Scalable Concurrent Priority Queue Algorithms

Nir Shavit Asaph Zemach

October 24, 1998

Abstract

This paper addresses the problem of designing *bounded range priority queues*, that is, queues that support a fixed range of priorities. Bounded range priority queues are fundamental in the design of modern multiprocessor algorithms – from the application level to lowest levels of the operating system kernel. While most of the available priority queue literature is directed at existing small-scale machines, we chose to evaluate algorithms on a broader concurrency scale using a simulated 256 node shared memory multiprocessor architecture similar to the MIT Alewife. Our empirical evidence suggests that the priority queue algorithms currently available in the literature do not scale. Based on these findings, we present two simple new algorithms, *LinearFunnels* and *FunnelTree*, that provide true scalability throughout the concurrency range.

1 Introduction

Priority queues are a fundamental class of data structures used in the design of modern multiprocessor algorithms. Their uses range from the application level to lowest levels of the operating system kernel [34]. The most effective concurrent priority queue algorithms currently available in the literature [4, 17, 6, 8, 16, 18, 29, 36] use ingenious heap based implementations to support an unbounded range of allowable priorities. Unfortunately, as we will show, their scalability is rather limited. This may be part of the reason why many highly concurrent applications are designed using sequential priority queue implementations [5, 26], or alternatively, by avoiding the use of priorities altogether. Though hard to quantify, there is reason to believe that the availability of effective concurrent priority queue implementations will ultimately simplify application design, making prioritization possible without having to take steps to limit concurrent access to the priority queue.

This paper addresses the problem of designing scalable priority queue structures. We concentrate on the design of *bounded range priority queues*, that is, queues that support a fixed range of priorities, as can be found for example in operating systems schedulers [26, 5]. By concentrating on queues with a fixed range of priorities, we can offer new algorithmic approaches that avoid the use of tightly synchronized structures such as priority heaps, search trees, and skip-lists [4, 17, 6, 8, 16, 18, 27, 29, 36].

We began our research by evaluating the performance of representative bounded range priority queue implementations based on algorithms from the literature in the context of a shared memory multiprocessor architecture with minimal hardware support. We used only the most common synchronization primitives such as register-to-memory-swap and compare-and-swap[1]. The representative algorithms, described in Section 2, included those of Hunt et al. [17], a variant of the skip lists of Pugh [27], and "list of bins" and "tree of bins" structures using the MCS locks of Mellor-Crummey and Scott [24].

[1] The load-linked/store-conditional pair or full/empty bits can also be used

We evaluated the above structures by running a series of simple synthetic benchmarks using the well accepted Proteus simulator [10] of Brewer et. al. We used Proteus to simulate a 256 processor ccNUMA multiprocessor similar to the Alewife of Agarwal et al. [1] with realistic memory bandwidth and latency. Though this is not a real 256 node machine, we note that previous research by Della-Libera [11] has shown that with appropriate scaling, Proteus simulates a 32 node Alewife machine[2] accurately for the kinds of data structures tested in this paper.

Our empirical evidence, presented in Section 4, indicates that the concurrent heap of Hunt et. al [17] and the skip lists of Pugh [27] have performance problems even at low concurrency levels.[3] On the other hand, the simple "list of bins" and "tree of bins" structures using MCS-locks perform exceptionally well at low concurrency levels. Unfortunately, for each of the above algorithms, there is a point on the concurrency scale at which contention becomes so high that performance degrades and the algorithm becomes unusable.

Based on our findings, we present in Section 3 two simple new algorithms, *LinearFunnels* and *FunnelTree*. These are variants of the "list of bins" and "tree of bins" structures that use new forms of the *combining funnel* coordination mechanism of Shavit and Zemach [33] in place of MCS locks [24] in the "bins" and tree "nodes." Combining funnels are a randomized variant of software combining trees [15, 13, 35] that support efficient parallel fetch-and-increment operations.[4] Our new versions of combining funnel structures allow us to support a novel bounded fetch-and-decrement operation. Though other structures like diffracting trees [32] and counting networks [3] provide efficient implementations of fetch-and-increment, their operations cannot be readily transformed into the new bounded fetch-and-increment required for our priority queues.

As the research of Lim and Agarwal [22, 23], Della-Libera and Shavit [12], and Karlin et al. [19] has shown, the key to delivering good performance over a wide range of concurrency levels, is the ability of a data structure to adapt to the load actually encountered. The adaption techniques of Lim and Agarwal [22] use a centralized form of coordination that replaces one entire data structure by another, say, an MCS queue-lock with a combining-tree, in order to handle higher (respectively, lower) load. Our approach here is to avoid replacing one complete structure with another, as this would require a more centralized (as opposed to distributed) algorithmic solution and strong coordination. Instead, to achieve adaptability, we construct the parts of our data structures that are potential hot-spots (their "internal nodes" and "bins") using a localized adaptive mechanism – combining funnels. In Section 3.1 we describe combining funnels in more detail and outline the new variant of the funnels we use in our constructions.

The final part of our paper, in Section 4, compares the performance of the new adaptive *LinearFunnels* and *FunnelTree* algorithms to former methods. Our conclusion is that they have the potential of being the first algorithms to provide scalable performance throughout the concurrency range.

2 Concurrent Priority Queue Implementations

We begin evaluating the performance of priority queue alternatives by testing implementations of various published methods and other methods which we consider natural choices. Our implementation of these structures was optimized to the best of our ability while still maintaining the characteristics of the original. By comparing algorithms from

[2] Though Alewife was designed to scale to hundreds of processors, the largest machine currently available has 32 nodes.

[3] Keep in mind however that they are "general" algorithms, designed to allow an unbounded range of priorities.

[4] Though we chose to use funnels, one can alternately use adaptive mechanisms such as [22] on a local level. However, their peak performance under high load is that of an optimal combining tree, which is not as efficient as a combining funnel [33].

widely varying "families," we hope to minimize the role of the actual implementation properties of the algorithms themselves. The following section describes the priority queue algorithms studied in this paper. For each we provide pseudo-code which defines the algorithm's behavior and illustrates important implementation issues. A discussion of different consistency conditions as they apply to priority queues appears in Appendix B.

SingleLock This is a heap based priority queue implemented as an array with a single MCS lock [24] on the entire data structure, as shown in Figure 11 in Appendix A. This algorithm supports arbitrary priorities and is linearizable. It is a representative of the class of centralized lock-based algorithms.

