UNIVERSITÄT SALZBURG

Fast and Scalable k-FIFO Queues

Christoph M. Kirsch Michael Lippautz Hannes Payer

Technical Report 2012-04 June 2012

Department of Computer Sciences

Jakob-Haringer-Straße 2
5020 Salzburg
Austria
www.cosy.sbg.ac.at

Technical Report Series

Fast and Scalable k-FIFO Queues*

Christoph M. Kirsch Michael Lippautz Hannes Payer

Department of Computer Sciences
University of Salzburg, Austria
firstname.lastname@cs.uni-salzburg.at

Abstract. We introduce fast and scalable algorithms that implement bounded- and unbounded-size, lock-free, linearizable k-FIFO queues with empty (and full) check. Logically, a k-FIFO queue can be understood as queue where each element may be dequeued out-of-order up to $k-1$ or as pool where each element is dequeued within a k-bounded number of dequeue operations. We show experimentally that there exist optimal and robust k that result in best performance and scalability. We then demonstrate that our algorithms outperform and outscale many state-of-the-art concurrent queue and pool algorithms on different workloads. We finally demonstrate a prototypical controller which aims at identifying optimal k automatically at runtime for best performance.

1 Introduction

We are interested in the design and implementation of fast concurrent FIFO queues and pools 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, linearizable k-FIFO queues with empty (and full) check. Logically, enqueueing an element into a k-FIFO queue appends the element at the end of the queue. A k-FIFO queue can then be understood as queue where each element may be dequeued out-of-order up to $k-1$ or as pool where each element is dequeued within $m+k$ number of dequeue operations and m is the number of elements in the pool when the element was enqueued. Thus a 1-FIFO queue is a regular FIFO queue and a k-FIFO queue provides bounded fairness [6] for finite k. The details are discussed in Section 2.

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) [12] but without a sentinel node. The details are discussed in Section 3. Logically, the idea of k-segments is similar to the idea of segments in the Segment Queue (SQ) [4]. On data structure (but not code) level SQ is related to the US algorithm, which improves

* This work has been supported by the National Research Network RiSE on Rigorous Systems Engineering (Austrian Science Fund S11404-N23) and the National Science Foundation (CNS1136141).

upon SQ in performance and scalability through less overhead and reduced contention and in features through a linearizable empty check. 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 FIFO queue and pool 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 Sequential Specification

$$enqueue_k(e)(q,w) = (q \cdot e, w)$$

$$dequeue_k(e)(q,w) = \begin{cases} (\varepsilon, k) & \text{if } e = null, q = \varepsilon \quad \text{(L1)} \\ (q', k) & \text{if } q = e \cdot q' \quad \text{(L2)} \\ (e_1 \dots\ e_{i-1} e_{i+1} \dots\ e_n, w-1) & \text{if } e = e_i, q = e_1 \dots\ e_n, \quad \text{(L3)} \\ & 1 < i \leq min(w,n) \\ error & \text{otherwise} \quad \text{(L4)} \end{cases}$$

Fig. 1. k-FIFO sequential specification

We define a sequential specification of the k-FIFO queue algorithms to facilitate a high-level discussion here and a linearizability argument in Section 3. The formal definition of the sequential specification is shown in Figure 1. Similar to a regular FIFO queue, a k-FIFO queue provides an enqueue and a dequeue operation.

Let the tuple (q,w) denote the state of a k-FIFO queue where q is the sequence of n queue elements with $n \geq 0$ and $w \geq 1$ is an integer called window size, which determines the range of elements starting at the oldest queue element out of which one element is selected to be removed and returned by a dequeue operation. The initial, empty state of the k-FIFO queue is (ε, k).

