Fast and Scalable, Lock-free k-FIFO Queues

Christoph M. Kirsch, Michael Lippautz, and Hannes Payer

University of Salzburg
firstname.lastname@cs.uni-salzburg.at

Abstract. We introduce fast and scalable algorithms that implement bounded- and unbounded-size lock-free k-FIFO queues on parallel, shared memory hardware. Logically, a k-FIFO queue can be understood as queue where elements may be dequeued out-of-order up to $k-1$, or as pool where the oldest element is dequeued within at most k dequeue operations. The presented algorithms enable up to k enqueue and k dequeue operations to be performed in parallel. Unlike previous designs, however, the algorithms also implement linearizable emptiness (and full) checks without impairing scalability. We show experimentally that there exist optimal and robust k that result in best access performance and scalability. We then demonstrate that our algorithms outperform and outscale all state-of-the-art concurrent pool and queue algorithms that we considered in all micro- and most macrobenchmarks. Moreover, we demonstrate a prototypical controller which identifies optimal k automatically at runtime achieving better performance than with any statically configured k.

1 Introduction

We are interested in the design and implementation of fast concurrent pools and queues whose access performance scales with the number of available processing units on parallel, shared memory hardware. We introduce two algorithms that implement bounded-size (BS) and unbounded-size (US), lock-free k-FIFO queues with linearizable emptiness (and full) check. The BS algorithm maintains a bounded-size array of elements that is dynamically partitioned into segments of size k called k-segments while the US algorithm maintains an unbounded list of k-segments. Thus up to k enqueue and k dequeue operations may be performed in parallel. The US algorithm simplifies for $k = 1$ to an algorithm that implements a lock-free FIFO queue similar to the lock-free Michael-Scott FIFO queue (MS) [1] but without a sentinel node. See Section 3 for more details.

The idea of k-segments has first appeared in the so-called Segment Queue (SQ) [2]. The US algorithm uses k-segments but improves upon SQ in performance and scalability through less overhead and reduced contention by performing removal of elements from k-segment slots lazily. Dequeue operations only perform costly compare-and-swap operations when used k-segments slots are actually found. The enhancement is necessary to obtain positive scalability on the hardware and for the workloads that we considered. Moreover, both k-FIFO queue algorithms implement a linearizable emptiness check which SQ does not. In other words, upon dequeueing attempts SQ may not always find and retrieve elements but instead return nothing even though the queue is not empty. A linearizable emptiness check may be required for implementing application requirements such as termination. We discuss the relation to SQ in detail in Section 4.

In Section 5, before presenting a detailed performance analysis of our algorithms relative to a variety of concurrent pool and queue algorithms, we show experimentally that there exist optimal and robust k that result in best performance and scalability. Interestingly, performance generally increases with k but only up to a certain point which is determined by a tradeoff between degree of parallelism and management overhead. Our algorithms outperform and outscale all other algorithms that we considered in almost all threading and contention scenarios. Finally, we discuss a prototypical controller that adjusts k dynamically and automatically at runtime outperforming any static and manual k configuration on the workload that we considered.

2 k-FIFO Queue Sequential Specification

A k-FIFO queue is a restricted out-of-order k-relaxation of a FIFO queue as defined in the framework for quantitative relaxation of concurrent data structures [3]. We informally discuss the sequential specification of a k-FIFO queue.

Let Σ bet the set of queue methods with input and output values defined as

$$\Sigma = \{\texttt{enq(x)}, \texttt{deq(x)} \mid x \in D\} \cup \{\texttt{deq(null)}\}$$

where D is the set of elements that can be enqueued and dequeued from the queue and `deq(null)` represents a dequeue returning empty. We refer to sequences in Σ^* as queue sequences.

The sequential specification of a FIFO queue is the set $S \subseteq \Sigma^*$ that contains all valid FIFO queue sequences. Informally, valid FIFO queue sequences are sequences where elements are enqueued in the same order as they are dequeued. Furthermore, a `deq(null)` only happens if the queue is empty at the time of `deq(null)`, i.e., every element which gets enqueued before `deq(null)` also gets dequeued before `deq(null)`. For example, the queue sequence

$$\texttt{enq(a)enq(b)deq(a)enq(c)deq(b)deq(c)}$$

belongs to S whereas

$$\texttt{enq(a)enq(b)deq(b)enq(c)deq(a)deq(c)}$$

does not.

A restricted out-of-order k-relaxation of a FIFO queue is the set $S_k \subseteq \Sigma^*$ containing all sequences with a distance of at most k to the sequential specification S of the queue. Informally, the distance is the number of elements overtaking each other in the queue, i.e., an element e may overtake at most $k-1$ elements and may be overtaken by at most $k-1$ other elements before it is dequeued. A 1-FIFO queue thus corresponds to a regular FIFO queue. The above example sequences are therefore within the specifications of a 1-FIFO and a 2-FIFO queue, respectively. We show in Section 3.2 that our k-FIFO queue algorithms indeed implement k-FIFO queues.

3 k-FIFO Queue Algorithms

We present the algorithms of the lock-free bounded-size (BS) and unbounded-size (US) k-FIFO queues for $k > 0$. The pseudo code of the algorithms is depicted in Listing 1.1. The occurrence of the ABA problem is made unlikely through version numbers (also

known as ABA counters). We refer to values enhanced with version numbers as atomic values. We use compare-and-swap (CAS) operations to atomically swap in values at given locations. The gray highlighted code is only used in the BS version. We present the general idea of the algorithms followed by a detailed discussion of the BS algorithm. We then discuss the US algorithm by outlining its differences to the BS algorithm and finally show informally that both algorithms are linearizable with respect to the *k*-FIFO specification.