HuntEtAl This is the priority queue implementation of Hunt et al. [17]. In this algorithm (see Figure 11 in Appendix A.) there is a single lock which protects a variable holding the size of the heap. All processors must acquire it in order to begin their operations, but unlike the previous data structure it is not held for the duration of the operation. Rather, the heap's size is updated, then a lock on either the first or last element of the heap is acquired and then the first lock is released. In order to increase parallelism insertions traverse the heap bottom-up while deletions proceed top-down, insertions also employ a novel bit-reversal technique which allows a series of insertion operations to proceed up the heap independently without getting in each other's way. The implementation is based on the code from the authors' `FTP` site, optimized for Proteus. This algorithm supports arbitrary priorities, and is linearizable. It is a representative of a class of algorithms such as that of Rao and Kumar [29], Ayani [4] and Yan and Zhang [36], that use centralized locking but are sophisticated in terms of minimizing the number and duration of lock-access while traversing the shared heap structure.

2.1 Bin and counter based algorithms

The priority queue algorithms in the sequel use two shared data structures which we call `counter` and `bin`. A counter is a shared object that holds an integer and supports fetch-and-increment (`FaI`) and fetch-and-decrement (`FaD`) operations on it. Increment or decrement operations may optionally be *bounded* meaning they will not update the value of a counter beyond some specified bound (`BFaI` and `BFaD`). A bin (sometimes called a bag or a pool) is an object which holds arbitrary elements, and supports insertion of a specified element (`bin-insert`), an emptiness test (`bin-empty`), and removal of an unspecified element (`bin-delete`). Simple code for implementing these data structures appears in Figure 1. The code of `bin-insert` and `bin-delete` uses locks explicitly, but the implementation of `FaI` and `BFaI` uses the `atomically{}` operator to stress that we do not implement them with locks: we either execute these operations in hardware or implement them using combining funnels.

SkipList This is a priority queue algorithm based on Pugh's skip list structure [27], and optimized for a fixed set of priorities, as seen in Figure 12 in Appendix A. We pre-allocated N links where each link contains a bin which stores items with the link's priority. To insert an element of priority i, processor p adds the item to the bin in the i-th link. If the link is not currently threaded into the skip list it inserts it using Pugh's concurrent skip list insertion algorithm [28]. For deletion we follow the ideas of Johnson [18], by setting aside a special "delete bin." Delete operations first attempt to remove items from this bin. The first processor to find the bin empty unlinks the first bin in the skip list and sets the delete buffer to point to it. This lowers contention in the deletion phase.

```
//fetch-and-increment
int FaI(counter p)
{
    atomically {
      old = p.val
      p.val++
    }
    return old
 }

//bounded fetch-and-decrement
int BFaD(counter p, int bound)
{
   atomically {
    old = p.val
    if (old > bound) p.val--
   }
   return old
}
```

```
bin-insert(bin b, elem_type e)
{
  acquire(b.lock)
  if(b.size < MAXSIZE)
    b.elems[b.size++] = e
  release(b.lock)
}

bin-empty(bin b)
{
  return (b.size == 0)
}

bin-delete(bin b)
{
  acquire(b.lock)
  if (not bin-empty(b))
    e = b.elems[--b.size]
  release(b.lock)
  return e
}
```

Figure 1: *Right:* code for shared counter operations; *Left:* simple bin implementation.

Skip lists have been shown to have the same complexity and better empirical performance than search-tree based methods [28], and so we use this algorithm to represent the performance delivered by the class of search-tree based priority queue algorithms [6, 8, 16, 18].

SimpleLinear As shown in Figure 2, the algorithm maintains an array of bins. To insert an item with priority i a processor simply adds it to the i-th bin. The `delete-min` operation scans bins from smallest to largest priority and attempts to delete an element from non-empty bins it encounters. The scan stops when an element is found. When the bins are implemented as in Figure 1, this data structure is linearizable.

SimpleTree An algorithm based on binary trees of counters illustrated in Figure 3. The tree has N leaves where the i-th leaf holds items of priority i. $N-1$ shared counters

```
bin Bins[MAXPRI]

insert(elem_type e)
{
   bin-insert(Bins[e.pri] , e)
}
```

```
delete-min()
{
   for(i=0; i<MAXPRI;i++)
     if(not bin-empty(Bins[i])) {
       e = bin-delete(Bins[i])
       if (e != NULL) return e
     }
   return e
}
```

Figure 2: **SimpleLinear:** code for a simple bounded range priority queue algorithm with a linear layout.

```
tree_leaf Leaves[MAXPRI]
tree *Root

insert(elem_type e)
{
   n = Leaves[e.pri]
   bin-insert(n->bin, e)
   while( not root(n) ) {
      p = n->parent
      if ( n == p->left_child )
        FaI(p->counter)
      n = p
   }
}
```

```
delete_min()
{
  n = Root
  while( not leaf(n) ) {
    i = BFaD( n->counter , 0 )
    if ( i > 0 ) n = n->left_child
    else  n = n->right_child
  }
  e = bin-delete(n->bin)
  return e
}
```

Figure 3: **SimpleTree:** code for a simple bounded range priority queue algorithm with a binary tree layout showing associated counter operations.

in the tree's internal nodes count the total number of items in the all the leaves of the subtree rooted in a node's left (lower priority) child. `delete-min` operations start at the root and descend to the smallest priority non-empty leaf by examining the counter at each node. They go right if it is zero and decrement it and go left otherwise. This decision is implemented using a *bounded fetch-and-decrement* operation (`BFaD`). Since processors can overtake each other, insertions must not proceed in the same top-down manner or incorrect executions can occur. Instead, insertions of an item with priority i traverse the tree bottom-up from the i-th leaf. When ascending from the left child, the parent's counter is incremented using a fetch-and-increment operation.

In the following section we present two combining funnel based implementations of the latter two algorithms, **LinearFunnels** and **FunnelTree**.

3 The New Funnel Based Algorithms

The counter and bin implementations provided in Figure 1 are not scalable [31, 32, 33] and will be sources of contention [25] if simultaneously accessed by many processors in both SimpleLinear and SimpleTree. To create our LinearFunnels and FunnelTree priority queues, we replace the simple data structures in these potential trouble spots with combining funnel based implementations. Specifically, we employ combining funnel based counters in the inner tree nodes of SimpleTree and combining funnel based stacks to implement the bins in both algorithms.