The enqueue operation is a function from queue states and queue elements to queue states. It logically inserts an element at the end of the queue. The dequeue operation is a function from queue states and queue elements or the *null* return value, which indicates an empty queue, to queue states. It logically removes an element among the elements $e_1 \dots\ e_i$ from the queue where $i = \min(w,n)$, if the queue is not empty, and otherwise returns *null*. The window size w is maintained with every enqueue operation, decreases with every dequeue operation that returns an element e_i with $i > 1$, and is reset to k when element e_1 is dequeued. Intuitively, with a k-FIFO queue up to $k-1$

younger elements may be dequeued before a given element where each dequeued element is not younger than the w-oldest element, and *null* is only returned when the queue is empty. In particular, dequeueing an element e_i from a k-FIFO queue $e_1 \dots e_n$ with $k > 0$ may take anywhere between l and u dequeue operations that may return the elements $e_1 \dots e_{i-1} e_{i+1} \dots e_{u+1}$ where $l = \lfloor \frac{i}{k} \rfloor \cdot k$ and $u = min(l+k, n) - 1$. Thus a 1-FIFO queue is a regular FIFO queue and a k-FIFO queue provides bounded fairness [6] for finite k. For brevity we use the terms enqueue and dequeue with queues as well as pools throughout the paper.

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 standard ABA counters. We refer to values enhanced with ABA counters as atomic values. 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 linearizability with respect to the k-FIFO sequential specification.

When implementing a linearizable FIFO queue the head and tail pointers may become scalability 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`, which may hold pointers to the actual inserted items or point to `null` indicating an empty slot. 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 (line 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 at most $\texttt{TESTS} \geq 1$ array locations for an empty slot starting with the selected index wrapping around at index $\texttt{tail}+k-1$. In our experiments we observed that $\texttt{TESTS} = k$ results in the best performance. 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 compare-and-swap (`CAS`) operation

Listing 1.1. Lock-free bounded-size (BS) and unbounded-size (US) k-FIFO queue algorithms. Gray highlighted code is only used in the BS version

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

(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. If any of these steps fails a retry is performed. If no empty slot is found in the current tail k-segment the `enqueue` method tries to increment `tail` by k using `CAS` (line 17) and then retries. Hence in the worst-case only `TESTS` slots may be used in a k-segment if `TESTS` $< k$.

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 16). If this k-segment is not empty and `head_old` did not change in the meantime the queue is full and `false` is returned (line 15).

The `committed` method (line 19) 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 thread (line 20, 28, 37) 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 25). 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 28) the method tries to undo the insertion using `CAS` (line 28). 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 ABA counter in the `head` atomic value using `CAS` (line 32). 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 34).

The `dequeue` method is depicted in Listing 1.1 (line 38). 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. 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 43) which would trigger a retry. If an item was found (line 44) the method first checks whether `head` equals `tail` (line 45). 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 empty check. Afterwards the method tries to remove the item using `CAS` (line 48) and returns it if the removal was successful (line 49). Otherwise 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 52). Otherwise the method tries to increment `head` by k (line 53) and performs a retry.

Note that to hold n items in the BS k-FIFO queue `size` has to be at least $\lceil \frac{n}{\texttt{TESTS}} \rceil \times k + k$. The additional k-segment is necessary to implement linearizable empty and full checks.

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`.

The US k-FIFO queue algorithm implements a FIFO queue of k-segments, called k-segment queue. 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.

Any FIFO queue algorithm may be used to implement the k-segment queue. We developed a lock-free FIFO queue which performs better than the lock-free Michael-Scott FIFO queue (MS) [12] when used as k-segment queue in the US k-FIFO queue algorithm. We implemented the k-segment queue such that there is always at least one k-segment, even if it is empty, in the queue, which enables fast and direct access to the head and tail k-segments. The pseudo code of our k-segment queue is shown in the Appendix B. The `advance_tail` method adds a new k-segment to the k-segment queue if `tail` did not change in the meantime, i.e., `tail` still points to `tail_old`. The `advance_head` method removes the head k-segment if it is not the only k-segment in the queue and if it did not change in the meantime, i.e., `head` still points to `head_old`. Before removing the head k-segment it marks that k-segment as deleted. The `committed` method in the US version is similar to the BS version. It decides based on whether the current head k-segment is marked as deleted or not if it has to undo the insertion, instead of performing a range check.

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 particular, the empty US 1-FIFO queue contains an empty 1-segment in which the first enqueued element is stored, in contrast to MS. 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). The pseudo code is in Appendix C.

3.2 Correctness

We informally show that both the BS and US k-FIFO queue algorithms are linearizable [8] with respect to the k-FIFO sequential specification in Section 2.

