A Practical, Scalable, Relaxed Priority Queue

Tingzhe Zhou
tiz214@lehigh.edu
Lehigh University

Maged Michael
maged.michael@acm.org
Facebook

Michael Spear
spear@cse.lehigh.edu
Lehigh University

ABSTRACT

Priority queues are a fundamental data structure, and in highly concurrent software, scalable priority queues are an important building block. However, they have a fundamental bottleneck when extracting elements, because of the strict requirement that each `extract()` returns the highest priority element. In many workloads, this requirement can be relaxed, improving scalability.

We introduce ZMSQ, a scalable relaxed priority queue. It is the first relaxed priority queue that supports each of the following important practical features: (i) guaranteed success of extraction when the queue is nonempty, (ii) blocking of idle consumers, (iii) memory-safety in non-garbage-collected environments, and (iv) relaxation accuracy that does not degrade as the thread count increases. In addition, our experiments show that ZMSQ is competitive with state-of-the-art prior algorithms, often significantly outperforming them.

ACM Reference Format:
Tingzhe Zhou, Maged Michael, and Michael Spear. 2019. A Practical, Scalable, Relaxed Priority Queue. In *48th International Conference on Parallel Processing (ICPP 2019), August 5–8, 2019, Kyoto, Japan.* , 10 pages. https://doi.org/10.1145/3337821.3337911

1 INTRODUCTION

Priority queues are an important data structure for high-performance scalable systems. However, unlike ordered and unordered maps, for which there are many known high-performance data structures [3, 5], scalable priority queues remain elusive. While there have been several concurrent priority queues in the research literature [7, 9, 10, 14], they all achieve sub-linear scalability for mixed workloads. One of the most significant challenges in creating a scalable priority queue is its strict sequential specification, which requires each `extractMax()`[1] operation to return the highest-priority element in the queue. This creates a scalability bottleneck.

Recent works [1, 13, 15] showed that programs can tolerate when `extractMax()` returns a *high*-priority element that is not the *highest*-priority element, so long as there is a bound on the number of consecutive calls to `extractMax()` that do not return the highest-priority element. There are two reasons why such a relaxation is acceptable. First, while a *linearizable* [6] priority queue guarantees a total order on `extractMax()` operations, it does not guarantee ordering between an `extractMax()` and subsequent use of the returned value. That is, if Thread T_1 extracts element E_1, and then T_2 extracts E_2, where $E_1 > E_2$, programs typically do not synchronize *after* the call to `extractMax()`, and thus T_2 may use E_2 before T_1 uses E_1. Second, in many graph algorithms, processing elements out of order still contributes to the forward progress of an application [12]. If E_2 is processed before E_1, then either (a) E_1's subsequent processing will not invalidate the work done with E_2, or else (b) re-processing E_2 will be quick, because some of the total work on E_2 will have been done already. As an example of the former, consider a priority scheduler for client-submitted jobs: As long as the customer paying for high priority work is guaranteed the service-level agreement, it does not matter if other work, for other customers, occasionally happens first. As an example of the latter, consider Dijkstra's single-source shortest path algorithm: The work done processing elements out of order still advances the computation toward a solution.

Relaxed priority queues balance a decrease in the accuracy of `extractMax()` for an increase in scalability. The most important design decision involves how to relax the queue. All prior work makes accuracy a function of the thread count: as the thread count increases, the likelihood of `extractMax()` returning the highest-priority element decreases. Furthermore, prior algorithms do not support blocking on empty queues and do not guarantee that extractMax() from a nonempty queue always succeeds in extracting an element. Many real programs expect to be able to block threads that are without work [4], and thus cannot use such queues.

In this paper, we introduce ZMSQ, the first relaxed priority queue that supports the following practical features: guaranteed success of extraction when the queue is nonempty, the ability to block consumer threads, memory-safety without depending on automatic garbage collection, and relaxation accuracy that does not degrade as the thread count increases.

The ZMSQ algorithm uses several novel techniques, described in detail in the rest of the paper, to achieve the above features. It uses a small shared pool of high priority elements for fast extraction, and periodically replenishes the pool from the main data structure. The use of a scalable shared pool without any thread-specific structures enables the algorithm to guarantee that it observes an empty queue only when indeed there are no elements in the queue. The shared pool is structured to support optional scalable low-latency consumer blocking, as well as non-blocking conditional extraction and/or spin-waiting. The data structures are organized to be amenable to protection by hazard pointers [11] for memory safety. The data structures are managed such that a tunable level of relaxation is maintained (provided the queue contains enough elements) regardless of the number of threads and regardless of the input and use patterns.

[1]WLOG, this paper assumes that larger values represent higher priorities.

ICPP 2019, August 5–8, 2019, Kyoto, Japan

ACM ISBN 978-1-4503-6295-5/19/08…$15.00
https://doi.org/10.1145/3337821.3337911

2 RELATED WORK

2.1 Relaxed Priority Queues

The k-LSM priority queue [15] employs thread local log-structured merge-trees (LSMs) of at most k elements. When the size of a thread's local LSM exceeds k, the thread merges its LSM into a global LSM. `ExtractMax()` returns the larger key obtained from the thread-local and the global LSM. This guarantees the key is one of the Tk largest keys, where T is the number of threads. Since there is no synchronization on thread-local LSMs, and less contention on the global LSM, the k-LSM scales linearly for insertions, and satisfactorily for `extractMax()`. However, thread local structures makes it difficult to determine when the queue is empty: if all of the queue's items are in T_1's LSM, then calls to `extract()` by T_2 cannot return them. The MultiQueue [13] also employs thread local queues, and suffers these same problems.

The SprayList [1] represents the relaxed priority queue as a skiplist, and relaxes the precision of `extractMax()` by "spraying" the access to a range of keys at the front of skiplist. The size of the range available to `extractMax()` is proportional to T. SprayList achieves scalability in `extractMax()` by reducing the contention on the first node. However, like k-LSM and MultiQueue, SprayList's `extractMax()` becomes increasingly imprecise as the thread count T increases. To prevent costly synchronization when traversing the underlying skiplist on insertions and extractions, traversals are optimistic, and elements are removed from the skiplist lazily. This necessitates the use of a tracing garbage collector.

2.2 The Mound

The mound [9] is a lock-free concurrent heap implemented as a binary tree of sorted lists. For every tree node N_p with children N_{cl} and N_{cr}, $N_p.list.head \geq max(N_{cl}.list.head, N_{cr}.list.head)$. To insert key k, a thread chooses a random empty leaf, and then does a binary search on the path from that leaf to the root (N_R), stopping when it finds a node N_c with parent N_p, for which $N_c.list.head \leq k$ and $N_p.list.head > k$. It then inserts k as the head of $N_c.list$. `ExtractMax()` removes the head from N_R's list, and then checks if $N_R.list.head$ became smaller than the head of one of its children's lists. If so, it swaps the lists of N_R and the child with the larger list head value. The swapping process recurses downward as necessary to restore invariants in every subtree.