3.1 Combining funnel basics

The following section outlines the key elements of the combining funnel data structure. The interested reader can find further details in [33]. The funnel is composed of a (typically small) number of *combining layers*, implemented as arrays in memory. These are used by processors accessing the same serial object to locate each other and combine. As processors pass through a layer, they read a PID (processor ID) from a randomly chosen array element, and write their own in its place. They then attempt to collide with the processor whose ID they read. A successful collision allows a processor to exchange information and update its operations. For example, assume processors p and q attempt to access a bin object concurrently with operations `bin-insert(A)` and `bin-insert(B)` respectively. Both processors pass through the first of the bin's

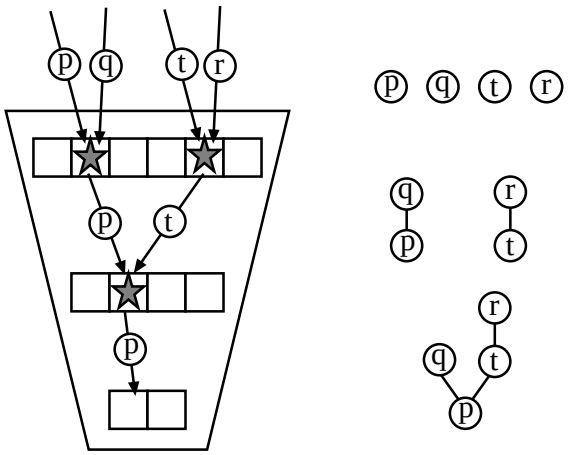

Figure 4: Example of processors going through a funnel. On the left we see p, q, r and t as they go through the first layer where p collides with q and t with r, then q and r wait while p and t advance to the second layer where they collide. On the right side we see how combining trees are dynamically formed by collisions, when the waiting processor becomes the child of the advancing processor.

combining layers and p reads q's ID. If p manages to collide with q, it updates its operation to `bin-insert({A,B})`, while q changes its operation to "`wait for B to be inserted`". Since p is now doing both operation it is considered q's parent. When the parent completes its operations, it informs its children of the results. Processor p can now continue to the next layer, if it exists. Suppose it collides with some other processor r who is doing `bin-insert(C)` and p becomes the child. Now, r's operation will be `bin-insert({A,B,C})`, while p changes its operation to "`wait for A and B to be inserted`". When r's operation is complete, it informs p, who in turn will inform q. After passing through all funnel layers, a processor is said to exit the funnel, and may now try to apply its (updated) operation on the object, in our example, the bin. Figure 4 illustrates this process.

In the scenario just given, suppose p's operation was `bin-delete()` when it collides with q. Since p wishes to remove an item, while q wants to add one, it is natural to have p return q's item, thereby satisfying both requests, without either p or q needing to access the bin itself. This process is called *elimination* [31], as it allows operation with reversing semantics to be eliminated in the early stages of funnel traversal. Elimination improves performance since it allows some operations to complete sooner and reduces contention.

Finally, funnels support *adaption*. By having each processor decide locally how many combining layers to traverse before applying its operation to the object, it can trade-off overhead (going through more layers) for increased chances of collision. Under high load, the extra overhead of more layers is justified, but under low load there is no contention so it is better to simply apply the operation and be done.

3.2 Priority queues using funnels

In order to construct combining funnel variants of SimpleLinear and SimpleTree, we need to provide implementations of some basic primitives using funnels. The paper of Shavit and Zemach [33] contains detailed pseudo-code for constructing a combining funnel based counter which supports the fetch-and-increment operation, we use that

implementation where `FaI` is needed. In section 3.3 we present a novel construction for implementing the `BFaD` operation, which cannot be directly implemented using either diffracting trees or counting networks. Together these two primitives provide for all the needed counter support. To implement the bins we use the stack construction of [33]. Thus our `bin-insert` is actually `push`, `bin-delete` is `pop`, and `bin-empty` reads the current value of the stack's pointer and compares it to zero. Like the funnels that implement them, these priority queues support quiescent consistency.

We chose to use stacks in place of bins since their funnel implementation is simple, linearizable and supports elimination. However, this can cause unfairness (and even starvation) among items of equal priority, as later insertions occlude earlier ones. Whether or not this matters is application dependent. If it does, one could either use FIFO queues (and give up elimination) or use a hybrid data structure which supports elimination in the funnel, but queues items internally in FIFO order.

LinearFunnels Returning to figure 2 we see that SimpleLinear requires only bins, which we implement using funnel based in the most straightforward manner. However, there is one point that should be stressed: the `delete-min` operation queries each bin to see if it is empty before trying to remove a value. This is crucial to the performance of LinearFunnels, since testing for emptiness is *much* faster (requires only one read) than actually going through the funnel trying to remove an element.

FunnelTree As in the previous case we replace the bin operations with stack operations. The calls to `FaI` and `BFaD` are replaced by funnel based implementation, but only for counters at the top four levels of tree, for counters below that we use MCS locks. We do not require funnels at the deeper levels since contention (traffic) there is much lower than at the top. Our experiments show that, at a cost of about 5% in performance we could have avoided making the cut-off decision and used funnels throughout the tree. Those at the bottom would have automatically shrunk, adapting to prefer low overhead and fewer combining layers. However, in our simulation this would have taken too long, so we chose this approach instead (details in the full paper).

3.3 The scalable bounded counter algorithm using funnels

In Figure 10 in Appendix A we present pseudo-code for a bounded fetch-and-increment/decrement counter using combining funnels. Note that for our priority queue scheme, we could get by with a counter that supports only two operations: an (unbounded) increment, and a bounded decrement that returns only an indication of whether the counter was decremented or not. However, for the sake of completeness, we provide an implementation that supports `bounded-fetch-and-decrement(`x`,bound)` whose pseudo-code definition appears in Figure 1, and an analogous `bounded-fetch-and-increment(`x`,bound)`.

Given a fetch-and-add operation, it is possible to construct this type of bounded counter directly as Gottlieb et al. [15] have shown. For example, bounded fetch-and-increment can be done by first applying a `fetch-and-add(`x`,1)` operation and if the value returned is greater than the bound, applying a `fetch-and-add(`x`,-1)` operation and returning the bound. Since this approach can potentially require two fetch-and-add operations, which in our case would require two traversals of the combining funnel, we will show how to avoid the extra overhead by incorporating the bounds checking operation directly into the funnel algorithm. This approach also allows to integrate elimination seamlessly into the algorithm. As seen in Figure 5, the performance gain from using elimination can be as high as 250%.