The linearization point [8] of the `enqueue` method that inserts an item is the successfully executed `CAS` in line 9 if the `committed` method subsequently returns `true`. In the BS version, the queue may be full. The linearization point of the full check is in line 14 if the tail k-segment was found to be full, an item was found in the head k-segment, and the `head` pointer did not change in the meantime.

The linearization point of the `dequeue` method that returns an item is the successfully executed `CAS` in line 48. The linearization point of the empty check is in line 51 if no item was found in the head k-segment and `head` still points to `tail`.

The BS and US k-FIFO queue algorithms are linearizable with respect to the k-FIFO sequential specification if for each obtained concurrent history [8] there is a sequential history in which concurrent methods are ordered according to their linearization points and these linearization points are within the k-FIFO sequential specification. This is equivalent to requiring that for the linearization point of a dequeue method, the value of the item must satisfy the k-FIFO sequential specification. An enqueue method always inserts an item at a random position in the tail k-segment. An enqueue method encountering the full state in the BS k-FIFO queue means that at the linearization point tail is pointing to the k-segment before head, the tail k-segment is full, and the head k-segment

contains at least one item. This implies that at the linearization point, the queue is logically full. A dequeue method returning `null` means that at its linearization point head and tail were pointing to the same k-segment and the k-segment was found to be empty. This implies that at the linearization point, the queue is logically empty. Let l be the number of items in the head k-segment and let w be the window size which according to the k-FIFO sequential specification is set to k before the first item gets dequeued from that k-segment. For the dequeue method returning a non-`null` item the selected item cannot be younger than the l-oldest item in the k-FIFO queue for $l \leq w \leq k$. This is because one of the l items is the youngest item in the k-segment and younger items than the l-oldest item can only get dequeued when the `head` pointer advances, i.e., after dequeueing the l items for which $w \geq l$. Moreover, an item can at most be overtaken by $l-1 \leq w-1 \leq k-1$ younger items. This is because no new items can get added to the head k-segment when dequeue methods remove items from it. If `head` and `tail` point to the same k-segment the dequeue method first advances `tail` before an item is removed. Hence the head k-segment has to become empty before younger items are dequeued, i.e., after dequeueing the $l+1$ items with $w \geq l$.

The BS and US k-FIFO queue algorithms are lock-free [7] since the enqueue and dequeue methods may not make progress only when other concurrent threads successfully enqueue or dequeue an element. An informal discussion is in the Appendix A.

4 Related Work

The topic of this paper is part of a recent trend towards scalable but semantically weaker concurrent data structures [13, 10, 11] acknowledging the intrinsic difficulties of implementing deterministic semantics in the presence of concurrency [5]. We relate our BS and US k-FIFO queue algorithms to existing concurrent data structure algorithms, which we also implemented 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) [12], and the flat-combining FIFO queue (FC) [9] (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) [1] 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) [4] implements the k-FIFO sequential specification where $k = r$ and r defines the range $[0, r-1]$ of a random number. RD is based on MS where the dequeue operation was modified in a way that the random number determines which element is returned starting from the oldest element. RD does not scale better than MS in experiments reported on elsewhere [4].

The Segment Queue (SQ) [4] is similar to the US k-FIFO queue but without a linearizable empty check. Hence SQ may return empty even if the queue is not empty. SQ is implemented by a lock-free FIFO queue of segments but here the queue of segments

may become empty. A segment can hold s queue elements. An enqueue operation inserts an element at an arbitrary position of the youngest segment. A dequeue operation removes an arbitrary element from the oldest segment. When a segment becomes full a new segment is added to the queue. When a segment becomes empty it is removed from the queue. A thread performing a dequeue operation starts looking for an element in the oldest segment. If the segment is empty it is removed and the thread checks the next oldest segment and so on until it either finds an element and returns that, or else may return *null*. SQ does not perform better than MS in experiments reported on elsewhere [4]. A reason for that may be that SQ tries to atomically swap each value in a segment using `CAS` until a valid item is removed or inserted, respectively. In comparison the US k-FIFO queue algorithm first tries to find a corresponding index in the k-segment and only performs a CAS on the location of the index.