FIFO queues are usually implemented using head and tail pointers, where elements are dequeued at the head and enqueued at the tail pointer. When implementing a linearizable FIFO queue, these head and tail pointers may become scalablity bottlenecks. The principle idea of the BS and US *k*-FIFO queue algorithms is to reduce contention on the head and tail pointers by maintaining an array (BS) or a list (US) of so-called *k*-segments each consisting of *k* slots, instead of maintaining an array or list of plain queue elements. A slot may either point to `null` indicating an empty slot or may hold a so-called item, which is our implementation concept for enqueued elements. An enqueue operation is served by the tail *k*-segment and a dequeue operation is served by the head *k*-segment. Hence, up to *k* enqueue and *k* dequeue operations may be performed in parallel.

The BS *k*-FIFO queue algorithm is based on an array of atomic values of a given `size`. For simplicity we restrict `size` to be a multiple of *k*. The queue `tail` and queue `head` pointers are also atomic values. Both initially point to the slot at index zero.

The `enqueue` method is depicted in Listing 1.1. Given an `item` representing an element to be enqueued, the method returns `true` when the `item` is successfully inserted and `false` when the queue is full. First the method tries to find an empty slot in the tail *k*-segment which is located in between the indices $[\texttt{tail}, \texttt{tail}+k[$ using the `find_empty_slot` method (line 5). The `find_empty_slot` method randomly selects an index in between $[\texttt{tail}, \texttt{tail}+k[$ and then linearly searches for an empty slot starting with the selected index wrapping around at index $\texttt{tail}+k-1$. Afterwards the `enqueue` method checks if the *k*-FIFO queue state has been consistently observed by checking whether `tail` changed in the meantime (line 6) which would trigger a retry. If an empty slot is found (line 7) the method tries to insert the `item` at the location of the empty slot using a `CAS` operation (line 9). If the insertion is successful the method verifies whether the insertion is also valid by calling the `committed` method (line 10), as discussed below. The `enqueue` method retries in the following scenarios. If no empty slot is found in the current tail *k*-segment the `enqueue` method tries to increment `tail` by *k* using `CAS` (line 19) and then retries. If `tail` cannot be incremented without overtaking `head` (line 13) and the *k*-segment to which `head_old` points is empty (line 14) the method tries to increment `head` by *k* using `CAS` (line 18). If this *k*-segment is not empty and `head_old` did not change in the meantime (line 15) the queue is full and `false` is returned (line 16).

The `committed` method (line 21) validates an insertion. It returns `true` when the insertion is valid and `false` when it is not valid. An insertion is valid if the inserted item already got dequeued at validation time by a concurrent operation (line 22, 30, 36) or the tail *k*-segment where the item was inserted is in between the current head *k*-segment and the current tail *k*-segment but not equal to the current head *k*-segment

(line 27). If the tail k-segment where the item was inserted is not in between the current head k-segment and the current tail k-segment (line 29) the method tries to undo the insertion using `CAS` (line 30). If the tail k-segment where the item was inserted is equal to the current head k-segment a race with concurrent dequeueing threads may occur which may not have observed the insertion and may try to advance the `head` pointer in the meantime. This would result in loss of the inserted item. To prevent that the method tries to increment the version number in the `head` atomic value using `CAS` (line 34). If this fails a concurrent dequeue operation may have changed `head` which would make the insertion potentially invalid. Hence after that the method tries to undo the insertion using `CAS` (line 36). The `committed` method returns `false` (line 38) if the insertion was undone in any of these cases.

The `dequeue` method is depicted in Listing 1.1. It returns an `item` if the queue is not empty, and returns `null` if the queue is empty. Similarly to the `enqueue` method the `dequeue` method first tries to find an item in between the indices $[\texttt{head}, \texttt{head}+k[$ using the `find_item` method (line 43). The `find_item` method randomly selects an index in between $[\texttt{head}, \texttt{head}+k[$ and then linearly searches for an item starting with the selected index wrapping around at index $\texttt{head}+k-1$. Afterwards the `dequeue` method checks if the queue state has been consistently observed by checking whether `head` changed in the meantime (line 45) which would trigger a retry. If an item was found (line 46) the method first checks whether `head` equals `tail` (line 47), checking whether the queue only consists of a single k-segment. If this is the case the method tries to increment `tail` by k to prevent starvation of items in the queue and to provide a linearizable emptiness check. Afterwards the method tries to remove the item using `CAS` (line 50) and returns it if the removal was successful (line 51). If the `CAS` fails due to a concurrent dequeue, a retry is performed. If no item is found, `head` equals `tail`, and `tail` did not change in the meantime `null` is returned indicating an empty queue (line 53). This is enough to show that the queue has been empty at the linearization point (see correctness part below). If `tail` did change in the meantime, the operation tries to increment `head` by k (line 55) and retries, since no other operation could have enqueued an item into this k-segment anymore.

Note that to hold n items the BS k-FIFO queue has to consist of at least $\lceil \frac{n}{k} \rceil \times k + k$ atomic values. The additional segment, introduced by `dequeue` (line 47, 48), is necessary to avoid starvation of items which is possible if enqueue and dequeue operations work on the same segment.

3.1 US k-FIFO Queue Algorithm

The US k-FIFO queue algorithm differs from the BS version in the implementation of the `committed`, `advance_tail`, and `advance_head` methods. The gray highlighted code in Listing 1.1 is not used in the US version since there is no full state. Hence the `enqueue` method always returns `true`.

An enqueue operation is served by the tail k-segment. When this k-segment is full, a new k-segment is added to the `tail`. A dequeue operation is served by the head k-segment. When this k-segment is empty it is removed from the k-segment queue except if it is the only k-segment in the queue.