Relaxing the mound invariant at the root could transform the mound into a relaxed priority queue. However, the mound is extremely sensitive to the order in which elements are inserted. We found that after inserting a series of randomly chosen values, the quality of elements in each list was poor, with the second element in any node's list rarely exceeding the first value in either child's list. For experiments with a mix of `insert` and `extractMax()`, the average length of lists decreased over time. At the time scale of real-world workloads, the mound becomes a heap, rendering this approach to relaxation ineffective.

3 DATA STRUCTURE DESIGN

ZMSQ employs the mound's structure, but substantially improves the quality of a node's data versus the mound. There are two goals: ensuring each node has many elements, and ensuring the elements at each node are close in value, so that nodes near the root will have many elements that are close in priority. ZMSQ also introduces an explicit mechanism for extracting many operations at once, a new synchronization strategy, memory safety, and support for blocking threads when the queue is empty.

3.1 Data Types

We define $TNode$ as a node in the ZMSQ tree, consisting of a set of values and a $lock$. To reduce latency and synchronization, a $TNode$ caches its set's min and max values, as well as its $count$ of elements, in `atomic` variables that are only updated while holding $lock$.

$TNode$s are organized as a binary tree. Conceptually, each field has $left$, $right$, and $parent$ fields. In practice, the ZMSQ $nodes$ field is an array of arrays of $TNode$s. In $nodes$, the sub-array at position i stores 2^i $TNode$s. This representation of a binary tree allows binary searches along the path from any node to the root.

The remaining fields of ZMSQ are $leafLevel$, $batch$, $targetLen$, $pool$, and $poolNext$. $leafLevel$ indicates the deepest level of $nodes$ whose sub-array contains non-null values. $batch$ and $targetLen$ are user-defined parameters: $batch$ sets an upper bound on the number of elements (in addition to the maximum) that can be produced by `extractPool()`, and $targetLen$ defines the number of elements to try and store in each $TNode$. A set may hold at most $2 \times targetLen$ elements. $pool$ is a reference to a set of up to $batch$ elements, and $poolNext$ is an atomic integer.

3.2 The Insertion Algorithm

Our insertion algorithm appears in Listing 1. It aims to achieve a high number of elements in each $TNode$'s set. It also seeks to reduce the range of elements in each set, with an aim of having most, if not all, of the elements in the set exceed the maximum elements in the sets of any $TNode$'s $left$ and $right$ children.

In the original mound, `insert(`k`)` finds a $TNode$ into which it can insert k as the new maximum without violating any parent/child invariants. The motivation for this design was to avoid locking. However, as previously discussed, this strategy can not ensure that sets have many items. In our initial experiments, we found that after a few million operations in a mixed workload with an equal number of `insert()` and `extractMax()` operations, where `insert()` selected keys based on a normal distribution, the mound degraded to a regular heap.

To increase set size, when an `insert(k)` operation selects as its starting point a leaf that is at least three levels deep, for which $max > k$ and $count < targetLen$ (lines 8-9), ZMSQ inserts k into the leaf's set (lines 36-45). When there are many elements in the priority queue, this ensures that most non-leaf $TNode$s have $targetLen$ elements in their set, because it is unlikely that a leaf will migrate upward before it receives many insertions.

When `insert(k)` must traverse upward (lines 6-7), the insertion must increase the number of elements at a $TNode$. In a manner similar to mound insertions, the default behavior of ZMSQ insertions is to find a node N, such that $N.max \leq k \wedge N.parent.max > k$ (line 51), and atomically insert k into N's set (lines 11-35). Though generally beneficial, this default strategy can mean that a $TNode$ has *too many* elements. In the the pathological case, this can lead to the entire ZMSQ devolving into a single set. It can also lead to

Listing 1: Insert

```
1  function selectPosition(val)
2      while true do
3          level ← leafLevel
4          for attempt ∈ 1 . . . 1 + level^2 do
5              slot ← rand() mod 2^level
6              if nodes[level][slot].max <= val then
7                  return ⟨level, slot, false⟩ // insert as max of some ancestor
8              if level > 3 ∧ nodes[level][slot].count < targetLen then
9                  return ⟨level, slot, true⟩ // insert as non-max of this TNode
10         expandTree(level) // couldn't find good leaf, so expand tree
11 function regularInsert(level, slot, val)
12     if level = 0 then
13         nodes[level][slot].lock.acquire()
14         if val < nodes[level][slot].max then
15             nodes[level][slot].lock.release()
16             return false // Root changed, became poor candidate
17         nodes[level][slot].set.insert(val)
18         nodes[level][slot].max ← val
19         nodes[level][slot].count ← nodes[level][slot].count + 1
       else
20         nodes[level − 1][slot/2].lock.acquire() // lock the parent
21         nodes[level][slot].lock.acquire()
22         if val ≥ nodes[level − 1][slot/2].max ∨ val <
             nodes[level][slot].max then
23             nodes[level − 1][slot/2].lock.release()
24             nodes[level][slot].lock.release()
25             return false // Node or parent changed, became poor candidate
           // next line may update nodes[level-1][slot/2].min
26         val ← swapValueFromParentOpt(level-1, slot/2, val)
           // if val swapped from parent.set, this.min or this.max could change
27         nodes[level][slot].set.insert(val)
28         nodes[level][slot].max ← max(nodes[level][slot].max, val)
29         nodes[level][slot].min ← min(nodes[level][slot].min, val)
30         nodes[level][slot].count ← nodes[level][slot].count + 1
31         nodes[level − 1][slot/2].lock.release()
32     if nodes[level][slot].count > 2 × targetLen then
33         startPruning(level, slot) // recursively distribute from set to children
34     nodes[level][slot].lock.release()
35     return true
36 function forceInsert(node, val)
37     node.lock.acquire()
38     if val > node.max ∨ node.count > targetLen then
39         node.lock.release()
40         return false // Could not insert as non-max in node's set
41     node.set.insert(val)
42     node.min ← min(node.min, val)
43     node.count ← node.count + 1
44     node.lock.release()
45     return true
46 function Insert(val)
47     while true do
48         ⟨level, slot, force⟩ ← selectPosition(val)
49         if force ∧ forceInsert(nodes[level][slot], val) then return
50         if ¬force then
               // Find node root s.t. node.val ≤ val ∧ node.parent.val > val
51             ⟨level, slot⟩ ← binarySearchPosition(level, slot, val)
52             if regularInsert(level, slot, val) then return
```

extractPool() returning too many elements, which can result in too much relaxation. To avoid these problems, when inserting k into $N.set$, we also check $N.count$, and if it becomes more than twice $targetLen$, we split N's set, and then merge the second half of the set (with smaller elements) into N's children (lines 32-33). If this makes either child's set too big, we repeat the splitting process on that child. In our experiments, we found that the split does not happen frequently when $targetLen$ is larger than 16.