The lock-free linearizable pool (BAG) [14] 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 empty 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) [3] 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 empty 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) [2] 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

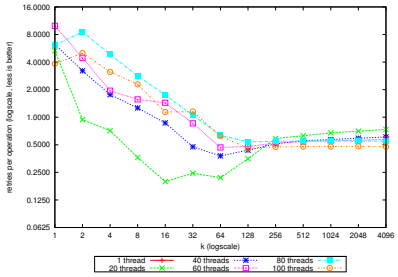
We evaluate the performance and scalability of the BS and US k-FIFO queue algorithms. All experiments ran on an Intel-based server machine with four 10-core 2.0GHz

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. The framework emulates a multi-threaded producer-consumer workload where each thread is either a producer or a consumer. The framework 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. The framework uses static preallocation for memory used in the data structures with touching each page before running the benchmark to avoid paging issues.

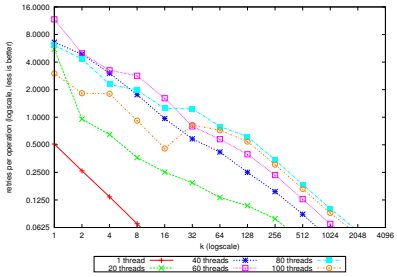
5.1 Understanding k

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.

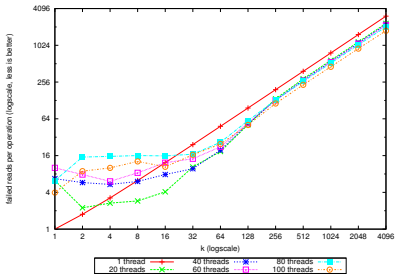
Figure 2 depicts this performance analysis with non-pre-filled results on the left and pre-filled results on the right side. Intuitively, one would expect that a larger k results in fewer retries because of reduced contention among the inserting (line 9) and removing (line 48) `CAS` operations. Figure 2(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 2(a) depicts this behaviour 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 2(c) and 2(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 2(e) and 2(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



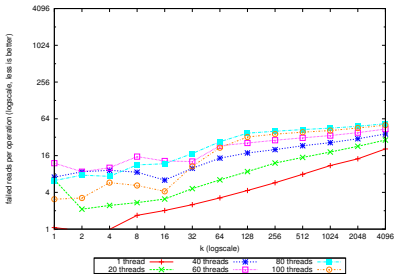
(a) BS number of retries per operation of very high contention producer-consumer benchmark ($c = 1000$, $i = 0$)



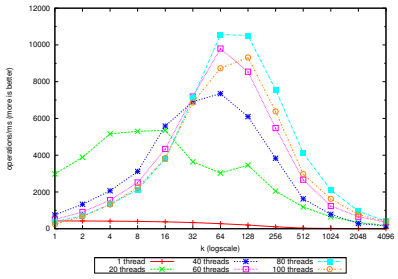
(b) BS number of retries per operation of very high contention producer-consumer benchmark ($c = 1000$, $i = 5000$)



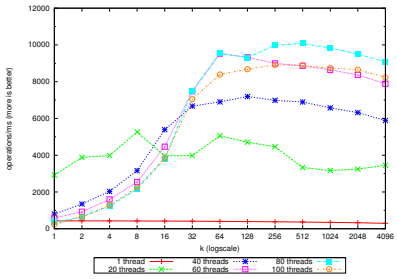
(c) BS number of failed reads per operation of very high contention producer-consumer benchmark ($c = 1000$, $i = 0$)



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



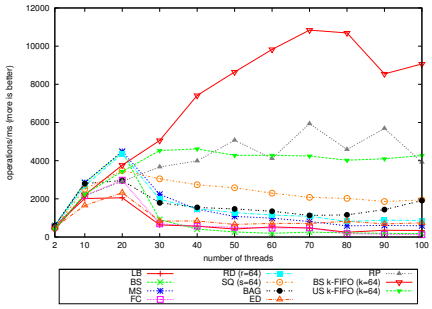
(e) BS performance of very high contention producer-consumer benchmark ($c = 1000$, $i = 0$)