Listing 1.1. Lock-free *k*-FIFO queue algorithms

```
1 enqueue(item):
2   while true:
3     tail_old = get_tail();
4     head_old = get_head();
5     item_old, index = find_empty_slot(tail_old, k);
6     if tail_old == get_tail():
7       if item_old.value == EMPTY:
8         item_new = atomic_value(item, item_old.version + 1);
9         if CAS(&tail_old->segment[index], item_old, item_new):
10          if committed(tail_old, item_new, index):
11            return true;
12      else:
13        if tail_old.value + k == head_old.value:
14          if segment_not_empty(head_old, k):
15            if head_old == get_head():
16              return false;
17          else:
18            advance_head(head_old, k);
19        advance_tail(tail_old, k);
20
21 bool committed(tail_old, item_new, index):
22   if tail_old->segment[index] != item_new:
23     return true;
24   head_current = get_head();
25   tail_current = get_tail();
26   item_empty = atomic_value(EMPTY, item_new.version + 1);
27   if in_queue_after_head(tail_old, tail_current, head_current):
28     return true;
29   else if not_in_queue(tail_old, tail_current, head_current):
30     if !CAS(&tail_old->segment[index], item_new, item_empty):
31       return true;
32   else: //in queue at head
33     head_new = atomic_value(head_current.value, head_current.version + 1);
34     if CAS(&head, head_current, head_new):
35       return true;
36     if !CAS(&tail_old->segment[index], item_new, item_empty):
37       return true;
38   return false;
39
40 item dequeue():
41   while true:
42     head_old = get_head();
43     item_old, index = find_item(head_old, k);
44     tail_old = get_tail();
45     if head_old == get_head():
46       if item_old.value != EMPTY:
47         if head_old.value == tail_old.value:
48           advance_tail(tail_old, k);
49         item_empty = atomic_value(EMPTY, item_old.version + 1);
50         if CAS(&head_old[index], item_old, item_empty):
51           return item_old.value;
52       else:
53         if head_old.value == tail_old.value && tail_old.value == get_tail():
54           return null;
55         advance_head(head_old, k);
```

Any lock-free FIFO queue algorithm may be used to implement the queue of *k*-segments. We developed a lock-free FIFO queue which performs better than the lock-free Michael-Scott FIFO queue (MS) [1] when used as *k*-segment queue in the US *k*-FIFO queue algorithm. Our implemented *k*-segment queue always contains at least one usable *k*-segment, even if this *k*-segment is empty, enabling fast and direct access to the head and tail *k*-segments [4].

The US algorithm simplifies for $k = 1$ to an algorithm that implements a lock-free FIFO queue similar to the MS algorithm but without a sentinel node. In contrast to MS,

the empty US 1-FIFO queue contains an empty 1-segment in which the first enqueued element is stored. Subsequent dequeue operations may lead to a queue with an empty 1-segment at the head but only because 1-segments are removed lazily (which may also be done eagerly avoiding empty 1-segments altogether in a non-empty queue).

3.2 Correctness

Proposition 1. *The k-FIFO queue algorithms are linearizable with respect to the sequential specification of a k-FIFO queue.*

Proof. Without loss of generality, we assume that each item enqueued in and dequeued from the queue is unique. Furthermore we call a segment s' reachable from a k-segment s, if either $s' = s$, or s' is reachable from $\texttt{s} \rightarrow \texttt{next}$ (US), or s is within the range $[\texttt{head}, \texttt{tail}]$ (BS). Similar, a k-segment s is removed from the queue if it is not reachable from head (US), or not within the range of k-segments $[\texttt{head}, \texttt{tail}]$ anymore (BS).

An item is logically contained in the queue, if `enqueue(i)` has already `committed` and there exists a reachable k-segment containing a segment whose value is `i`. Note that only having a slot containing the item is not enough to guarantee that the item is logically in the queue, because the slot could be in a k-segment that is not reachable anymore (because it has been removed).

We identify linearization points [5] of the `enqueue` and `dequeue` methods depicted in Listing 1.1 to show that the sequential history obtained from a concurrent history by ordering methods according to their linearization points is in the sequential specification of a k-FIFO queue. The linearization point of `enqueue` that inserts an item is the successfully executed `CAS` inserting the item (line 9) if the following call to `committed` validates it and therefore returns `true`. Additionally, the queue may also be full in the BS version. The linearization point of the full check is the last read of an index in `find_empty_slot` (line 5), if there is still an item left in the observed head k-segment (line 14), and the `head` pointer did not change in the meantime (line 15). The linearization point of the `dequeue` method that returns an item is the successfully executed `CAS` with an `EMPTY` item (line 50). The linearization point of the emptiness check is the first read of an index in `find_item` (line 43) in the head k-segment, if the `head` pointer then points to `tail`, and `tail` did not change in the meantime (line 53).

The correctness argument is based on the following facts.

1. An item is enqueued in the queue exactly once. This is a consequence of our unique-items assumption and the control flow of the `enqueue` method that can only modify one slot a time. If the `committed` method identifies an invalid insertion, it removes the item using `CAS` and retries. Hence, at every point in time the item is at most in one k-segment.

2. If an enqueue operation returns full, then during its execution there must be a state at which at least n items have been inserted successfully into the queue (BS only). Since returning full is without any side effect, it suffices to prove the existence of a state which corresponds to a logically full queue. Inserting n items in the queue results in $\lceil \frac{n}{k} \rceil + 1$ used k-segments of which all but the `head` segment contain k items, i.e., they are full. The current `tail` points to the last k-segment before `head` and has been found full. If then the `head` segment also contains at least one item, it is not possible to advance `tail`. Hence, the queue is full.

3. An item is dequeued at most once. If an item `i` is in the queue it can only be removed once, because of 1. and a statement which replaces `i` with `EMPTY`. If `i` is in some slot, but not logically in the queue, then `enqueue` removes it and retries the insertion before committing again. We show that while `i` is in some slot, but not logically in the queue, no `dequeue` can return `i`. Clearly, the call to `committed` has to return `false`, implying that no other thread modifies the slot of `i`. Otherwise, either the first `if` statement (line 22) or the following failed `CAS` attempts (lines 30 and 36) of replacing `i` with `EMPTY` lead to returning `true`. When the control flow reaches the only point for returning `false` in `committed` it is guaranteed that there is no slot containing `i` anymore, making it impossible to dequeue the item.