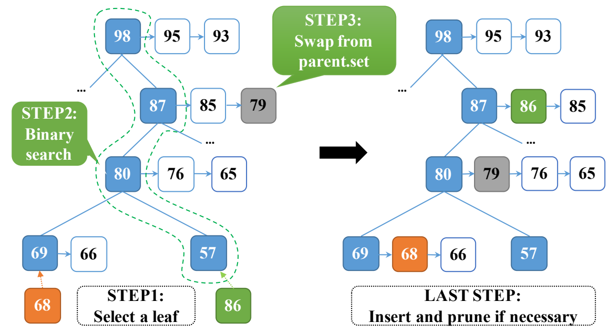


Figure 1: Inserting two nodes with $targetLen = 3$

The top few levels experience the most contention from extractMax(). To avoid increasing contention, we do not apply the above optimization on the top three levels of the tree. Even so, these changes have the effect of stabilizing $TNode.set$ size. We ran an experiment in which the ZMSQ was initialized with 1M elements and $targetLen = 32$, and then we performed 8M insert()/extractMax() pairs. After initialization, $count$ varied from 32 to 51 across all non-leaf nodes. Upon completion of the experiment, the average $count$ was 32 for all nodes (standard deviation 2.76).

These changes offer three main benefits. First, they reduce the cost of tree traversals, since the tree is more compact. Especially for extractMax(), which may need to migrate a set from root to leaf, this reduces the number of levels by 4–5. Second, less memory is required, since the number of $TNodes$ is reduced substantially. Finally, these changes decrease the frequency with which our relaxed version ($batch > 0$) of extractMax() touches the root.

In addition to improving the average size of sets, we also modified the insert() algorithm to improve the *quality* of sets. Suppose value k is to be added as the maximum in N_c's set. We first inspect its parent's min: if $N_c.parent.min < k$, then we insert k into the parent, and move $N_p.min$ into N_c. (Note that k may not be the smallest element at N_p, and $N_p.min$ may be smaller than some of the elements in N_c.) This technique decreases the range of values in N_p, which improves quality. It may increase the range of values in N_c. However, there will be more opportunity to improve the quality of the set in N_c in subsequent insertions, since it cannot satisfy any extractMax() calls until N_p satisfies at least two. This enhancement occurs on line 26.

Figure 1 depicts these two changes to insertion. For reference, consider the original mound data structure. To insert 86, the original mound would place it before 80 as the new head of that node's list. To insert 68, the original mound would either (a) randomly choose to start at 57, and make 68 the new head of that list, or (b) randomly choose to start at 69, which would increase the depth of the mound and cause 68 to be a child of 69.

In the ZMSQ algorithm, insert(86) would find node 80, but would also observe that 80's parent's set would be more compact if 79 was replaced with 86. Thus it inserts 86 into the parent's set and moves 79 to the set containing 80. Recursive splitting does not happen in this case, since the length of the set containing 80 is not larger than $2 \times targetLen$. Note that there is no added

Listing 2: ExtractMax

```
function extractPool()
    nodes[0][0].lock.acquire()
    if nodes[0][0].count = 0 then
        nodes[0][0].lock.release()
        return ⊥ // Empty root → empty tree
    if batch > 0 ∧ poolNext >= 0 then
        nodes[0][0].lock.release()
        return false // can't move from root's set to non-empty pool
    while ∃i pool[i] ≠ ⊥ do spin // Wait for lagging consumers to finish
    val ← nodes[0][0].max
    nodes[0][0].set.remove(val)
    size ← min(batch, nodes[0][0].count − 1)
    moveToPool(nodes[0][0].set, size)
    nodes[0][0].count = nodes[0][0].count − size − 1
    poolSize ← size
    swapSetDownward(0,0)
    return val
function extractFromPool()
    if batch = 0 ∨ poolNext < 0 then return ⊥
    index ← poolNext.fetchSub(1)
    if index ≥ 0 then
        val ← pool[index]
        pool[index] ← ⊥
        return val
    else return ⊥
function ExtractMax(T &v)
    while true do
        val ← extractFromPool()
        if val = ⊥ then val ← extractPool()
        if val ≠ ⊥ then return val
        backoffWait()
```

synchronization: locks on the nodes holding 87 and 80 were already required. As long as *targetLen* is not too large, this optimization improves the accuracy of ZMSQ without a measurable increase in overhead.

Similarly, `insert(68)` exhibits a new behavior when it selects the node containing 69. Since $node.count < targetLen$, `forceInsert` is used to insert 69 in the set. This helps to reduce the height of the tree, and also increases the density of sets. We forbid this operation for the top three levels of the tree, since this optimization may cause the set to contain a low-priority element.

3.3 Extracting Elements

Pseudocode for extracting elements from the ZMSQ appears in Listing 2. When $batch > 0$, `extractMax()` uses an auxiliary data structure, *pool*, whose size is given by $poolNext$. It decrements $poolNext$ to get an *index*. If $index \geq 0$, the thread returns the value at $pool[index]$. Otherwise, it must replenish the pool. It extracts $n \leftarrow min(N_R.count, batch{+}1)$ elements from the root, reserving the largest for itself and placing the remainder into *pool* in sorted order. Setting $poolNext$ allows subsequent `extractMax()` calls to use the pool, instead of the tree. `extractMax()` then restores invariants by recursively trading sets between parents and children, starting at the root, until every parent's set's maximum is greater than the maximum in either child's set. Note that when $batch = 0$, the ZMSQ `extractMax()` algorithm behaves exactly like the mound, and is guaranteed to return the largest element in the priority queue.

3.4 Concurrency

In order to achieve scalable concurrency, we require techniques that can perform each operation atomically (i.e., ensure that no operation observes the intermediate state of another operation). At a high level, we place a lock in each $TNode$, and threads may not modify a $TNode$ without holding the lock on that $TNode$. Note that threads may *read* atomic fields of a $TNode$ without holding the $TNode$'s lock. Parents are always locked before children.

The `insert(k)` operation takes several forms. The simplest inserts into a non-head position of a leaf. In this case, after reading $N.max$ and $N.count$ and determining that N has space for k in a non-head position of its set, the thread locks N and double-checks $N.max$ and $N.count$. If either has changed in an unsatisfactory manner, the `insert()` restarts. Otherwise, k is added to N's set, $N.min$ is possibly updated, and then the lock is released.

The second form inserts k as the maximum in N while ensuring $N.parent.max$ remains larger than k. After reading $N.max$ and $N.parent.max$, we lock $N.parent$ and N, then double-check that $k \geq N.max \wedge k < N.parent.max$. If not, we unlock both nodes and restart the `insert()`. Otherwise, we insert k in $N.set$ and update $N.max$. Note that if a concurrent insertion reads $N.max$ before it is updated, there is no danger: if the insertion decides that N is its target node, it will re-check $N.max$ after locking N. If its key is smaller than $N.max$, then completing the insertion will only increase $N.max$. If its key is greater than $N.max$ and the operation traverses upward, changes to $N.max$ are immaterial.

The third form inserts k at N, making N's set too large. When we transfer elements to N's children, we must ensure no value v appears to be "missing" as it moves from $N.set$. Before unlocking N, but after splitting N's set in half, we lock N's children. Then N can be unlocked, and then the elements added to the children. Since the children are locked before N is unlocked, no subsequent `extractMax()` can see the pre-split state of the child and the post-split state of N. Likewise, any changes to *max* fields during this process will not interfere with concurrent `insert()` operations, for the reasons outlined in the previous paragraph. Once the lock on N is released, if a child's set is too large, the process of migrating the second half of its set downward can be repeated on its children as needed. In practice, this is rare.