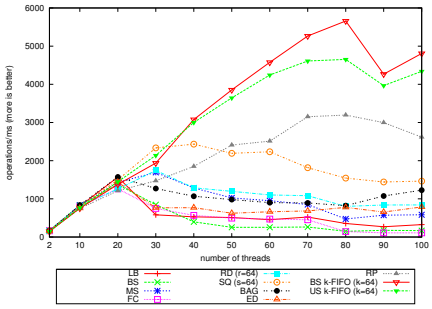
(f) BS performance of very high contention producer-consumer benchmark ($c = 1000$, $i = 5000$)

Fig. 2. Very high and low contention producer-consumer microbenchmarks with an increasing number of k for different amounts of threads on a 40-core (2 hyperthreads per core) server

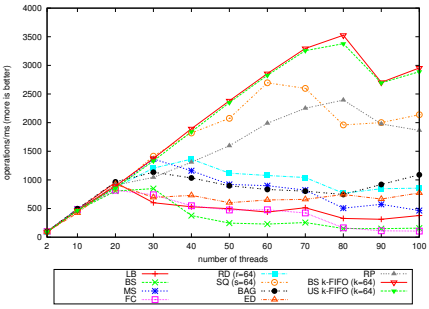
sense that there exists only a single range of close-to-optimal k. Furthermore, the population density of a k-segment that is used for dequeueing has an impact on the range of k where good performance can be observed. If k gets too large, i.e., the population in the



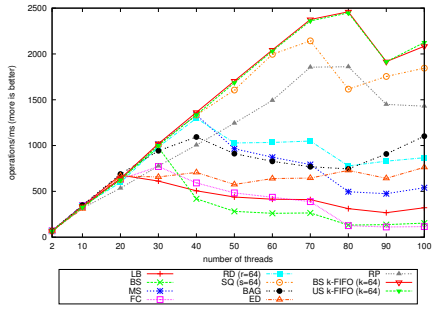
(a) Performance and scalability of very high contention producer-consumer benchmark ($c = 1000$, $i = 0$)



(b) Performance and scalability of high contention producer-consumer benchmark ($c = 4000$, $i = 0$)



(c) Performance and scalability of medium contention producer-consumer benchmark ($c = 7000$, $i = 0$)



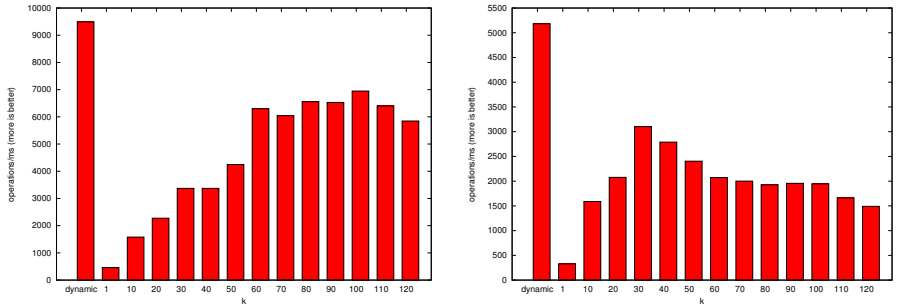
(d) Performance and scalability of low contention producer-consumer benchmark ($c = 10000$, $i = 0$)

Fig. 3. Producer-consumer microbenchmarks with an increasing number of threads on a 40-core (2 hyperthreads per core) server

dequeueing segment is sparse, the performance significantly decreases. The depicted behaviour 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 $r = s = k = 64$, which turned out be a good k (on average) 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. The BAG implementation differs from the originally proposed one in a randomized work stealing selection. This modification was necessary to make BAG competitive in our benchmarks. Figure 3(a) illustrates the results for the very high contention workload where



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

Fig. 4. Variable-load producer-consumer microbenchmarks with an increasing number of static k versus a dynamically controlled k on a 40-core (2 hyperthreads per core) server

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.

Similarly, the results with our high contention scenario, depicted in Figure 3(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 3(c) and 3(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 empty check.

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 empty 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 4(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 4(a).

6 Conclusions

We have introduced fast and scalable algorithms that implement bounded- and unbounded-size, lock-free, linearizable k-FIFO queues with empty (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.