4. If a dequeue operation returns empty, then during its execution, there must be a state at which there a no items in the queue. Similar to 2. returning empty is without any side effects, so it suffices to show that there exists a state which corresponds to a logical empty state. The queue is empty at the time it observes the first slot in `find_item` as empty, if then all slots are observed as empty and if then the observed `head _old` points to `tail_old` and the actual `tail` did not change in the meantime. This is enough because it guarantees that no slot gets modified by enqueue or dequeue operations until the slots' state is checked.

5. An item j cannot be dequeued before an item i, if they are both in the queue and i, j are in segments s, s', respectively, with s' reachable from s and $s \neq s'$. The segment s' can become the head segment only after the segment s has been removed. Moreover, unreachable segments can not contain items that are logically in the queue as validated in `committed`. These observations imply that with respect to linearization points `dequeue(i)` precedes `dequeue(j)`, which only happens if s' is the head segment.

6. An item i can overtake at most $k-1$ other elements. Assume the item i resides in some segment s' reachable from the current head segment s. Because of 5. i cannot overtake any items until s' becomes the head k-segment. In the case that s' becomes the head k-segment, the item i may be dequeued first, effectively overtaking at most $k-1$ other items.

7. An item i is overtaken by at most $k-1$ other items. Again, because of 5. an item i can only be overtaken by elements residing in the same k-segment. As a result i can only be overtaken by at most all $k-1$ other items in the segment.

□

The following fairness property follows from Facts 1. and 5.:

Corollary 1. *Dequeuing an element e from a k-FIFO queue may take at most $n+k$ dequeue operations, where n is the number of elements in the queue when e was inserted into the queue.*

Proposition 2. *The k-FIFO queue algorithms are lock-free.*

Similar to others (cf. [1]), one can show that both algorithms are lock-free by demonstrating that whenever some operation retries another operation is making progress. The somewhat lengthy argument can be found elsewhere [4] and is omitted due to space limitations. Note that the US k-FIFO queue allocates memory dynamically through a lock-free allocator.

4 Related Work

We relate our BS and US k-FIFO queue algorithms to existing concurrent data structure algorithms, which we also implemented for and evaluated in a number of experiments in Section 5.

The following queues implement regular unbounded-size FIFO queues: a standard lock-based FIFO queue (LB), the lock-free Michael-Scott FIFO queue (MS) [1], and the flat-combining FIFO queue (FC) [6] (FC). LB locks a mutex for each data structure operation. With MS each thread uses at least two CAS operations to insert an element into the queue and at least one CAS operation to remove an element from the queue. FC is based on the idea that a single thread performs the queue operations of multiple threads by locking the whole queue, collecting pending queue operations, and applying them to the queue. The lock-free bounded-size FIFO queue (BS) [7] is based on an array of fixed size where elements get inserted and removed circularly and enqueue operations may fail when the queue is full, i.e. every array slot holds an element.

The Random Dequeue Queue (RD) [2] is based on MS where the dequeue operation was modified in a way that, given a configurable integer r, a random number in $[0, r-1]$ determines which element is returned starting from the oldest element.

The Segment Queue (SQ) [2] is closely related to the US k-FIFO queue. SQ is based on a list of segments of size $s > 0$ where s is a configurable integer. With SQ an enqueue operation inserts an element at an arbitrary empty position of the youngest segment using a CAS operation. This is similar to our proposed US k-FIFO queue. Analogously, with SQ a dequeue operation logically removes an element at an arbitrary position of the oldest segment using a CAS on a deleted flag. SQ considers all slots for deletion, i.e. it eagerly tries to remove an item (even it already has been removed). T This is different to the US k-FIFO queue where elements are removed lazily, i.e., a CAS operation is only performed on a slot still holding an item.

Experiments reported elsewhere [2] show that in certain configurations (parameters r and s, respectively), RD and SQ scale better than MS. However, the improved scalability comes at the expense of peak performance, as the data also shows that the best overall throughput is reached by MS with only few threads. Neither SQ nor RD reach that peak performance with any number of threads.

In contrast to the US k-FIFO queue, SQ does not provide a linearizable emptiness check. Consider the following example of using SQ: Starting with an empty queue, thread A enqueues the first element into the queue at position $s-1$ of a new segment of size s. After that thread A attempts to perform a dequeue operation by iterating over the slots of the new segment just finding empty slots until slot $s-1$. Right before checking the slot at position $s-1$ thread A gets descheduled. Thread B now performs $s-1$ enqueue operations and fills up the whole segment with elements. After that Thread B performs a dequeue operation and removes the element from the slot at position $s-1$. Now, thread A wakes up, checks the slot at position $s-1$, encounters that also this slot is empty, and returns *null*. However, at any point in time there was at least one element in the queue. In other words, SQ reported empty although the queue was never empty.

The lock-free linearizable pool (BAG) [8] is based on thread-local lists of blocks of elements. Each block is capable of storing up to a constant number of elements. A thread performing an enqueue operation always inserts elements into the first block of

its thread-local list. Once the block is full, a new block is inserted at the head of the list. A thread performing a dequeue operation always tries first to find an element in the blocks of its thread-local list. If the thread-local list is empty, work stealing from other threads' lists is used to find an element. The work-stealing algorithm implements a linearizable emptiness check by repeatedly scanning all threads' lists for elements and marking already scanned blocks which are unmarked when elements are inserted. The implementation works only for a fixed number of threads.