At first blush it might seem that adding the bounds check is only a trivial departure from the standard combining based fetch-and-add operation [13, 15, 33] since it appears that one can combine as usual and perform the bounds check at the root of the combining structure. Unfortunately, this approach is wrong due to the subtle fact that bounded

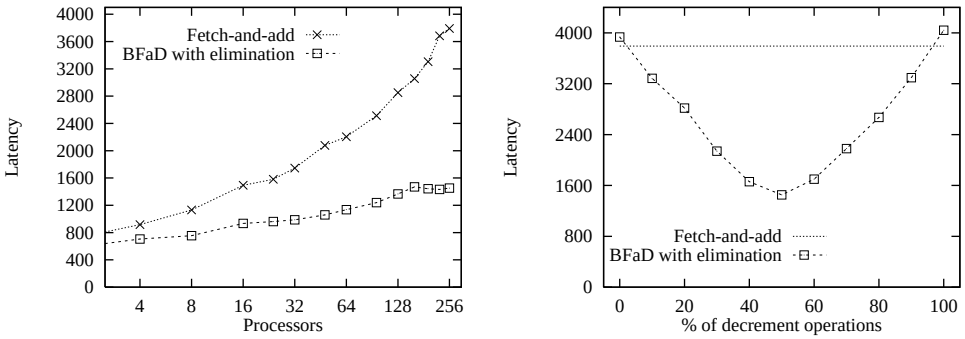

Figure 5: Comparison of latencies of combining funnel based fetch-and-add and bounded decrement. With an equal number of increment and decrement operations at varying degrees of concurrency elimination makes the bounded version substantially more efficient (*left*). At 256 processors with varying distribution of operations (*right*) eliminations are rarer and eventually fetch-and-add becomes faster since it does not incur bounds-checking overhead.

operations are not commutative. For example, consider a counter whose value is 0 and two operations: bounded-fetch-and-decrement (with `bound=0`) and fetch-and-increment, that are applied to it. If the decrement is applied first, both operations return 0, and the counter's value changes to 1. If operations are applied in the opposite order, the increment returns 0 and the decrement 1, and the counter ends up with the value 0. Because the order in which we apply operations is important, problems arise when trying to apply an entire tree of operations at once and determining what each processor's return value should be, in parallel. The way we overcome these difficulties is by forcing our trees to be homogeneous, that is, contain only one kind of operation. Because all operations are the same, we need only know how many there are to determine the final value of the counter.

4 Performance

Our tests were performed on a simulated 256 processor distributed-shared-memory multiprocessor similar to the MIT *Alewife* machine [1] of Agarwal et al. The simulator used is *Proteus*[5], a multiprocessor simulator developed by Brewer et al. [9, 10]. In our benchmarks processors alternate between performing some local work and accessing the priority queue. When accessing the queue processors choose whether to insert a random value or apply a delete-min operation based on the result of an unbiased coin flip. In all the experiments we show here local work was kept at a small constant and the queue was initially empty. In each experiment we measured *latency*, the amount of time (in cycles) it takes for an average access to the object. Each series of experiments either keeps the concurrency constant while changing the number of priorities or varies the concurrency while keeping the number of priorities constant. We ran a set of preliminary benchmarks using 256 processors and a queue of two priorities to find the set of funnel parameters (layer width, depth of funnel, delay times, etc.) which minimized latency. We used this set of parameters in all our funnels for both counters and stacks.

Our experiments show that at low concurrency levels simple algorithms are best. The total number of priorities determines whether one should use SimpleLinear (for few priorities) or SimpleTree (for a large number of priorities). As concurrency increases

[5]Version 3.00, dated February 18, 1993.

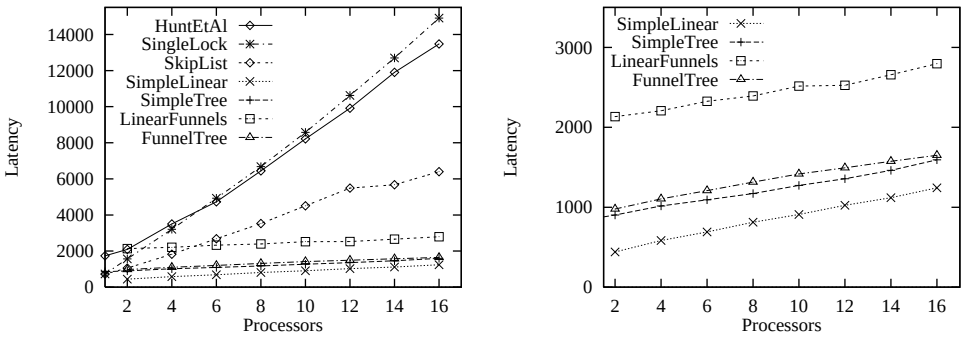

Figure 6: Latency of different priority queue implementations with 16 priorities at low concurrency levels. On the right is a close-up of the bottom part of the graph on the left.

contention begins to undermine the effectiveness of these implementations. Thus at 256 processors with 4 or less priorities, a linear arrangement of combining funnel based stacks (LinearFunnels) is the most effective priority queue. Again, supporting more priorities favors a binary tree layout, so FunnelTree is the method of choice for 8 priorities and above, at high concurrency levels.

4.1 Experimental results

The graphs in Figure 6 compare latency of operations at low concurrency levels (16 processors or less) for 16 priorities. The two unbounded algorithms SingleLock and HuntEtAl display a linear increase in latency as concurrency rises, with the algorithm of Hunt et al. performing slightly better. These results are expected and match those of [17]. The queue contains relatively few items at any time, so the fact that the HuntEtAl algorithm allows more parallelism in the access to the queue itself is of little benefit – processors spend most of their time trying to acquire the initial lock. The SkipList algorithm fairs slightly better due to its efficient insertion method and delete buffer. The right-hand-side of Figure 6 allows a closer look at the algorithms with lower latency. All four have roughly equal slopes, but the SimpleLinear method with its low overhead, especially for insertion is the clear leader. LinearFunnels is about 2-3 times slower, mostly due to the overhead of funnels versus MCS locks (see [33] for details). The two tree based methods, FunnelTree and SimpleTree, have nearly the same latency, with FunnelTree being about 10% slower, and both methods about 40–50% slower than SimpleLinear. Though both SimpleTree and HuntEtAl contain a serial bottleneck the former is faster since it only performs a `FaI` or `BFaD` at the root, while the latter must acquire another lock during the initial critical section.