References

1. Formal verification of an array-based nonblocking queue. In *Proc. Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 507–516. IEEE, 2005.
2. Y. Afek, M. Hakimi, and A. Morrison. Fast and scalable rendezvousing. In *Proc. International Conference on Distributed Computing (DISC)*, pages 16–31, Berlin, Heidelberg, 2011. Springer-Verlag.
3. Y. Afek, G. Korland, M. Natanzon, and N. Shavit. Scalable producer-consumer pools based on elimination-diffraction trees. In *Proc. European Conference on Parallel Processing (Euro-Par)*, pages 151–162. Springer, 2010.
4. Y. Afek, G. Korland, and E. Yanovsky. Quasi-linearizability: Relaxed consistency for improved concurrency. In *Proc. Conference on Principles of Distributed Systems (OPODIS)*, pages 395–410. Springer, 2010.
5. H. Attiya, R. Guerraoui, D. Hendler, P. Kuznetsov, M. Michael, and M. Vechev. Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. In *Proc. of Principles of Programming Languages (POPL)*, pages 487–498. ACM, 2011.
6. N. Dershowitz, D. Jayasimha, and S. Park. Bounded fairness. In *Verification: Theory and Practice*, volume 2772 of *Lecture Notes in Computer Science*, pages 440–442. Springer, 2004.
7. M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., 2008.
8. M. Herlihy and J. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.
9. D. H. I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proc. Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 355–364. ACM, 2010.
10. C. Kirsch and H. Payer. Incorrect systems: It's not the problem, it's the solution. In *Proc. Design Automation Conference (DAC)*. ACM, 2012.
11. C. Kirsch, H. Payer, H. Röck, and A. Sokolova. Brief announcement: Scalability versus semantics of concurrent FIFO queues. In *Proc. Symposium on Principles of Distributed Computing (PODC)*, pages 331–332. ACM, 2011.
12. M. Michael and M. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proc. Symposium on Principles of Distributed Computing (PODC)*, pages 267–275. ACM, 1996.
13. N. Shavit. Data structures in the multicore age. *Communications of the ACM*, 54:76–84, March 2011.
14. H. Sundell, A. Gidenstam, M. Papatriantafilou, and P. Tsigas. A lock-free algorithm for concurrent bags. In *Proc. Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 335–344, New York, NY, USA, 2011. ACM.

A *k*-FIFO Queues are Lock-free

An algorithm is lock-free if it guarantees that infinitely often some method call finishes in a finite number of steps [7]. This liveness property ensures that at least one thread in a system makes progress at any time. We show informally that *k*-FIFO queues are lock-free, c.f. [12].

In the US and BS *k*-FIFO queues an enqueue operation only loops in the following cases:

- The condition in line 6 fails. This is the case if `tail` is changed concurrently by some other thread in line 17, after the tail *k*-segment has been observed as full, or in line 46, to prohibit starvation of items in the first *k*-segment. A retry that happens because other threads observed the tail *k*-segment as full (and thus did advance `tail`) does not cause starvation, because `tail` would have to be fixed in the local thread anyways (causing a retry). Both types of retries are bounded by one loop iteration. If `tail` changes subsequently, other threads were able to perform enqueue or dequeue operations.
- The condition in line 7 never becomes `true`. This is the case if the *k*-segment is always full, meaning that other threads did succeed in enqueueing items.
- The inserting `CAS` in line 9 fails. This is the case if either the item or the ABA counter of the item in the slot changed. The item changes if some other thread was able to concurrently enqueue an item , indicating overall progress. Furthermore the `CAS` also fails (due to the ABA counter) if an item has been concurrently enqueued and dequeued in this slot. This is either the case if some threads have been faster and concurrently performing an enqueue and dequeue operation, leaving the item back as `null`, or some enqueue operation had to be undone (see next point). In both cases the system makes progress.
- The `committed` method returns `false` in line 10. This is the case if an operation successfully enqueued an item, and then successfully undid the operation by removing it. The undo step is required if both, `head` and `tail`, have been moved while an item has been inserted and the inserted item is not in the range of those two pointers. However, moving `head` and `tail` is only done while enqueueing and dequeueing items, indicating overall progress.
- (Only BS.) The condition in line 13 is `true` and the one following in line 14 is `false`. This is the case if the queue is observed as full while the head *k*-segment is empty. It is then necessary to make this head *k*-segment usable again (for insertion) by advancing `head`. This can only be a condition for starvation if other threads succeed in enqueueing and dequeueing items.