The lock-free elimination-diffraction pool (ED) [9] uses FIFO queues to store elements. Access to these queues is balanced using elimination arrays and a diffraction tree. While the diffraction tree acts as a distributed counter balancing access to the queues, elimination arrays in each counting node increase disjoint-access parallelism. Operations hitting the same index in an elimination array can either directly exchange their data (enqueue meets dequeue), or avoid hitting the counter in the node that contains the array (enqueue meets enqueue or dequeue meets dequeue). If based on non-blocking FIFO queues, the presented algorithm lacks a linearizable emptiness check. If based on blocking queues, there is no empty state at all. Parameters, i.e., elimination waiting time, retries, array size, tree depth, number of queues, queue polling time, need to be configured to adjust ED to different workloads.

The synchronous rendezvousing pool (RP) [10] implements a single elimination array using a ring buffer. Both enqueue and dequeue operations are synchronous. A dequeue operation marks a slot identified by its thread id and waits for an enqueue operation to insert an element. An enqueue operation traverses the ring buffer to find a waiting dequeue operation. As soon as it finds a dequeue operation they exchange values and return. There exist adaptive and non-adaptive versions of the pool where the ring buffer size is adapted to the workload.

5 Experiments

We evaluate the performance and scalability of the BS and US k-FIFO queue algorithms. All experiments ran on an Intel-based server machine with four 10-core 2.0GHz Intel Xeon processors (40 cores, 2 hyperthreads per core), 24MB shared L3-cache, and 128GB of memory running Linux 2.6.39. We implemented a framework to benchmark and analyze different queue and pool implementations under configurable scenarios and workloads. For microbenchmarking the framework emulates a multi-threaded producer-consumer workload where each thread is either a producer or a consumer. The workload can be configured for a different number of threads (n), number of enqueue or dequeue operations each thread performs (o), the computational load performed between each operation (c), the number of pre-filled items (i), and the queue implementation to use. The computational load c between two consecutive operations is created by iteratively calculating π. A computation with $c = 1000$ takes a total of $2300ns$ on average. We fix the operations per thread to $o = 1000000$ and the number of producers and consumers to $\frac{n}{2}$ for all benchmarks. We evaluate the performance and scalability of the queues under low ($c = 10000$), medium ($c = 7000$), high ($c = 4000$) and very high ($c = 1000$) contention. In order to analyze the performance impact and applicability of our k-FIFO queues on real applications we use three different macrobenchmarks: Mandelbrot set calculation, graph traversal computing the spanning tree, and graph traversal computing the transitive closure of a graph.

To avoid paging and caching issues we use our own lock-free allocator, which touches memory pages upon program initialization and can be used to return cache- and page-aligned memory. For the purpose of benchmarking we do not free allocated memory. Freeing memory in lock-free algorithms is an orthogonal problem and can be solved by using hazard pointers [11].

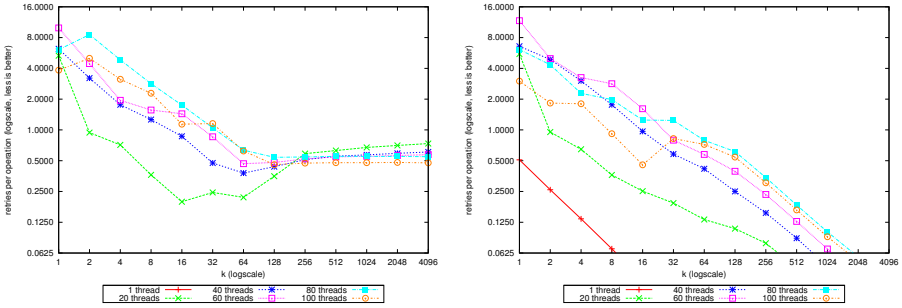
5.1 Understanding k

In order to provide a better understanding of the effect of k on performance and scalability we first evaluate the performance of the BS version with increasing k. We omit the measurements for the US version as the results are similar. Performance is measured under very high ($c = 1000$) contention without ($i = 0$) and with ($i = 5000$) pre-filling the queue. Other contention scenarios lead to similar results. We relate the performance of our producer-consumer benchmark, measured in operations per millisecond, to the number of retries per operation and the number of so-called failed reads. Retries are an indicator of contention among `CAS` operations. They occur whenever an enqueue (line 2) or a dequeue (line 39) operation has to take another iteration of the while loop. Failed reads are attempts to find empty slots or items in `find_empty_slot` (line 5) and `find_item` (line 42), respectively. Both retries and failed reads produce overhead and should thus be minimized in order to improve overall performance.

Figure 1 depicts this performance analysis with non-pre-filled results on the left and pre-filled results on the right side. Intuitively, one would expect that a larger k results in fewer retries because of reduced contention among the inserting (line 9) and removing (line 48) `CAS` operations. Figure 1(b) shows that this is true for a setting where the queue is pre-filled with items, i.e., a queue with an initially dense population in the k-segment that is used for dequeueing. However, for a workload where the queue is initially empty there exists a turning point from which the number of retries starts to grow with increasing k. Figure 1(a) depicts this behavior which appears when the k-segment used for dequeueing is only sparsely populated most of the time. In this case the dequeueing operations are likely to contend on the same, rare items in the head k-segment. Figure 1(c) and 1(d) illustrate the number of failed reads. As long as the number of retries is decreasing, failed reads are slowly increasing with larger k since the k-segments to search for items or empty slots get bigger. As soon as the number of retries reaches the turning point in the pre-filled case failed reads are increasing exponentially. Figure 1(e) and 1(f) then visualize the impact of an increasing k on the performance and show that there exists an optimal k with respect to performance. The optimal k is also robust in the sense that there exists only a single range of close-to-optimal k. Furthermore, the population density of a k-segment that is used for dequeueing has an impact on the range of k where good performance can be observed. If k gets too large, i.e., the population in the dequeueing segment is sparse, the performance significantly decreases. The depicted behavior of k suggests a controller that optimizes k to dynamic workloads.