At low concurrency levels overhead is the critical performance factor. As the number of processors increases, the effects of contention, serialization and parallelism play an ever increasing role. This is evident in the graph in Figure 7 which compares the four most promising methods: SimpleLinear, SimpleTree, LinearFunnels, and FunnelTree for levels of concurrency between 2 and 256 processors at 16 priorities. SimpleTree is the slowest method at high concurrency levels since contention at the root increases the time to perform counter operations there. SimpleLinear is fastest till about 32 processors and still performs relatively well at 128 processors. This might seem surprising till we notice that insertions are independent and have low overhead, and deletions perform at most 16 reads and only attempt to lock promising bins. Only at 128 processors does contention at individual bins begin to play a major role. The LinearFunnels algorithm

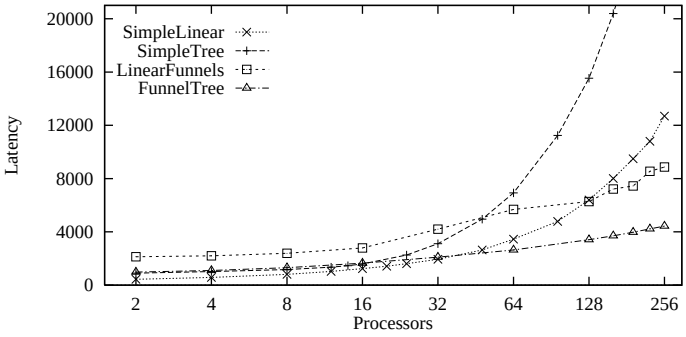

Figure 7: Latency of more scalable priority queue implementations with 16 priorities (*top*) at high concurrency levels.

		SimpleLinear			SimpleTree			LinearFunnels			FunnelTree		
N	P	Ins.	Del.	All	Ins.	Del.	All	Ins.	Del.	All	Ins.	Del.	All
16	16	0.4	1.9	1.2	1.3	1.8	1.6	0.6	4.9	2.8	1.4	1.8	1.6
128	16	0.3	5.3	2.8	1.8	1.9	1.8	0.6	28.3	14.0	1.8	2.1	1.9
16	64	0.9	5.3	3.4	5.7	8.1	6.9	0.8	9.8	5.7	2.0	3.2	2.6
128	64	0.4	8.1	4.6	6.0	7.8	6.9	0.6	34.7	16.1	2.6	3.5	3.0
16	256	3.9	19.3	12.7	29.3	41.1	35.3	0.9	17.4	8.9	3.3	5.8	4.4
128	256	0.6	21.0	10.2	29.3	40.9	34.7	0.8	66.6	31.5	4.2	6.7	5.1

Figure 8: Latencies (in thousands of cycles) for `insert` and delete-min in several bounded range priority queue implementations. N is the number of priorities and P is the number of processors.

avoids this contention and increases parallelism at a price of higher overhead, this pays off only at 128 processors and above. The FunnelTree method traverses less funnels and starts paying-off earlier, at 64 processors, where it becomes the performance leader. As concurrency increases, the parallelism inherent in the funnels and their ability to avoid contention cause the gap between FunnelTree and the other methods to widen, so that at 256 processors this method is 8 times faster than SimpleTree and 3 times faster then SimpleLinear.

Figure 8 offers a break-down of latency figures into cost of `insert` and `delete-min`. Generally, increasing the number of priorities increases the size of the data structure. This may mean more work is required to update the data structure – increasing latency, conversely, since processor are spread over more memory there is less contention – reducing latency. In SimpleLinear, for example, at 16 processors work is the dominant factor and going from 16 to 128 priorities increases latency by over 220%. The same change at 256 processors (when contention is more important) reduces latency by 12%. Funnel based methods are less susceptible to contention, so the added overhead of more funnels (to support more priorities) is the dominant factor in determining performance. For both tree based methods, `insert` is cheaper than `delete-min` because on average insertions update half as many counters along the path to the root.

The graphs in Figure 9 demonstrate the behavior of different algorithms as the range of priorities goes from 2 to 512 at 64 and 256 processors. In SimpleLinear the interplay between increased work (which affects mostly `delete-min`) and decreased contention (benefits `insert`) explains the "u" shaped curve. LinearFunnels slows linearly with each added priority since the overhead of each new funnel outweighs the saving in contention. The SimpleTree algorithm shows an almost constant latency for 64 processors (it was

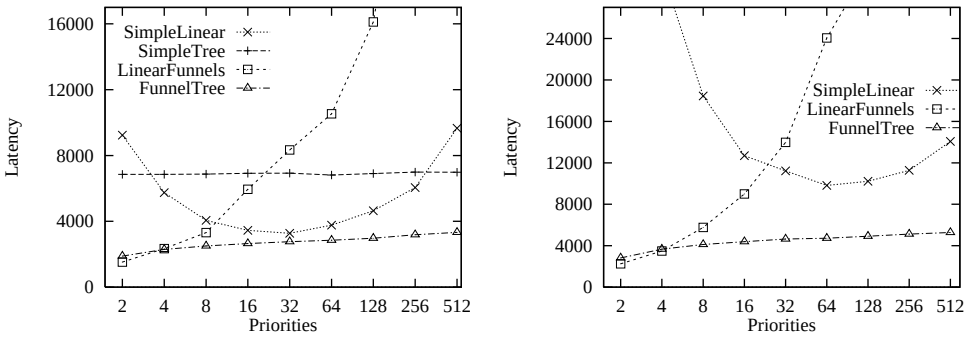

Figure 9: Latency of more scalable priority queue implementations with varying number of priorities at 64 processors (*left*) and 256 processors (*right*).

off the graph for 256), since at this level of concurrency latency is determined almost exclusively by the time to pass through the root. The growth in latency of FunnelTree is less than logarithmic since deeper nodes have less traffic then higher ones. Thus the funnels shrink, or we use MCS locks there. Together the two graphs in Figure 9 show that at high concurrency levels the only method that consistently works well and is better for nearly all priorities, is FunnelTree. Only when faced with a very small range of priorities should one consider using the LinearFunnels method. In both cases we see that the funnel based methods offer the best scalable solution to implementing parallel bounded range priority queues.