Finally, suppose k *could be* inserted at node N, but $N.parent.min < k$. As with the second form, we begin by locking $N.parent$ and N. Then we check $N.parent.min$. If swapping the minimum into N and placing k in $N.parent$ would be worthwhile, we may do so without further concurrency control: inserting into $N.parent$ cannot change $N.parent.max$, and the swap has the same safe interaction with concurrent operations as inserting k into N would have.

Next, consider `extractMax()` with $batch = 0$. In this case, we first lock the root (N_R), remove its largest element, and update its val. We then lock both children of N before inspecting their values. This step is essential, or else a concurrent insertion at a child of N could violate the main invariant. Once both are locked, we determine either (a) no exchanging of sets is needed, in which case the operation completes, or (b) the root and one of its children should be exchanged. In this case, one child is unlocked, the root and other child exchange sets, the root is unlocked, and then the invariant repair repeats with the locked child and its children.

When $batch > 0$, `extractMax()` can take two forms. In the first, it extracts up to $batch$ elements from N_R, puts them in *pool*, and restores invariants between N_R and its children. In the second, it only operates on the pool. Protecting *pool* and $poolNext$ with N_R's lock does not scale. Instead, to `extractMax()`, a thread first atomically decrements $poolNext$. If the result is not negative, it is the index of the position in *pool* whose value can be returned. Otherwise, the thread locks N_R and re-checks that $poolNext$ is negative. If not, N_R is unlocked and the operation retries. If so, the thread populates *pool* from N_R's set (reserving one value for itself), sets $poolNext$ via an atomic write, restores invariants between N_R and its children, and returns the value it reserved.

3.5 Safe Memory Reclamation

A significant benefit of our algorithm is that it does not require garbage collection. At any time, the algorithm holds references to at most three $TNode$s, or one $TNode$ and the *pool*. Furthermore, many of these accesses occur while a $TNode$ is locked. The only optimistic accesses are (a) to *pool*, in `extractMax()`, (b) to pairs of $TNode$s, during the traversal phase of `insert()`, and (c) while locking $TNode$s. As a result, we can use two hazard pointers [11] per thread. (The choice of set implementation may introduce a requirement for one more hazard pointer.)

For insertions, a traversal must acquire and release hazard pointers in a hand-over-hand manner. When `extractMax()` accesses the root, it must hold a hazard pointer on the root $TNode$. However, when it only accesses *pool*, no hazard pointer is needed: even if *pool* is a reference, the wait on line 8 of Listing 2 ensures that *pool* will not be reclaimed while a thread with a non-negative result from line 18 is accessing it.

3.6 Blocking

While research data structures often let threads spin when there is no work (i.e., consuming from an empty queue), production systems face multi-tenancy and pay-for service constraints. If waiting would be common, vendors and customers prefer that waiting threads block instead of spin.

We developed a low-latency blocking mechanism, which causes threads to sleep in `extractMax()` when the priority queue is empty. A sketch of the implementation appears in Listing 3. The general idea is to maintain a circular buffer of futexes (the Linux kernel's fast userspace mutex object). Note that by reading the low bit of futex f from userspace, a thread can determine if there are any threads sleeping on f.

In our design, two atomic integers count the total number of `insert()` and `extractMax()` operations. They also represent indexes into the circular buffer, representing the next position to sleep and the next position to wake. Each position in the circular buffer contains a futex, padded to fill a cache line. To block `extractMax()` when the queue is empty, we call our wakeup code after every `insert()` and our sleep code before every `extractMax()`. In the common case, each call is a single fetch-and-increment. When threads must modify futexes, the counters disperse threads, so that there is low contention on an array of futexes, and so that we do not wake too many threads at once. While the algorithm is general-purpose, its value derives from the fact that we can quickly and

Listing 3: Blocking algorithm

```
Data:
  futex[]   : Futex[]   // an array of futex
  cticket   : atomic    // id for recording number of extractMax, initially 1
  pticket   : atomic    // id for recording number of insert, initially 1
  Num       : const     // total number of futex
  Stride    : const     // gap between futex to avoid false sharing

function signalAfterInsert()
    p ← pticket.fetchAdd(1)
    loc ← getFutexArrayLoc(p)
    curfutex ← futex[loc]
    while true do
        ready ← p << 1
        if ready + 1 < curfutex then return
        if futex[loc].CAS(curfutex, ready) then
            if curfutex & 1 then
                futexWake(&futex[loc])
            return
        else curfutex ← futex[loc]

function waitBeforeExtractMax()
    c ← cticket.fetchAdd(1)
    if futexIsReady (c) then return
    loc ← getFutexArrayLoc(c)
    curfutex ← futex[loc]
    if curfutex & 1 then futexWait(&futex[loc], curfutex)
    if trySpinBeforeBlock () then return
    while true do
        curfutex ← futex[loc]
        if curfutex & 1 then futexWait(&futex[loc], curfutex)
        else if ¬futexIsReady(c) then
            blkfutex ← curfutex + 1
            if futex[loc].CAS(curfutex, blkfutex) then
                futexWait(&futex[loc], blkfutex)
        else return
```

accurately check if the queue is empty; otherwise, false waits and unnecessary system calls could occur.

3.7 Summary of Design Choices and Trade-offs

We conclude this section by briefly reviewing the properties of ZMSQ, and how they differ from other relaxed priority queues.

Accuracy. We define the accuracy of a relaxed priority queue by the number of consecutive `extractMax()` operations that fail to return the maximum key. In SprayList, the accuracy is based on probabilities that decrease as the number of threads (T) increases; however, a thread is guaranteed that repeatedly calling `extractMax()` will eventually return the maximum, and will return one of the first $O(Tlog^3T)$ elements with high probability. However, it is possible for `extractMax()` to fail even when the SprayList is not empty. In k-LSM, if T threads repeatedly call `extractMax()`, then the maximum value will be returned with frequency at least $1/(Tk)$. However, if the thread with the maximum in its LSM suspends, then an unbounded number of `extractMax()` operations will fail to return the maximum, unless some synchronization is added to access per-thread LSMs. Likewise, if a thread's LSM is empty, and the global LSM is empty, then its calls to `extractMax()` can fail even if other LSMs are full. In ZMSQ, the accuracy is not dependent on T, but instead on a tunable parameter $batch$. This allows the programmer to choose the relaxation, and the maximum is guaranteed to be returned with probability $1/batch$. This worst case occurs when the two largest elements (e_0 and e_1) are at the heads of the root node's set and the head of one of its children's sets. Note that this also guarantees that $k \times batch$ calls to `extractMax()` are guaranteed to return the top k elements. No similar bound exists for SprayList, MultiQueue, or k-LSM. Furthermore, `extractMax()` never fails to return a value when the queue is nonempty, and if a thread suspends, there is no risk of another thread's call to `extractMax()` failing to find the maximum value.