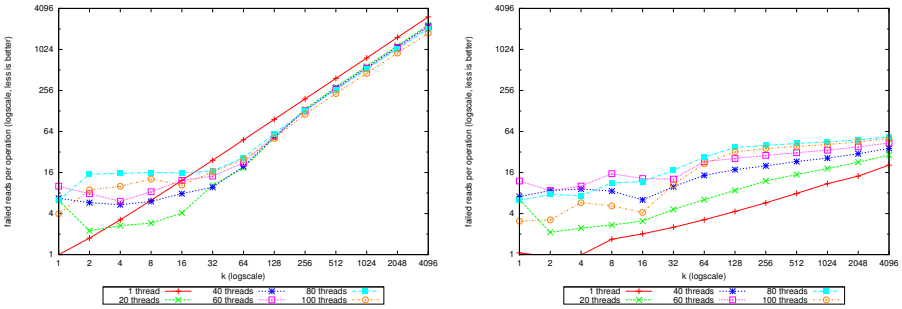
5.2 Performance and Scalability

We study the performance of LB, BS, MS, FC, RD, SQ, BS k-FIFO and US k-FIFO queues, and ED, BAG, and RP pools. For the RD, SQ, BS k-FIFO, and US k-FIFO queues we configure $r = s = k = 64$ (see Section 4), which we determined to be a fair



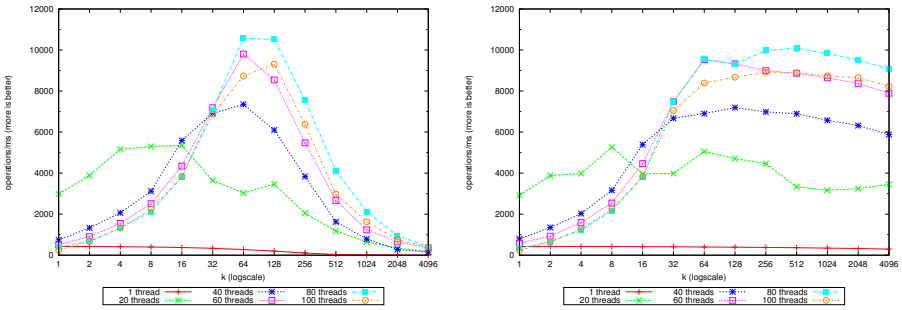
(a) BS number of retries per operation for very high contention ($c = 1000$, $i = 0$)

(b) BS number of retries per operation for very high contention ($c = 1000$, $i = 5000$)



(c) BS number of failed reads per operation for very high contention ($c = 1000$, $i = 0$)

(d) BS number of failed reads per operation for very high contention ($c = 1000$, $i = 5000$)



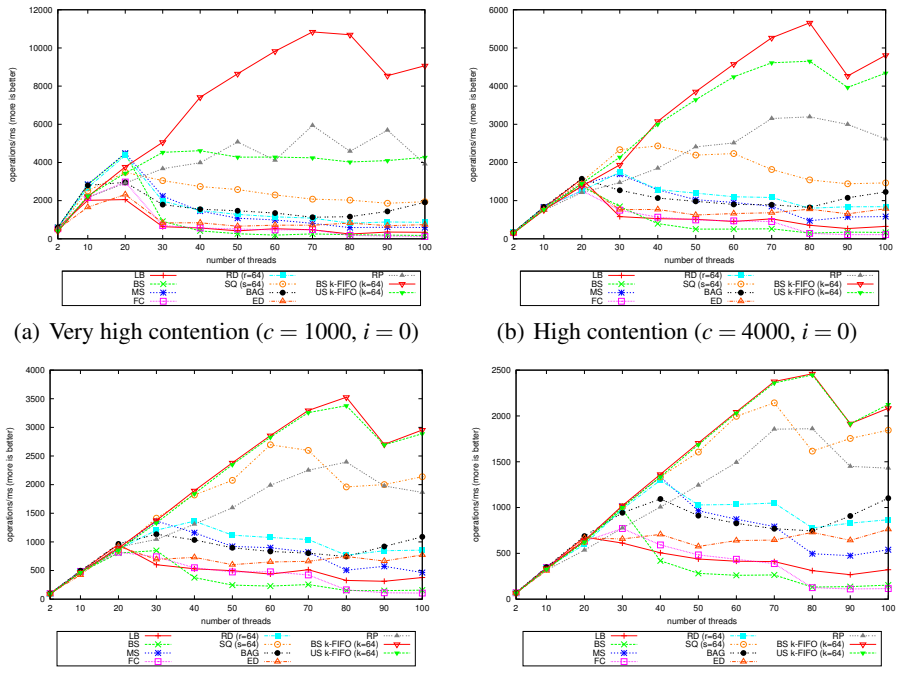
(e) BS performance of very high contention ($c = 1000$, $i = 0$)

(f) BS performance of very high contention ($c = 1000$, $i = 5000$)

Fig. 1. Very high and low contention producer-consumer microbenchmarks with an increasing number of k for different amounts of threads

and representative configuration for a broad range of thread combinations and workloads. We use the non-adaptive version of the RP algorithm since the number of threads is constant in each run.

Producer-Consumer. Figure 2(a) illustrates the results for the very high contention workload where MS and RD perform best for up to 20 threads. With more than 20 threads scalability is negative for all data structures except RP, BS k-FIFO, and US k-FIFO. The BS k-FIFO queue algorithm is the only algorithm that scales near-linearly.