5 Conclusions

We believe the results presented in this paper are very promising. They show that simple, almost trivial, algorithms can sometimes work quite well in practice. More importantly, they show that when performance degrades due to the appearance of hot spots, it is possible to remedy the implementation using a general technique such as our combining funnels. This is an indication that our combining funnels approach is an effective embeddable technique. If borne out by further research and real applications on large scale machines, it could mean that future parallel data structures could be constructed using the well understood techniques of today, with contention reducing parallelization methods used to "massage" the trouble spots.

6 Acknowledgments

We would like to thanks our colleague Dan Touitou for his numerous comments, and his suggestion of the tree based priority queue structure [30] we used for the SimpleTree and FunnelTree stuctures. We further wish to thank the anonymous referees.

References

[1] A. Agarwal, D. Chaiken, K. Johnson, D. Krantz, J. Kubiatowicz, K. Kurihara, B. Lim, G. Maa, and D. Nussbaum. The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor. In *Scalable Shared Memory Multiprocessors*, Kluwer Academic Publishers, 1991. Also as MIT Technical Report MIT/LCS/TM-454, June 1991.

[2] T. Agerwala, J.L. Martin, J.H. Mirza, D.C. Sadler, D.M. Dias and M. Snir. SP2 system architecture. *IBM Systems Journal*, 34(2):152–184, 1995.

[3] J. Aspnes, M.P. Herlihy and N. Shavit. Counting Networks. *Journal of the ACM*, Vol. 41, No. 5, September 1994, pp. 1020-1048.

[4] R. Ayani. Lr-algorithm: concurrent operations on priority queues. In *Proceedings of the 2nd IEEE Symposium on Parallel and Distributed Processing* pp. 22-25, 1991.

[5] G. Alverson and S. Kahan – TERA Computer Company. Personal communication, March 1998.

[6] J. Biswas and J.C. Browne. Simultaneous Update of Priority Structures In *Proceedings of the 1987 International Conference on Parallel Processing*, August 1987, pp. 124–131.

[7] J. Biswas and J.C. Browne. Data Structures for Parallel Resource Management. In *IEEE Transactions on Software Engineering*, 19(7), pp. 672-686, 1993.

[8] J. Boyar, R. Fagerberg and K.S. Larsen. Chromatic Priority Queues. Technical Report, Department of Mathematics and Computer Science, Odense University, PP-1994-15, May 1994.

[9] E.A. Brewer, C.N. Dellarocas. PROTEUS *User Documentation*. MIT, 545 Technology Square, Cambridge, MA 02139, 0.5 edition, December 1992.

[10] E.A. Brewer, C.N. Dellarocas, A. Colbrook and W.E. Weihl. PROTEUS: A High-Performance Parallel-Architecture Simulator. MIT Technical Report /MIT/LCS/TR-561, September 1991.

[11] G. Della-Libera. Reactive Diffracting Trees. Master's Thesis, Massachusetts Institute of Technology, 1997.

[12] G. Della-Libera and N. Shavit. Reactive Diffracting Trees. In *Proceedings of the 9th Annual Symposium on Parallel Algorithms and Architectures (SPAA)*, June 1997.

[13] J.R. Goodman, M.K. Vernon, and P.J. Woest. Efficient Synchronization Primitives for Large-Scale Cache-Coherent multiprocessors. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 64–75, April 1989.

[14] A. Gottlieb, R. Grishman, C.P. Kruskal, K.P. McAuliffe, L. Rudolph, and M. Snir. The NYU Ultracomputer - designing an MIMD parallel computer. *IEEE Transactions on Computers*, C-32(2):175–189, February 1984.

[15] A. Gottlieb, B.D. Lubachevsky, and L. Rudolph. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *ACM Transactions on Programming Languages and Systems*, 5(2):164–189, April 1983.

[16] Q. Huang. An Evaluation of Concurrent Priority Queue Algorithms. Technical Report, Massachusetts Institute of Technology, MIT-LCS/MIT/LCS/TR-497, May 1991.

[17] G.C. Hunt, M.M. Michael, S. Parthasarathy and M.L. Scott. An Efficient Algorithm for Concurrent Priority Queue Heaps. In *Information Processing Letters*, 60(3):151–157, November 1996.

[18] T. Johnson. A Highly Concurrent Priority Queue Based on the B-link Tree. Technical Report, University of Florida, 91-007. August 1991.

[19] A. Karlin, K. Li, M. Manasse and S. Owicki. Empirical Studies of Competitive Spinning for A Shared Memory Multiprocessor. In *13th ACM Symposium on Operating System Principles (SOSP)*, pp. 41–55, October 1991.

[20] C.P. Kruskal, L. Rudolph, and M. Snir. Efficient synchronization on multiprocessors with shared memory. In *Fifth ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, August 1986.

[21] 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, July 1990.

[22] B.H. Lim and A. Agarwal. Reactive Synchronization Algorithms for Multiprocessors. In *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pp. 25–35, 1994.

[23] B.H. Lim and A. Agarwal. Waiting Algorithms for Synchronization in Large-Scale Multiprocessors. In *ACM Transactions on Computer Systems*, 11(3):253–294, August 1993.

[24] J.M. Mellor-Crummey and M.L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, Feb 1991.

[25] G.H. Pfister and A. Norton. 'Hot Spot' contention and combining in multistage interconnection networks. *IEEE Transactions on Computers*, C-34(11):933–938, November 1985.

[26] L. Rudolph. Personal communication regarding StarT-NG system. `http://csg-www.lcs.mit.edu:8001/StarT-`, MIT, 1998.

[27] W. Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. In *Communications of the ACM*, 33(6):668–676, June 1990.

[28] W. Pugh. Concurrent Maintenance of Skip Lists. Technical Report, Institute for Advanced Computer Studies, Department of Computer Science, University of Maryland, College Park, CS-TR-2222.1, 1989.

[29] V. N. Rao and V. Kumar. Concurrent access of priority queues. *IEEE Transactions on Computers* 37, 1657-1665, December 1988.

[30] D. Touitou. Personal communication, June 1996.

[31] N. Shavit, and D. Touitou. Elimination Trees and the Construction of Pools and Stacks In *Proceedings of the 7th Annual Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 54-63, July 1995.

[32] N. Shavit and A. Zemach. Diffracting Trees. *ACM Transactions on Computer Systems*, 14(4):385–428, November 1996.

