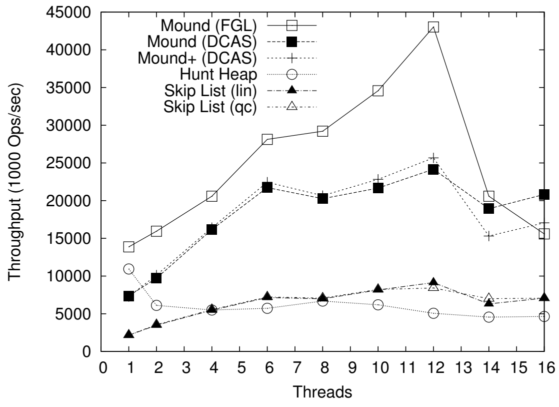
We next measured the impact of `extractMin`() on the depth of mounds. In Table 2, we see that randomization leads to levels remaining partly filled for much longer than in heaps. After 75% of the elements have been removed, the deepest level remains nonempty. Furthermore, we found that the repeated `extractMin`() operations decreased the average list size significantly. After 786K removals, the largest list in the mound had only 8 elements.

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



(a) Niagara2



(b) x86

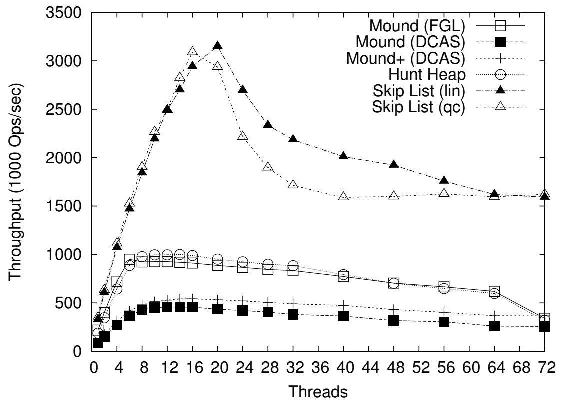
Figure 1: Insertion test: each thread inserts 2^{16} randomly selected values.

To simulate real-world use, we pre-populated a mound, and then executed 2^{20} operations (an equal mix of `insert`() and `extractMin`()), using randomly selected keys for insertions. The result in Table 3 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 elements.

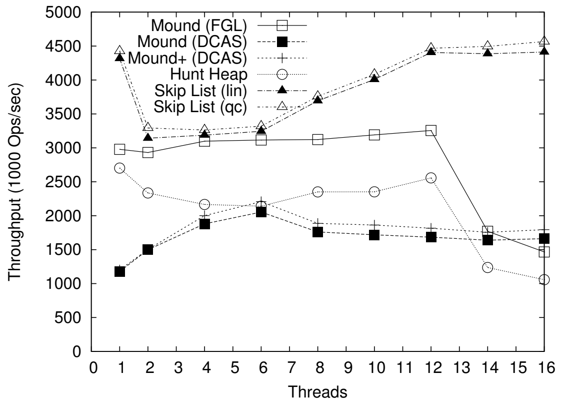
6.3 Insert Performance

Next, we evaluate the latency and throughput of `insert`() operations. As comparison points, we include the Hunt heap [10], which uses fine-grained locking, and two skiplist implementations. The first is a transformation of Fraser's skiplist [4] into a quiescently consistent priority queue [8].[1] The second is an unpublished linearizable skiplist-based priority queue, achieved by applying Lotan and Shavit's timestamp technique to the quiescently consistent implementation [13]. 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 1 presents `insert`() throughput. The extremely strong performance of the fine-grained locking mound is due both to its asymptotically superior algorithm, 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 structural hardware bottleneck after 8 threads, and high overhead due to our implementation of `DCAS` with multiple `CAS` instructions. 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.



(a) Niagara2



(b) x86

Figure 2: ExtractMin test: each thread performs 2^{16} `extractMin`() operations to make the priority queue empty.

6.4 ExtractMin Performance

In Figure 2, 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 empty. The skiplist implementation is almost perfectly disjoint-access parallel, and thus on the Niagara2, it scales well. 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.

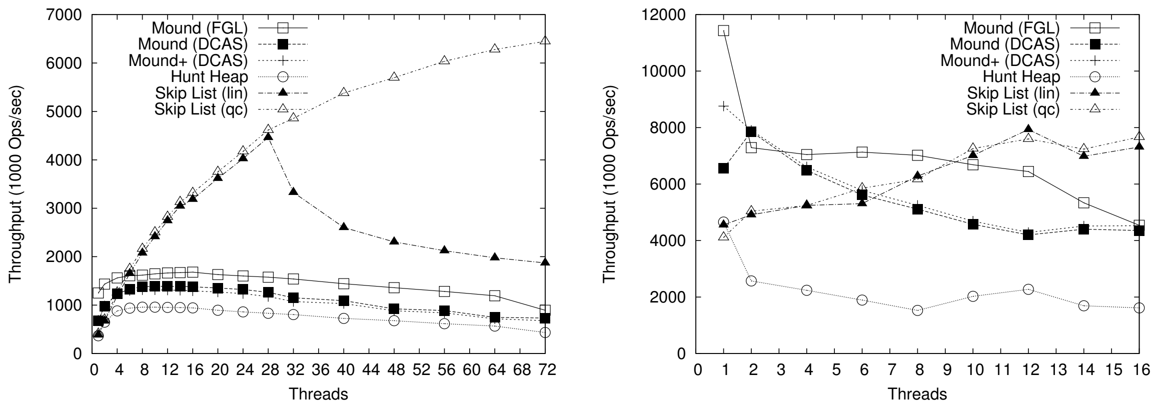
The algorithms of the locking mound and the Hunt queue are similar, and their performance curves match closely. Slight differences on the x86 are largely due to the shallower tree of the mound, and its lack of a global counter. However, in both cases performance is substantially worse than for skiplists. 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. We observe a slight benefit from the use of laziness in the "Mound+" algorithm, since it avoids some `DCAS`es.