(a) Very high contention ($c = 1000$, $i = 0$)

(b) High contention ($c = 4000$, $i = 0$)

(c) Medium contention ($c = 7000$, $i = 0$)

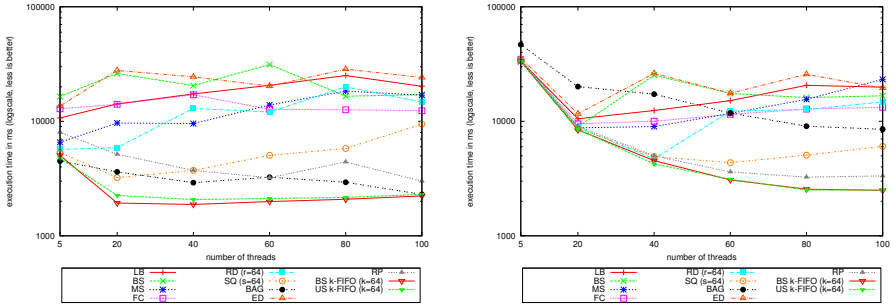
(d) Low contention ($c = 10000$, $i = 0$)

Fig. 2. Performance and scalablity of producer/consumer microbenchmarks with an increasing number of threads

Similarly, the results with our high contention scenario, depicted in Figure 2(b), show that the scalability turnaround is at 30 threads and that both k-FIFO versions outperform and outscale all other algorithms. As the contention gets less in Figures 2(c) and 2(d), the turnaround gets shifted to a larger number of threads. The difference in performance and scalability of all algorithms is less significant with more computational load. Note that SQ returns up to 2000 times falsely `null` due to the non-linearizable emptiness check.

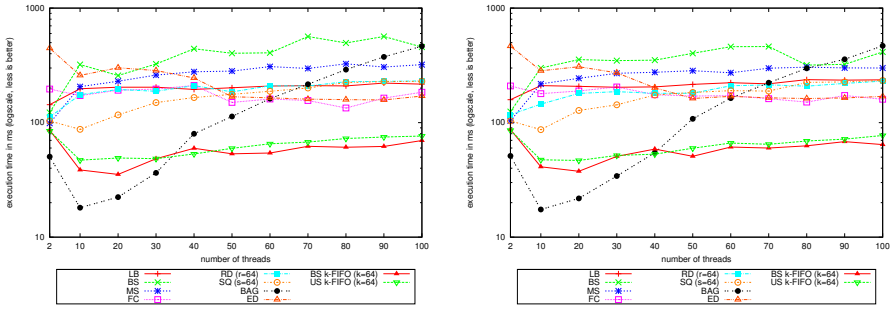
Mandelbrot. We computed and rendered two images of the Mandelbrot set [12] using producer and consumer threads and a shared data structure to distribute the computation across multiple cores. The producer threads divide the image into smaller blocks (4x4 pixels in our experiments), write block coordinates in descriptor blocks, and enqueue the descriptor blocks in the shared data structure. The consumer threads dequeue the descriptor blocks from the shared data structure, perform the Mandelbrot calculation on the blocks, and store the results in the corresponding blocks of the final Mandelbrot image. Hence, the workload between the consumer threads is balanced. We use a producer-consumer ratio of 1 : 4 in our experiments, i.e. for each producer thread we add four consumer threads.

The Mandelbrot macrobenchmark results are presented in Figure 3. Each run was repeated 10 times. We present the average execution time of the 10 runs as our metric of performance, less execution time is better. Figure 3(a) shows the performance of the low computational load Mandelbrot benchmark. Low computational load means that



(a) Low computational load (high contention) (b) High computational load (low contention)

Fig. 3. Parallel Mandelbrot set calculation with an increasing number of threads (producer-consumer ratio 1:4)



(a) Transitive closure (b) Spanning tree

Fig. 4. Performance and scalability of graph traversals on a random graph with 100000 vertices, 1000000 edges, and 160 starting vertices with an increasing number of threads

Mandelbrot computations are fast for most of the blocks, i.e. the number of iterations in the Mandelbrot calculations is small or zero, and thus the contention on the shared data structure is high. Both k-FIFO algorithms show the best performance, followed by the pools BAG and RP. The result of the high computational load Mandelbrot benchmark is depicted in Figure 3(b). High computational load means that the Mandelbrot computations are computationally intensive for most of the blocks, resulting in less contention on the shared data structure. The figure shows that the BS and US k-FIFO queues provide the best performance and scalability.

Graph Algorithms. We ran two macrobenchmarks with parallel versions of transitive closure and spanning tree graph algorithms [13] using random graphs consisting of 100000 vertices where 1000000 unique edges got randomly added to the vertices. Non-connected subgraphs are then connected using single edges. The shared data structure is prefilled with 160 randomly determined vertices. From then on each thread iterates over the neighbors of a given vertex and tries to process them (transitive closure or spanning tree operation). Depending on the algorithm, the check whether a neighboring vertex has already been processed is raceful (transitive closure), or exact (spanning tree). Already processed vertices are then ignored. After processing a vertex, it gets added to the shared data structure. The thread then gets a new vertex from the shared data structure. The graph algorithm terminates as soon as the global queue is empty.

The spanning tree and transitive closure macrobenchmark results are presented in Figure 4. Each run was repeated 10 times. We present the average execution time of the 10 runs as our metric of performance, less execution time is better. The RP pool is not used in this benchmark since it cannot handle a workload where producers are also consumers. Both benchmarks show how the data structures behave under extremely high contention. BAG results in best performance at 10 threads, and then scales negatively. This peak performance is reached because producers are also consumers and thus each thread can access its thread-local list. Note that this result for BAG can only be reached in tailored workloads as the other benchmarks show. In general, all data structures have problems scaling under this high contention. The best performance (except BAG) is reached by the BS and US k-FIFO queues.