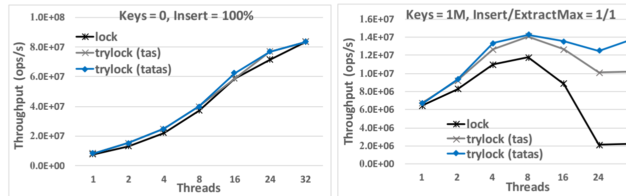
Sensitivity to Input Pattern. The mound is highly sensitive to input pattern; the SprayList is unaffected by input pattern. ZMSQ falls in between: by allowing insertion into non-head positions in a set, ZMSQ avoids the mound's worst-case pattern (inserts ordered decreasing by value lead to sets of size 1). ZMSQ does have a worst-case input pattern, where inserts occur in an order such that, for all nodes N, the non-head elements of a set rooted at node N are smaller than all values in the sets of N's descendent's. The randomized selection of a starting point for insertions makes it difficult to create this pattern. During testing, we randomly generated priorities to insert, and then calculated the average mean priority for each $TNode$; for such a workload, the largest values were always in the upper levels.

Generality. While our experiments are tailored to compare against existing systems in the contexts for which they were designed, we contend that ZMSQ is more general. It does not leak memory, and is hence correct in languages like C++. Its support of blocking allows its use in environments where spinning is not permissible. Unlike nonblocking queues, it can store arbitrary data types without requiring extra indirection, and it does not have high contention on a single head node, unlike a hypothetical SprayList based on a blocking skiplist.

4 EVALUATION

In this section, we present the results of a set of microbenchmarks and applications that measure the performance of ZMSQ. Tests were performed on a machine with two 2.1GHz Intel Xeon Platinum 8160 processors and 192GB of RAM. Each processor has 24 cores / 48 threads. Since the machine has non-uniform memory access latencies (NUMA) and our algorithms are not NUMA-aware, we limited experiments to a single processor. The machine ran Red Hat Linux server 7.4, and we used the GCC 7.2.1 compiler with O3 optimization. All data points were an average of 15 runs. We used the jemalloc allocator [2]. We considered two implementations of ZMSQ. The default implementation mirrors the mound, in that it implements its *set* as a linked list. Curves labeled "array" implement *set* as an unsorted array of maximum size $2 \times targetLen$.

The *targetLen* and *batch* parameters affect both performance and accuracy. Recall that *targetLen* represents the target size of the set in each $TNode$; it affects performance because it limits the value of *batch* and influences the frequency with which sets are split. *batch* places an upper bound on the number of elements that can be cached for subsequent `extractMax()` calls, before an expensive call to `extractPool()` is needed. When *batch* is zero, `extractMax()` *always* returns the largest element; as *batch* increases, accuracy can decrease. Our goal in this evaluation is two-fold: to show how these parameters affect the performance and accuracy of ZMSQ, and also to provide guidelines for users to choose the best configuration for their application requirements.



(a) Insert-only benchmark **(b) Mixed benchmark**

Figure 2: Influence of trylocks (Higher is better)

4.1 Lock Implementations

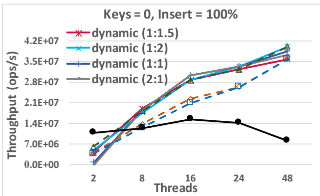
`insert()` makes heavy use of an optimistic read-before-lock pattern, where a thread T optimistically reads $TNode.max$ when selecting the right position to insert a value. These optimistic reads need to be checked again after the node is locked, and if the check fails, the lock has to be released and the operation retried. While correctness requires that we always execute the check, we can predict its failure: if T attempts to lock N, but N is locked, then $N.max$ is likely to change before T acquires $N.lock$, and thus T is likely to restart its operation. Based on this intuition, it could be beneficial to use a trylock when acquiring N, and to retry immediately on trylock failure. Note that retrying `insert()` will lead to choosing a different path through the tree, and thus it is unlikely to re-encounter the same locked node N.

In Figure 2, we run 1M operations on a ZMSQ configured with $batch = 32$ and $targetLen = 32$. In Figure 2a, all operations are inserts, the queue is initially empty, and keys are chosen from a normal distribution. In Figure 2b, there is an even mix of `insert()` and `extractMax()` operations, and the queue is initialized with with 1M keys. We compare three locks: the C++ `std::mutex`, a test-and-set (TAS) trylock, and a test-and-test-and-set (TATAS) trylock. The y-axis represents throughput.

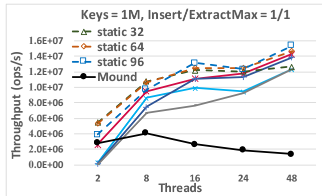
In Figure 2a, trylock only performs slightly better than regular locks. This is because `insert()` has small critical sections, and those critical sections rarely touch the same nodes of the tree, since each `insert()` chooses a random leaf as its starting point. In Figure 2b, the impact is more significant. With $batch = 32$, only 3% of `extractMax()` calls access the root, but when they do, they must lock three nodes, and they often swap sets and recurse downward. These critical sections are long relative to `insert()`, and using trylocks lets conflicting `insert()` operations give up early and try from a different starting point. As the queue size decreases, the performance gap increases (not shown), because `insert()` is more likely to access the root. Moreover, the TATAS trylock outperforms the TAS trylock, because it reduces cache contention on locks.

4.2 batch and targetLen

targetLen determines the average set size in each $TNode$, and influences the compactness of the tree. *batch* lower bounds the frequency with which `extractMax()` returns the largest value in the queue, and hence accuracy. It also alleviates contention at the root among concurrent `extractMax()` operations. To demonstrate how *batch* and *targetLen* work together to affect performance, we



(a) Insert-only benchmark



(b) Mixed benchmark

Figure 3: Influence of *batch* and *targetLen* (Higher is better)

(a) 1K

	Threads / batch	2	4	8	16	32	64
10%	ZMSQ	83	71	58	58	60	55
	SprayList	100	92	73	46	4	6
50%	ZMSQ	470	440	339	314	299	293
	SprayList	498	463	405	400	272	224

(b) 64K

	Threads / batch	2	4	8	16	32	64
0.1%	ZMSQ	58	52	41	38	31	31
	SprayList	65	58	34	12	6	2
1%	ZMSQ	604	566	409	337	269	168
	SprayList	655	649	620	510	500	179
10%	ZMSQ	6005	6034	5964	5678	5435	5232
	SprayList	6552	6542	6504	6337	6242	5887

Table 1: Accuracy affected by `batch` for ZMSQ and thread numbers for SprayList (Higher is better)

present two sets of configurations in Figure 3: the dynamic configuration increases the size of *batch* and *targetLen* as the thread count increases, so that the ratio *batch*/*targetLen* is constant, and the smaller of the two numbers equals the thread count. For example, when the thread count is 8, dynamic (1: 1.5) represents `batch` (8), `targetLen` (12). The static configuration keeps *batch* and *targetLen* equal and constant across all thread counts.

With 100% inserts (Figure 3(a)), the mound has 1.76× the performance of the best ZMSQ on 2 threads. This highlights the added overheads that come from ZMSQ's quality-enhancing modifications. The benefit of this added cost is not merely in the relaxed `extractMax()`: we even see it in the scalability of the 100% insert workload, where our modifications deliver a shallower tree, denser lists, and more work at leaves, which all contribute to better scalability. The experiment also shows a tradeoff: when *targetLen* grows, there are more cache misses when exchanging a value between a *TNode* and its parent, due to list traversal. With a *targetLen* of 64 and 96, this hurts performance.