[33] N. Shavit and A. Zemach. Combining funnels. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing*, pages 61–70, Puerto Vallarta, Mexico, June 28th – July 2nd 1998.

[34] G. Alverson, S. Kahan and R. Korry. Processor Management in the Tera MTA Computer System. Available via `ftp` from `http://www.tera.com/www/archives/library/processor.html`.

[35] P.C. Yew, N.F. Tzeng, and D.H. Lawrie. Distributing Hot-Spot Addressing in Large-Scale Multiprocessors. *IEEE Transactions on Computers*, C-36(4):388–395, April 1987.

[36] Y. Yan and X. Zhang. Lock Bypassing: An Efficient Algorithm for Concurrently Accessing Priority Heaps. *ACM Journal of Experimental Algorithmics*, vol. 3, 1998. `http://www.jea.acm.org/1998/YanLock/`

[37] A. Zemach. Distributed Coordinated Data Structures. Ph.D. dissertation, Tel-Aviv University, March 1998.

A Appendix: Pseudo Code For Some Priority Queue Implementations

The way in which we make sure all trees are homogeneous is by forcing all processors at a given funnel layer to be roots of trees of the same size. Processors enter the first layer of the funnel and are roots of trees of size 1 (themselves only). When processors with like operations (both perform `BFaD` or `FaI`) combine one becomes the root of a tree of size 2, and advances to the next funnel layer. If operations are reversing (one is a `BFaD` and the other a `FaI`) they eliminate and exit the funnel at once. Processors only advance to the next funnel layer after they have collided and combined with a processor who is the root of a tree with the same size and operation. Colliding with trees of the same size but opposing operations yields an elimination. Collisions between trees of different sizes are not allowed. This scheme keeps all processors that are roots of trees of size 2^i at the i-th layer and makes sure collisions between trees of the same size are not rare. In the case of elimination there is still some subtle interaction with the bounds checking, but accounting for its effects can be made simple by assuming the operations from the trees are interleaved, so that the counter never moves more than one away from its original position.

The following is a walk-through of the code in Figure 10. We assume per-processor public data is accessed through a `my` pointer, and that `obj` encapsulates data and functions specific to the object.

The fields of `my` are:

`Sum:` The current sum of all operations combined with.

`Location:` The address of the funnel, and the layer number within the funnel, that this processor is currently traversing. This value is set to `NULL` if the processor is not in any funnel and thus is unavailable for collisions.

`Result:` Is set when the result of the operation is known. Holds two values, the first is either `ELIM` or `COUNT` which indicates whether the operation was eliminated part-way or acquired the `obj`. The second is the operation's return value.

`Adaption_factor:` An indication of load, which is used to modify the width of the funnel layers in response to load variations.

The fields of `obj` are:

`MainVal:` The current value of the counter.

`levels:` The number of funnel layers.

`attempts:` The number of collision attempts a processor should try before accessing `MainVal`.

`width[]:` An array holding the respective width values of funnel layers.

`layer[][]:` A two dimensional array holding the elements of each funnel layer.

`spin[]:` An array holding the the respective amounts of time a processor should delay at each layer.

Lines 1–2 set up the data structures for the operation. Lines 6–24 contain the collision code. A processor p picks a random layer location, swaps its `my` pointer for the one written there, q, and attempts to collide by locking itself and q (lines 8–11). If the collision succeeds, q is either added to p's list if their operation are the same (lines 19–21), or an elimination occurs and p short-cuts to get the counter's value (lines 12–18). If p discovers that it has been collided with (lines 25–26) it goes on to wait for a value (line 39). Every `obj->attempts` collision attempts p tries to perform a bounded decrement operation on the central counter using a compare-and-swap operation (line

```
 int Bounded_Fetch_And_Decrement(fna_obj *obj)
 {
 1  my->Sum = total = -1
 2  d=0                                  // initially at depth 0
 3  my->Location = < obj , d >           // of funnel "obj"
 4  mainloop: while(1) {
 5    for(n=0; n<obj->attempts && d<obj->levels; n++) {
 6      wid = my->Adaption_factor * obj->width[d]
 7      r = random(0 , wid)
 8      q = SWAP(obj->layer[d][r], my)          // read partner
 9      if( q == NOBODY ) continue
10      if( CaS(my->Location, <obj, d> , EMPTY) )    // attempt
11        if( CaS(q->Location, <obj, d> , EMPTY) ) { // collision
12          if ( q->Sum + my->Sum == 0 ) {      // opposites?
13            val = obj->MainVal                // short-cut
14            if(val==BOT) val++                // to counter
15            q->Result = < ELIM , val-1 >      // and
16            my->Result = < ELIM, val >        // eliminate
17            break mainloop
18          }
19          my->Sum += q->Sum                   // combine
20          my->Location = < obj, ++d >         // advance layer
21          append q to list-of-children        // add q as child
22          n = 0
23        }
24        else my->Location =  < obj, d >
25      for(i=0;i< obj->spin[d]; i++)       // delay to combine
26        if(my->Location != <obj, d>) break mainloop
27    }
28    if( CaS(my->Location, <obj, d>, EMPTY) ) {
29      val = obj->MainVal                  // calc new value
30      new = val + my->Sum                 // for central
31      if ( new < BOT ) new = BOT
32      if( CaS(obj->MainVal, val, new)) {
33        my->Result = < COUNT , val>       // updated central
34        break mainloop                    // distribute results
35      }
36      my->Location = <obj, d>
37    }
38  }
39  while( my->Result == EMPTY) ;      // wait for result
40  < event, val > = my->Result
41  foreach q in list-of-children {    // iterate over children
42    if( event == ELIM )              // if eliminated
43      q->Result = < ELIM, val >      //  all get same result
44    else {
45      q->Result = < COUNT , val + total >
46      total += q->Sum
47    }
48  }
 }
```

Figure 10: Code for bounded fetch-and-decrement.

32). If this succeeds p, moves to the distribution phase. The distribution phase begins with code that waits for a result (lines 39–40), and when it arrives p iterates over all its children, handing out a value to each of them by setting their `my->result` field (lines 41-46). When distributing values it is important to know whether an elimination has occurred since then we distribute the same value to all operations in the tree (due to the interleaved ordering of the operations: inc, dec, inc, and so on).