6.5 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 3 as a more realistic evaluation than the previous single-operation experiments.

The experiment has some surprising results. The most unexpected is the difference between quiescent consistency and linearizability in the skiplist on the Niagara2. Their behavior is expected to differ only when there are concurrent `insert`() and `extractMin`() operations. We see that at higher thread counts, the linearizable implementation experiences a significant drop. The drop roughly corresponds to an expected inflection point related to hardware multithreading, and its severity suggests that more `extractMin`()

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



(a) Niagara2

(b) x86

Figure 3: Equal mix of random `insert`() and `extractMin`() operations on a queue initialized with 2^{16} random elements.

operations are incurring $O(nun\_threads)$ overhead. Another source of overhead is that at high thread counts, the shared counter used to achieve linearizability becomes a bottleneck.

On the x86, we see that the locking mound provides the best performance until 10 threads, but that it again suffers under preemption. The lazy and regular 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 CASes becomes a significant overhead.

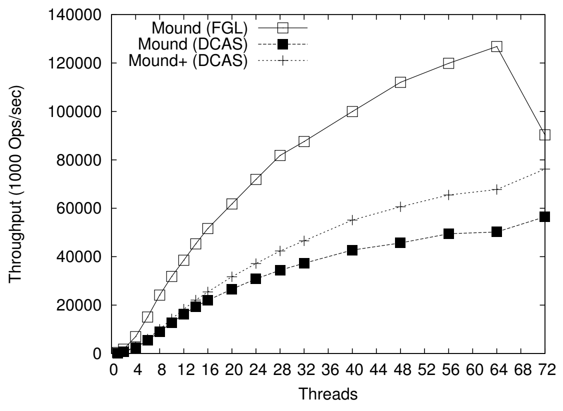
6.6 ExtractMany Performance

One of the advantages of the mound is that it stores a collection of elements at each tree node. As discussed in Section 5, implementing `extractMany`() entails only a simple change to the `extractMin`() operation. However, its effect is pronounced. As Figure 4 shows, `extractMany`() scales well.

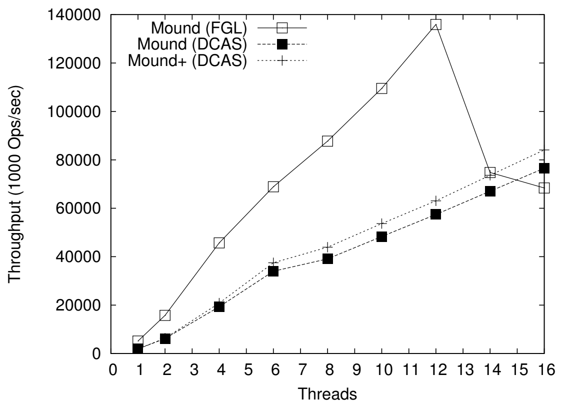
This scaling supports our expectation that mounds will be a good fit for applications that employ prioritized or probabilistic work stealing. However, there is a risk that the quality of data in each list is poor. For example, if the second element in the root list is extremely large, then using `extractMany`() will not provide a set of high-priority elements. Table 4 presents the average list size and average value of elements in a mound after 2^{20} insertions of random values. As desired, extracted lists are large, and have an average value that increases with tree depth. Similar experiments using values from smaller ranges are even more pronounced.

7 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



(a) Niagara2



(b) x86

Figure 4: ExtractMany performance. The mound is initialized with 2^{20} elements, and then threads repeatedly call `extractMany()` until the mound is empty.

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

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. In workloads that can tolerate quiescent consistency, skiplists remain the preferred choice.

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 [2], 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] 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.

[3] 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.

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

[5] T. Harris. A Pragmatic Implementation of Non-Blocking Linked Lists. In *Proceedings of the 15th International Symposium on Distributed Computing*, Lisbon, Portugal, Oct. 2001.

[6] 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.

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

[12] A. Kogan and E. Petrank. Wait-Free Queues with Multiple Enqueuers and Dequeuers. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, San Antonio, TX, Feb. 2011.

[13] 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.

[14] V. Luchangco, M. Moir, and N. Shavit. Nonblocking k-compare-single-swap. In *Proceedings of the 15th ACM Symposium on Parallel Algorithms and Architectures*, San Diego, CA, June 2003.

[15] M. Michael. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, June 2004.

[16] M. M. Michael. Scalable Lock-Free Dynamic Memory Allocation. In *Proceedings of the 25th ACM Conference on Programming Language Design and Implementation*, Washington, DC, June 2004.

[17] 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.

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

[19] H. Sundell, A. Gidenstam, M. Papatriantafilou, and P. Tsigas. A Lock-Free Algorithm for Concurrent Bags. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures*, San Jose, CA, June 2011.

[20] 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.

[21] R. K. Treiber. Systems Programming: Coping With Parallelism. Technical Report RJ 5118, IBM Almaden Research Center, Apr. 1986.