In Figure 3(b), dynamic configurations perform significantly worse at low thread counts: their *targetLen* values are too small, and the ZMSQ structure resembles a heap. Small *targetLen* values also increase latency for both `extractMax()` and `insert()`. In contrast, the static configurations have large *targetLen* values even for small thread counts. Dynamic (1:1.5) generally performs best among all dynamic configurations. With profiling, we found that dynamic (1:1.5) had the highest percentage of full sets. Dynamic(1:2) and (2:1) tend to perform worse than dynamic(1:1): when *targetLen* » *batch*: a *TNode*'s min is often very small, and remains in the *TNode* after a call to `extractPool()`, causing many recursive swaps. We also found that when *batch* » *targetLen*, `extractPool()` typically extracts fewer than *batch* elements.

64 offered the most consistent performance, but 96 offered the best performance at high thread counts. Reasons include the impact of *targetLen* on the cost of accessing sets when exchanging elements between a parent and child during `insert()`, and the effect of *batch* on the frequency of calls to `extractPool()`. Since the size of *batch* matters when there is a high contention for `extractMax()`, higher *batch* values increase in importance as the thread count increases. We recommend the static (*batch*=48, *targetLen*=72) configuration as the default setting.

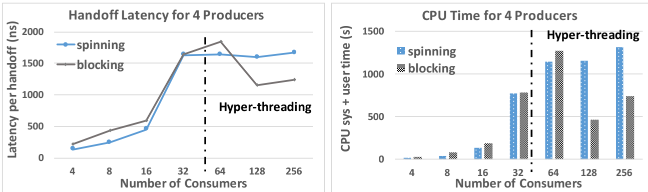
4.3 Accuracy

Next, we measure the accuracy of ZMSQ and compare it with the SprayList [1], which is considered the current state-of-the-art in relaxed priority queues. In the experiments, we initialize each queue with 1K and 64K randomly generated keys without duplicates. For the 1K sized queues, we execute 102 (10%) and 512 (50%) `extractMax()` operations, and report the number of returned keys that are in the top 102 and 512 respectively. For the 64K sized queue, we execute 65 (0.1%), 655 (1%), and 6553 (10%) `extractMax()` operations. For the SprayList, we vary the number of threads, since the precision of the SprayList depends on the number of threads (i.e., with 1 thread, the SprayList is a strict priority queue). For ZMSQ, we set *targetLen* to 64 and vary *batch*, because the accuracy depends exclusively on *batch* whenever $batch \leq targetLen$.

Table 1a shows the accuracy test for the small queue size. More than half of `extractMax()` operations meet the threshold in ZMSQ among all configurations. We can see the accuracy decreases as *batch* increases. However, the accuracy does not show a significant change when *batch* grows beyond 8. The result suggests that ZMSQ provides high-quality results. Recall that high *batch* values mean that an increasing amount of data is being provided by the *pool*, and a decreasing amount is guaranteed to be optimal. Since the *pool* is filled with (mostly) high-priority values, the accuracy does not degrade as *batch* increases.

In contrast, SprayList accuracy shows a more significant drop-off, especially when the thread count exceeds 32. This is because each `extractMax()` is guaranteed to obtain a key from a region close to the front of the SprayList. However, the size of the region is proportional to the thread count. With fewer than 8 threads, the SprayList has better accuracy than ZMSQ, because the spray strategy guarantees a small region, and every `extractMax()` returns a key close to the best key. At 32 threads and beyond, the SprayList is even worse than a FIFO queue when extracting the top 10% from a small queue and the top 0.1% from a large queue.

In Table 1b, we consider a larger queue. ZMSQ is competitive except for the 1% test with $batch > 8$. This is because of a brief dip in performance for our technique of improving quality: the first few additions to the ZMSQ are at shallow depths, for which we do not apply our accuracy-improving techniques. These *TNodes* will have few elements. As insertions increase the depth of the ZMSQ, some leaf *TNodes* have the chance to achieve good density, but they propagate upward quickly, at which point they serve more `extractMax()` than their accuracy should permit. This is a transient



(a) Handoff latency

(b) CPU time for 1M handoffs

Figure 4: Blocking VS. Spinning latency (Lower is better)

state during initialization, and it passes quickly, so that by the time 10% of the elements have been extracted, elements are usually of high quality. As future work, we will look into ways to adjust *targetLen* and *batch* based on the number of prior operations on the ZMSQ, to prevent accuracy violations from manifesting during this brief transitional phase from shallow to deep trees.

In general, the ZMSQ provides competitive accuracy compared to the SprayList. Neither is likely to degrade to the pessimal performance of a FIFO queue, and both are subject to occasional perturbations that lead to one or the other having higher accuracy. Furthermore, in ZMSQ concurrency does not reduce accuracy, since small batch sizes can be used even at high thread counts. This should afford the user more opportunity to tune and balance performance and accuracy.

4.4 Blocking

Figure 4 shows the impact of blocking strategies for a producer/-consumer workload. Note that consumers can encounter an empty queue. We measure the latency of a producer/consumer handoff and correlate it to the blocking strategy. We show that our futex design does not hurt performance at low thread counts, and helps performance at high thread counts and hyperthreading.

Our choice of having consumers outnumber producers is motivated both by the low complexity of insertions in ZMSQ, and also by common industry practice. We considered 2, 4, and 8 producers and varied from 2 to 256 consumers. Queue are initially empty, with $batch = 32$. Due to limited space, we only show the case with 4 producers, but the other results exhibited the same behavior. Notice, too, that each socket in our machine contains 24 cores/48 threads, so both hyperthreading and preemption effects are at play.

Figure 4a shows the latency for each handoff for 1M total handoffs. With 4 producers and 4 consumers, the latency for spinning is 133ns, and blocking adds 50ns per handoff. The spinning algorithm always achieves lower latency when threads do not outnumber cores. However, at high thread counts (more than 64 consumers), the latency per handoff with blocking is significantly better.

To further evaluate the efficiency of blocking, we used the `time` command in Linux to calculate the CPU execution time for 1M handoffs. The result is shown in Figure 4b. When the number of consumers is below 64, blocking uses 1% to 90% more CPU time than spinning. However, the blocking algorithm reduces the CPU execution time by more than half for more than 64 consumers. In addition to these results, it is worth noting that in systems with indeterminate arrival of new elements, a common case in practical systems, the savings in CPU usage as a result of supporting consumer blocking are unbounded.

4.5 Micro-Benchmarks

Next, we compare the performance of the SprayList, Mound, and ZMSQ. ZMSQ curves labeled "(array)" implement $TNode.set$ as a fixed-size array. Otherwise, the set is implemented as a singly linked list. Based on the discussion in Section 4.2, we use $batch = 48$ and $targetLen = 72$ for the ZMSQ experiments. The only exception is the "ZMSQ-BEST" curve. This shows the best performer at each thread count, from the seven configurations in Figure 3. Additional discussion of the impact of tuning parameter appears in Section 4.7.