The way in which we adaption fits into the funnel framework is by altering the width of the funnel layers. In line 7 of Figure 10 a processor picks a random location in its current layer not from the entire width `obj->width[l]` but from a fraction `my->Adaption_factor` of this width. By changing the value of `Adaption_factor` between 0 and 1, processors can, at a purely local level, determine the size of the funnel they will use. The decision is based on the actual load encountered, measured as the ratio of successful collisions to collision attempts (passes through the loop in lines 5–27). See [33] for more details.

B Appendix; Priority Queues in a Parallel Setting

In a sequential setting a priority queue is a data structure that supports two operations: insert(x) which inserts the element x into the queue, where x has a certain priority $pri(x)$ from some well ordered set; and x =delete-min() which removes and returns the element x whose priority is the smallest currently in the queue. A priority queue is said to have a *bounded range* if the set of possible priorities of elements is finite. In this case we map the priorities unto a the set of integers $[1, N]$ and say that the queue is of range N. Notice that the above definition refers to "the smallest priority element currently in the queue." This definition is not precise enough for the concurrent setting where multiple operations may overlap in time, since it is not clear when an element is really "in the queue."

We will use two standard consistency conditions to capture the behavior of our queues in a parallel environment, giving meaning to the notion of "the smallest priority element currently in the queue." The first condition, due to Herlihy and Wing, is linearizability [21]. A data structure is said to be linearizable if for every concurrent execution, one can associate with every operation a single time point within its actual execution interval at which the operation is said to take place, such that the associated execution is a valid sequential execution. Put differently, by compressing operations to a single point in time we can place them all on a single time-line, and the resulting execution must then conform to the object's sequential specification. Linearizability for a priority queue means that one can order all insert and delete-min operations consistently with the real-time order in which they occured, so that the set of minimal priorities according to this order is well defined.

Another consistency condition that is semantically weaker but allows more computational efficiency is Aspnes et. al's quiescent consistency [3]. An object is said to be quiescent at time t if all operations which began before t have completed, and no operation begins at t. An object is quiescently consistent if any operation performed on it can be compressed into a single point in time somewhere between the latest quiescent point before it began and the earliest quiescent point after it ends, so that it meets the object's sequential specification.[6] The main difference between quiescent consistency and linearizability is that quiescent consistency does not require preservation of real-time order: if an operation A completes before another operation B start, and there is an operation C which overlaps both A and B, B may be reordered before A. However, the priority guarantees of quiescently consistent priority queues are quite strong. Assume the priority queue Q is quiescent at time t_0 and contains the set of elements E, each

[6] Though one can define infinite executions with no quiecense point, the fact that a quiescent point may appear at any time in the execution imposes strong restrictions of the allowable implementations.

```
elem_type PriQ[MAXSIZE]
int NumElems
lock_t PQlock

insert(elem_type e)
{
  acquire(PQlock)
  Priq[NumElems++]=e
  //propagate e up the heap
  //using standard heap algorithm
  release(PQlock)
}

delete-min()
{
  acquire(PQlock)
  save = PriQ[0]
  Priq[0]=PriQ[--NumElems]
  //propagate PriQ[0] down the heap
  //using standard heap algorithm
  release(PQlock)
  return save
}
```

```
insert(elem_type e)
{
  acquire(PQlock)
  i = bit-reverse(NumElems++)
  acquire(PriQ[i].lock)
  PriQ[i]=e
  release(PQlock)
  //propagate e up the heap
  //using local locks at each node
}

delete-min()
{
  acquire(PQlock)
  acquire(PriQ[0].lock)
  save = PriQ[0]
  i = bit-reverse(--NumElems)
  acquire(PriQ[i].lock)
  Priq[0]=PriQ[i]
  release(PQlock)
  //propagate e down the heap
  //using local locks at each node
  return save
}
```

Figure 11: **SingleLock:** code for heap based priority queue using a single lock (*left*). **HuntEtAl:** code for priority queue algorithm using the algorithm of Hunt et al [17] (*right*). Error handling not shown.

```
skip_list *Head
skip_list Link[MAXPRI]
bin DelBin
lock_t DelLock

insert(elem_type e)
{
  bin-insert(Link[e.pri].bin , e)
  if(Link[e.pri].threaded == 0) {
     //insert Link[e.pri] into the
     //skip list using Pugh's algorithm
     Link[e.pri].threaded = 1
  }
}
```

```
delete-min()
{
  do{
    e = bin-delete(DelBin)
    if ( e == NULL && acquired(DelLock)) {
      s = Head
      //delete the first element
      //of the skip list using
      //Pugh's algorithm
      DelBin = s->bin
      release(DelLock)
    }
  }while( e==NULL)
}
```

Figure 12: **SkipList:** code for bounded range priority queue algorithm using skip lists. Error handling not shown. The operation `acquired` does not block, it merely attempts to acquire the lock and returns an indication of success.

with a different priority. If k delete-min operations are performed on Q ending at time t_1, the returned elements are exactly the k elements of Q with the smallest priority, which we denote by $\mathrm{Min}_k(E)$. This is exactly the same as with a linearizable priority queue. However, if between t_0 and t_1 there also occur operations which insert a new set of elements I into Q, then the elements returned by the delete-min operations will be from the set $\mathrm{Min}_k(E) \cup \mathrm{Min}_k(E \cup I)$. [7] In other words, if during a sequence of deletes new elements enter the queue, the set of returned minimal values may include minimum elements from the joint set of new and old enqueued values, yet there will be no exact timeline on which dequeue and enqueue operations can be ordered. Theoretically, this means that if new items of the same or smaller priority than the old minimum item are enqueued, they may be deleted in its place and it might remain in the queue. In many real-world sytems this is not a problem, and might even be considered an advantage since newly arrived smaller priority items can be serviced ahead of older low priority ones.

[7] If some elements have the same priority, the formal definitions become somewhat more complex as they must capture the uncertainty of choosing between elements with the same priority. Define $\mathrm{Pri}_p(E) = \{x \in E | pri(x) \leq p\}$, let $p1$ be the largest priority for which $|\mathrm{Pri}_{p1}(E)| < k$ and $p2$ be the smallest priority for which $|\mathrm{Pri}_{p2}(E)| \geq k$. Thus in the case where k delete-min operations are performed on a priority queue with no intervening insertion operations, the elements returned will be those of the set $\mathrm{Pri}_{p1}(E)$, and a subset of size $k - |\mathrm{Pri}_{p1}(E)|$ of the elements of priority $p2$.