In the US and BS *k*-FIFO queues a dequeue operation only loops in the following cases:

- The condition in line 43 fails. This is the case if `head` is changed concurrently by some other thread in line 16 (BS only) or line 53, after observing that the head *k*-segment is empty. A retry that happens because other threads observed the head *k*-segment as empty (and thus did advance `head`) does not cause starvation, because

`head` would have to be fixed in the local thread anyways (causing a retry). This type of retry is bounded by one loop iteration. If `head` changes subsequently, other threads were able to dequeue items.

- The dequeueing `CAS` in line 48 fails. This is the case if either the item or the ABA counter of the item in the slot changed. The item changes if some other thread was able to concurrently dequeue an item, indicating overall progress. Furthermore the `CAS` may also fail (due to a different item, or same item with different ABA counter) if a dequeue and enqueue operation happen concurrently.
- The condition in line 51 fails. This is the case if some other thread observed the head k-segment as empty. The condition then only fails if `head` is not `tail` or `tail` changed. This means that the pointer to the empty head k-segment can be advanced, which is bounded by one round and would otherwise have to be done in the local thread.

B *k*-Segment Queue

Listing 1.2. *k*-segment queue algorithm

```
1 global head;
2 global tail;
3
4 void init():
5   new_ksegment = calloc(sizeof(ksegment));
6   head = atomic_value(new_ksegment, 0);
7   tail = atomic_value(new_ksegment, 0);
8
9 atomic_value get_head():
10  return head;
11
12 atomic_value get_tail():
13  return tail;
14
15 void advance_tail(atomic_value tail_old):
16  tail_current = get_tail();
17  if tail_current == tail_old:
18    next_ksegment = tail_old->ksegment.next;
19    if tail_old == get_tail():
20      if next_ksegment.value != null:
21        CAS(&tail, tail_old, next_ksegment);
22      else:
23        new_ksegment = calloc(sizeof(ksegment));
24        if CAS(&tail_old->ksegment.next, next_ksegment, new_ksegment):
25          CAS(&tail, tail_old, new_ksegment);
26
27 void advance_head(atomic_value head_old):
28  head_current = head;
29  if head_current = head_old:
30    tail_current = get_head();
31    tail_next_ksegment = tail_current->ksegment.next;
32    head_next_ksegment = head_current->ksegment.next;
33    if head_current == get_head():
34      if head_current.value == tail_current.value:
35        if tail_next_ksegment.value == null:
36          return;
37        if tail_current == get_tail():
38          CAS(&tail, tail_current, tail_next_ksegment);
39      head_old->item.deleted = true;
40      CAS(&head, head_old, head_next_ksegment);
```

C Unbounded-size (US) 1-FIFO Queue

Listing 1.3. Unbounded-size (US) 1-FIFO queue algorithm (`get_head`, `get_tail`, `advance_head`, and `advance_tail` are implemented by the pseudo code of the k-segment queue in Listing 1.2 with $k = 1$)

```
1 bool enqueue(item):
2   while true:
3     tail_old = get_tail();
4     head_old = get_head();
5     item_old = tail_old->item;
6     if tail_old == get_tail():
7       if item_old.value == EMPTY:
8         item_new = atomic_value(item, item_old.counter + 1);
9         if CAS(&tail_old->item, item_old, item_new):
10          return true;
11      else:
12        advance_tail(tail_old);
13
14 item dequeue():
15   while true:
16     tail_old = get_tail();
17     head_old = get_head();
18     item_old = head_old->item;
19     if head_old == head:
20       if item_old.value != EMPTY:
21         item_new = atomic_value(EMPTY, item_old.counter + 1);
22         if CAS(&head_old->item, item_old, item_new):
23           return item_old.value;
24       else:
25         if head_old.value == tail_old.value && tail_old == get_tail():
26           return null;
27         advance_head(head_old);
```