All algorithms were implemented in C++. However, the SprayList is not memory-safe: logically deleted nodes can remain reachable for a long time, and cannot be safely reclaimed without garbage collection. Therefore, the SprayList always leaks memory in our experiments. The Mound was designed to use epoch-based reclamation, and the implementation we compare against leaks memory. While our focus is on the memory-safe ZMSQ, we include a result ("ZMSQ (leak)") that leaks memory, to assess the impact of memory management with hazard pointers.

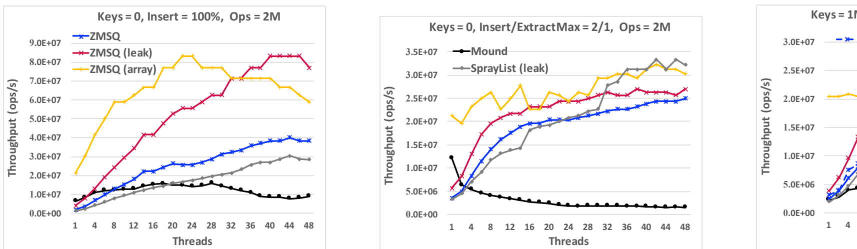
4.5.1 Mixed Push and ExtractMax. We first consider the performance of `insert()` under two scenarios: 100% inserts and 66% inserts. The throughput is calculated by executing 2M operations on a queue that is initially empty. SprayList only outperforms ZMSQ for the benchmark with 66% inserts and with more than 32 threads. Results appear in Figure 5(a)(b).

With 100% inserts, ZMSQ (array) has the best single thread performance by 17× versus the SprayList, with the memory-safe ZMSQ 56% faster and leaky ZMSQ 3× faster. Like mound, `insert` is asymptotically faster in ZMSQ than in SprayList. Additionally, the array implementation has little allocation and deallocation, and the absence of pointer chasing makes set management (e.g., for swapping an element with its parent during insertion) fast.

In the 66% workloads, the mound suffers, both because it devolves into a heap, and because of the cost of recursive cleanup in `extractMax()`. Our default (memory-safe) ZMSQ outperforms the SprayList until roughly the point of hyperthreading (> 24 threads). The strong performance of ZMSQ (array) derives in part from locality in `extractPool`, where the *pool* can be populated from the root's *set* with a constant number of cache misses.

In these two experiments, the overhead of memory safety can be seen in the difference between the ZMSQ and ZMSQ (leak) curves. While it is invalid to conjecture that the difference between these curves would also manifest as a depression in some hypothetical memory-safe SprayList, we are nonetheless encouraged: the ZMSQ is often the best algorithm, despite it offering two features (memory safety and blocking) that are not present in the SprayList.

Figure 5(c) considers an equal mix of `insert()` and `extractMax()` operations. In addition to the experiment with 20-bit keys in the figure, we also considered 7-bit keys. With 7-bit keys the relaxed priority queues are all too shallow to scale. Degradation was worst for mound, while sustained throughput and accuracy were best for ZMSQ. For 20-bit keys, ZMSQ scales to the full size of the machine, but the slope of its scalability changes after 8 threads. When we



(a) Insert-only **(b) 66% Insert + 34% extractMax** **(c) 50% Insert + 50% extractMax**

Figure 5: Performance of mixed insert and extractMax (Higher is better)

experimented with different *batch* and *targetLen* values, we were able to achieve higher throughput at high thread counts, albeit at the cost of worse throughput at low thread counts. While it is possible to dynamically select these values based on thread count, to do so would merely optimize a microbenchmark. We instead encourage users to tune these parameters along with the number of threads, the ratio of `insert()` to `extractMax()` operations, and the amount of non-queue work done by each thread. As before, ZMSQ (array) has the lowest single-thread overhead, by a factor of more than 5×, but does not scale as well.

4.5.2 Producer and Consumer Pattern. One of the most important patterns for a priority queue is a producer/consumer workload. We ran experiments where dedicated producer (`insert()`) and consumer (`extractMax()`) threads accessed a queue that was initially empty. We varied the producer/consumer ratio, and measured the time to transfer 1M items from producers to consumers.

Figure 6 shows performance for different ratios of consumers and producers. ZMSQ has strong performance across all of the ratios tested, even with precise memory reclamation. This is partly because ZMSQ `extractMax()` always returns a value when the queue is nonempty. In contrast, SprayList `extractMax()` can return ⊥ when the queue contains elements. For high thread counts, SprayList consumers make multiple `extractMax()` calls just to get one element from a non-empty queue.

In these experiments, we omitted ZMSQ (array), which was not significantly different from the list-based ZMSQ: in both cases, the queue typically has few elements, and thus pools tend to have few elements. The primary benefit of ZMSQ in these workloads is that insertion is fast, and thus consumers rarely wait to get data from a concurrent producer. We also disabled blocking features in these experiments, since SprayList does not support blocking.

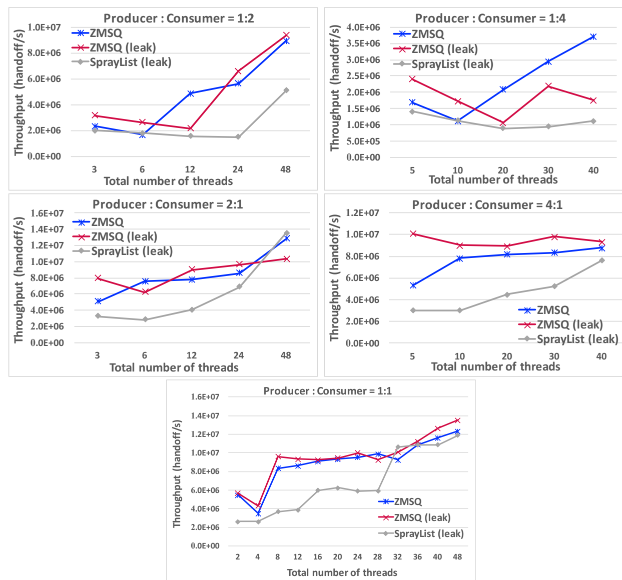
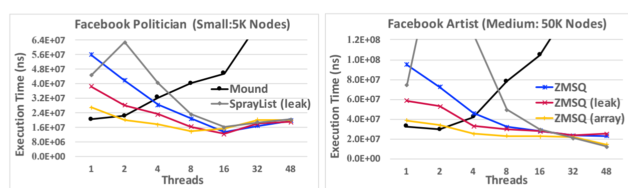


Figure 6: Producer / Consumer pattern (Higher is better)



(a) Small Graph **(b) Medium Graph**

Figure 7: Single source shortest path (Lower is better)

4.6 Single Source Shortest Path

In the above experiments, there was no penalty when `extractMax()` returned an item that was not the true maximum value in the queue. The conjecture behind relaxed priority queues is that realistic workloads can tolerate these inaccuracies. To validate this claim, we repeat experiments proposed by the SprayList authors, in which a concurrent single source shortest path algorithm is run on real-world data sets. We consider two graphs from Facebook: Artist has 50K nodes to process, and Politician has 6K nodes. We use the same experimental harness as [1]. Based on the optimal result for the tuning experiment in Section 4.7, ZMSQ used $batch = 42$ and $targetLen = 64$. Results appear in Figure 7.