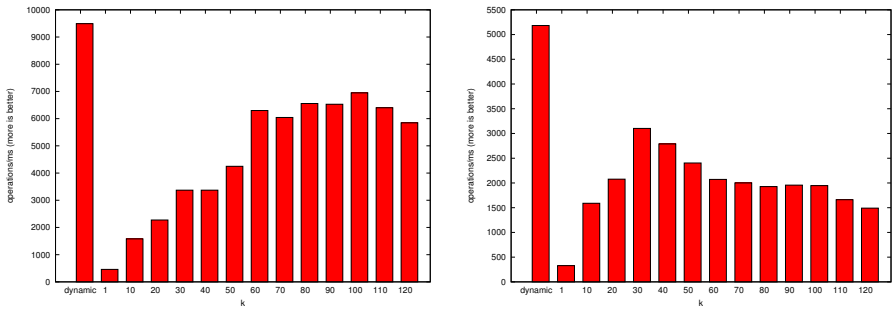
5.3 Dynamic k

We implemented a prototypical PID controller which aims at identifying optimal k automatically at runtime for best performance. Each thread i stores performed enqueue operations o_i and performed retries in enqueue operations r_i in thread-local counters. The controller runs in an extra thread, reads the thread-local counters of all n threads periodically (100ms), and resets them to 0 after reading. The goal of the controller is to minimize the ratio $\sum_{i=1}^{n} r_i / \sum_{i=1}^{n} o_i$. The controller operates in the approximately linear part of this ratio. With the US k-FIFO algorithm the controller determines the k-segment size that enqueue operations use to create new segments which store their size for dequeue operations to look up. For the BS k-FIFO algorithm the maximum k needs to be bounded to provide a linearizable emptiness check.

We use a variable-load producer-consumer microbenchmark to evaluate the performance of the controller. Each thread performs $o = 4000000$ operations and starts with $c = 1000$. The workload changes for each thread whenever $o/4$ operations are performed by changing c to 500, 2000, and 1500. We compare the BS and US k-FIFO algorithms with dynamically controlled k to the unmodified baseline versions with statically configured k ranging from 1 to 120. Figure 5(b) shows the performance of the dynamic US k-FIFO queue. Here the controller improves the performance by 60% over the best statically configured configurations. The dynamic BS k-FIFO queue performs about 30% faster than the best statically configured configuration, see Figure 5(a). As explained in Section 5.1 and illustrated in Figure 1, best performance for different levels of contention can be achieved through different configurations of k. The level of contention is proportional to the rate of retries to which k is therefore adjusted dynamically.

6 Conclusions

We have introduced fast and scalable algorithms that implement bounded- and unbounded-size, lock-free, linearizable k-FIFO queues with emptiness (and full) check. We showed experimentally for both algorithms that there exist optimal and robust k that result in best performance and scalability. Moreover, we demonstrated in experiments that our algorithms outperform and outscale many state-of-the-art concurrent queue and pool algorithms on different concurrent producer-consumer workloads. Finding the right k for different workloads is key for best performance and scalability. We suggest to either set k statically to around the number of available parallel processing units or use a controller which automatically adjusts k at runtime as shown in our experiments.



(a) BS k-FIFO queue (b) US k-FIFO queue

Fig. 5. Variable-load producer-consumer microbenchmarks with an increasing number of static k versus a dynamically controlled k

Acknowledgments. This work has been supported by the National Research Network RiSE on Rigorous Systems Engineering (Austrian Science Fund S11404-N23). We thank the anonymous referees for their constructive comments and suggestions.

References

1. Michael, M., Scott, M.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: Proc. Symposium on Principles of Distributed Computing (PODC), ACM (1996) 267–275
2. Afek, Y., Korland, G., Yanovsky, E.: Quasi-linearizability: Relaxed consistency for improved concurrency. In: Proc. Conference on Principles of Distributed Systems (OPODIS), Springer (2010) 395–410
3. Henzinger, T., Kirsch, C., Payer, H., Sezgin, A., Sokolova, A.: Quantitative relaxation of concurrent data structures. In: Proc. Symposium on Principles of Programming Languages (POPL), ACM (2013)
4. Kirsch, C., Lippautz, M., Payer, H.: Fast and scalable k-fifo queues. Technical Report 2012-04, Department of Computer Sciences, University of Salzburg (June 2012)
5. Herlihy, M., Wing, J.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. (TOPLAS) **12**(3) (1990) 463–492
6. Hendler, D., Incze, I., Shavit, N., Tzafrir, M.: Flat combining and the synchronization-parallelism tradeoff. In: Proc. Symposium on Parallelism in Algorithms and Architectures (SPAA), ACM (2010) 355–364
7. Colvin, R., Groves, L.: Formal verification of an array-based nonblocking queue. In: Proc. Conference on Engineering of Complex Computer Systems (ICECCS), IEEE (2005) 507–516
8. Sundell, H., Gidenstam, A., Papatriantafilou, M., Tsigas, P.: A lock-free algorithm for concurrent bags. In: Proc. Symposium on Parallelism in Algorithms and Architectures (SPAA), ACM (2011) 335–344
9. Afek, Y., Korland, G., Natanzon, M., Shavit, N.: Scalable producer-consumer pools based on elimination-diffraction trees. In: Proc. European Conference on Parallel Processing (Euro-Par), Springer (2010) 151–162
10. Afek, Y., Hakimi, M., Morrison, A.: Fast and scalable rendezvousing. In: Proc. International Conference on Distributed Computing (DISC), Springer (2011) 16–31
11. Michael, M.: Hazard pointers: Safe memory reclamation for lock-free objects. IEEE Trans. Parallel Distrib. Syst. **15**(6) (June 2004) 491–504
12. Mandelbrot, B.: Fractal aspects of the iteration of $z \to \lambda z(1-z)$ for complex λ and z. Annals of the New York Academy of Sciences **357** (December 1980) 249–259
13. Bader, D., Cong, G.: A fast, parallel spanning tree algorithm for symmetric multiprocessors (SMPs). Journal of Parallel and Distributed Computing **65** (2005) 994–1006