In the Artist workload, all of the queues scale. SprayList offers slightly better performance with hyperthreading (> 24 threads), but more variance and higher execution time at low thread counts. Furthermore, beyond 8 threads, the cost of memory management for ZMSQ is negligible. As in previous experiments, ZMSQ (array) has the best performance at low thread counts. However, at higher

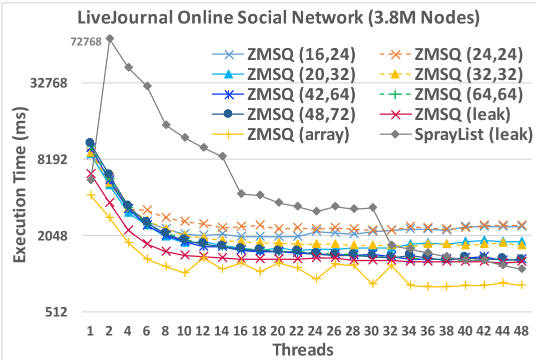


Figure 8: Parameter tunning for large SSSP (Lower is better)

counts the other ZMSQ implementations match it. In contrast, for Politician, the graph is too small to afford real opportunities for speedup. All three queue implementations degrade after 16 threads. However, the lower latency of the ZMSQ implementations leads to faster execution than SprayList up to 36 threads. For both workloads, the mound performs worse for all but the lowest thread counts: the precision of its `extractMax()` operation is not as important as avoiding locking the root.

4.7 Tuning ZMSQ

Figure 8 considers a larger data set. With 3.8M nodes, the LiveJournal Online Social Network [8] affords the opportunity to observe the impact of different $\langle batch, targetLen\rangle$ values. We also show the leaky and array versions of the best performing ZMSQ ($\langle 42, 64\rangle$). The y-axis is logarithmic.

At one thread, memory reclamation overheads cause all but ZMSQ (leak) and ZMSQ (array) to perform worse than SprayList. However, at two threads the SprayList performance degrades, because it ceases to act as a strict priority queue. In contrast, ZMSQ does not incur any accuracy penalty for adding threads: the addition of threads only leads to more available processing for the same amount of relaxation. By 12 threads, ZMSQ is 7× its single-thread performance, whereas concurrent SprayList does not even surpass its single-threaded performance until 14 threads. At this point, there are diminishing returns for ZMSQ, but performance is relatively stable. SprayList performance does not match ZMSQ until 36 threads, and never surpasses ZMSQ (array).

The "best" choice of *batch* and *targetLen* is largely independent of the thread count, and therefore tuning is straightforward: we needed to find the a good value for *batch*, and a good ratio between *batch* and *targetLen*. From previous subsections, we knew that $batch = targetLen$ would result in `extractPool()` rarely returning a full pool of *batch* elements. We also knew that larger *batch* values would result in lower accuracy, but better scalability. The seven curves in the figure were chosen based on an approximation of how a programmer would refine a search. There are two main findings. The first is that several choices delivered roughly the same performance. The second is that choices with good performance were easy to find. These results suggest that it will be easy for programmers to find good parameters.

5 CONCLUSIONS AND FUTURE WORK

In this paper, we introduced ZMSQ, a relaxed concurrent priority queue algorithm with novel techniques that enable it to support important practical features and deliver robust performance and accuracy. It is capable of scalable low-latency blocking, and guarantees the success of extraction from nonempty queues. It is memory-safe without dependence on automatic garbage collection. Its relaxation accuracy does not degrade with the increase in the number of threads, and its relaxation performance is robust regardless of input and use patterns. It scales well without too much relaxation (batch = 32), and when extractMax() does not return the maximum value, the returned values are of high priority.

The code for the ZMSQ is available as part of the open-source Facebook Folly library (as RelaxedConcurrentPriorityQueue). As future work, we plan to investigate the use of helper threads to improve the quality of sets in the ZMSQ. We are also looking into mechanisms that would insert high-priority items directly into the pool, so that they could be extracted immediately.

ACKNOWLEDGMENTS

We thank Dave Watson for his advice and guidance during the conduct of this research. This work was supported by the NSF under Grant CAREER-1253362. Any opinions, findings, and conclusions expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] Dan Alistarh, Justin Kopinsky, Jerry Li, and Nir Shavit. 2015. The SprayList: A Scalable Relaxed Priority Queue. In *Proceedings of the 20th ACM Symposium on Principles and Practice of Parallel Programming*. San Francisco, CA.
[2] Jason Evans. 2017. jemalloc memory allocator. http://http://jemalloc.net/.
[3] Vincent Gramoli. 2015. More Than You Ever Wanted to Know about Synchronization. In *Proceedings of the 20th ACM Symposium on Principles and Practice of Parallel Programming*. San Francisco, CA.
[4] Rachid Guerraoui, Hugo Guiroux, Renaud Lachaize, Vivien Quema, and Vasileios Trigonakis. 2019. Lock-Unlock: Is That All? A Pragmatic Analysis of Locking in Software Systems. *ACM Transactions on Computer Systems* 36 (2019), 1–149.
[5] Maurice Herlihy and Nir Shavit. 2008. *The Art of Multiprocessor Programming*. Morgan Kaufmann.
[6] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: a Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems* 12, 3 (1990), 463–492.
[7] Galen Hunt, Maged Michael, Srinivasan Parthasarathy, and Michael Scott. 1996. An Efficient Algorithm for Concurrent Priority Queue Heaps. *Inform. Process. Lett.* 60 (Nov. 1996), 151–157. Issue 3.
[8] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.
[9] Yujie Liu and Michael Spear. 2012. Mounds: Array-Based Concurrent Priority Queues. In *Proceedings of the 41st International Conference on Parallel Processing*. Pittsburgh, PA.
[10] Itay Lotan and Nir Shavit. 2000. Skiplist-Based Concurrent Priority Queues. In *Proceedings of the 14th International Parallel and Distributed Processing Symposium*. Cancun, Mexico.
[11] Maged Michael. 2004. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE TPDS* 15, 6 (2004), 491–504.
[12] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A Lightweight Infrastructure for Graph Analytics. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*. Farmington, PA.
[13] Hamza Rihani, Peter Sanders, and Roman Dementiev. 2015. Brief Announcement: MultiQueues: Simple Relaxed Concurrent Priority Queues. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*. New York.
[14] Hakan Sundell and Philippas Tsigas. 2005. Fast and Lock-Free Concurrent Priority Queues for Multi-Thread Systems. *J. Parallel and Distrib. Comput.* 65 (May 2005), 609–627. Issue 5.
[15] Martin Wimmer, Jakob Gruber, Jesper Larsson Traff, and Philippas Tsigas. 2015. The Lock-Free k-LSM Relaxed Priority Queue. In *Proceedings of the 20th ACM Symposium on Principles and Practice of Parallel Programming*. San Francisco.