Performance, Scalability, and Semantics of Concurrent FIFO Queues*

Christoph M. Kirsch Hannes Payer Harald Röck Ana Sokolova

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

Abstract. We introduce the notion of a k-FIFO queue which may dequeue elements out of FIFO order up to a constant $k \geq 0$. Retrieving the oldest element from the queue may require up to $k+1$ dequeue operations (bounded lateness), which may return elements not younger than the $k+1$ oldest elements in the queue (bounded age) or nothing even if there are elements in the queue. A k-FIFO queue is starvation-free for finite k where $k+1$ is what we call the worst-case semantical deviation (WCSD) of the queue from a regular FIFO queue. The WCSD bounds the actual semantical deviation (ASD) of a k-FIFO queue from a regular FIFO queue when applied to a given workload. Intuitively, the ASD keeps track of the number of dequeue operations necessary to return oldest elements and the age of dequeued elements. We show that a number of existing concurrent algorithms implement k-FIFO queues whose WCSD are determined by configurable constants independent from any workload. We then introduce so-called Scal queues, which implement k-FIFO queues with generally larger, workload-dependent as well as unbounded WCSD. Since ASD cannot be obtained without prohibitive overhead we have developed a tool that computes lower bounds on ASD from time-stamped runs. Our micro- and macrobenchmarks on a state-of-the-art 40-core multiprocessor machine show that Scal queues, as an immediate consequence of their weaker WCSD, outperform and outscale existing implementations at the expense of moderately increased lower bounds on ASD.

1 Introduction

We are interested in designing and implementing concurrent FIFO queues that provide high performance and positive scalability on shared memory, multiprocessor and multicore machines. By performance we mean throughput measured in queue operations per second. Scalability is performance as a function of the number of threads in a system. The ideal result is linear scalability and high performance already with few threads. This is nevertheless an unlikely outcome on multicore hardware where shared memory access is typically orders of magnitude slower than core computation. A still challenging yet more realistic outcome and our goal in particular is positive scalability, i.e., increasing performance with an increasing number of threads, up to as many

* This work has been supported by the Austrian Science Fund (National Research Network RiSE on Rigorous Systems Engineering S11404-N23 and Elise Richter Fellowship V00125) and the National Science Foundation (CNS1136141).

threads as possible, and high performance already with few threads. Achieving both performance and scalability is important since positive scalability but low performance with few threads may be even worse than negative scalability.

The key to high performance and positive scalability is parallelization with low sequential overhead. Earlier attempts to improve the performance and scalability of simple, lock-based FIFO queues include the lock-free Michael-Scott FIFO Queue [13] and, more recently, the Flat-Combining FIFO Queue [11], which we have both implemented for our experiments. Both algorithms tend to provide better performance and scale to more threads than lock-based FIFO queues. Another two recent examples of algorithms that aim at improving performance and scalability are the Random Dequeue Queue [1] and the Segment Queue [1], which we have also implemented and study here. An important difference to the former two algorithms is that the latter two provide only relaxed FIFO semantics in the sense that elements may be returned out of FIFO order. The goal is to increase parallelism further while the challenge is to maintain bounds on the relaxation of semantics.

Based on the same principle of improving performance and scalability at the expense of strict FIFO semantics we propose Scal queues for implementing FIFO queues with relaxed semantics. The idea is to maintain (a distributed system of) p instances of a regular FIFO queue (we chose Michael-Scott for our experiments) and then select, upon each enqueue or dequeue operation, one of the p instances before performing the operation on the selected instance without further coordination with the other instances. Thus up to p queueing operations may be performed in parallel. Selection is done by a load balancer whose implementation has an immediate impact on performance, scalability, and semantics. In particular, the load balancer determines how close the semantics of the Scal queue is to the semantics of a regular FIFO queue. We have implemented a variety of load balancers for our experiments to study the trade-off between performance, scalability, and semantics with Scal queues relative to the previously mentioned queues.

With the straightforward metric of operation throughput in place for measuring performance, the only remaining challenge is to quantify and to measure difference in semantics. For this purpose, we introduce the notion of semantical deviation as a metric for quantifying the difference in semantics between a queue with relaxed FIFO semantics and a regular FIFO queue. Intuitively, when running a given queue implementation on some workload, semantical deviation keeps track of the number of dequeue operations necessary to return oldest elements and the age of dequeued elements. However, measuring actual semantical deviation on existing hardware is only possible indirectly and approximatively through time-stamping invocation and response events of operations online, and then computing offline, using a tool that we developed, an approximation of the actual run that took place. The approximation is a sequence of linearization points that leads to a lower bound on the actual semantical deviation.

Here a key observation is that there exist upper bounds on semantical deviation independent of at least all workloads in a given class (e.g. with a fixed number of threads) for most of the implementations we consider. It turns out that these implementations are instances of the notion of a k-FIFO queue for different $k \geq 0$ where $k+1$ is their worst-case semantical deviation from a regular FIFO queue. A k-FIFO queue may dequeue elements out of FIFO order up to k. In particular, retrieving the oldest element from

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

$$dequeue_k(e)(q,l) = \begin{cases} (\varepsilon, 0) & \text{if } e = null, q = \varepsilon \quad \text{(L1)} \\ (q, l+1) & \text{if } e = null, q \neq \varepsilon, \text{(C1) } l < k \quad \text{(L2)} \\ (q', 0) & \text{if } q = e \cdot q' \quad \text{(L3)} \\ (e_1 \ldots e_{i-1} e_{i+1} \ldots e_n, l+1) & \text{if } e = e_i, q = e_1 \ldots e_n, \quad \text{(L4)} \\ & 1 < i \leq n, \text{(C2) } l < k, \\ & \text{(C3) } i \leq k+1-l \\ error & \text{otherwise} \quad \text{(L5)} \end{cases}$$

Fig. 1. Sequential specification of a k-FIFO queue (a FIFO queue w/o lines (L2), (L4); a POOL w/o conditions (C1), (C2), (C3))

the queue may require up to $k+1$ dequeue operations (bounded lateness), which may return elements not younger than the $k+1$ oldest elements in the queue (bounded age) or nothing even if there are elements in the queue. Depending on the implementation k may or may not depend on workload characteristics such as number of threads or may even be probabilistic. The non-determinism in the choice of elements to be returned provides the potential for performance and scalability which, in our benchmarks, tend to increase with increasing k.

We summarize the contributions of this paper: (1) the notion of k-FIFO queues (previously presented in a brief announcement [12]), (2) Scal queues, (3) the notion of semantical deviation, and (4) micro- and macrobenchmarks showing the trade-off between performance, scalability, and semantics.

In Section 2, we formally define k-FIFO queues and then discuss the existing concurrent algorithms we consider. In Section 3, we introduce Scal queues along with the load balancers we have designed and implemented. Semantical deviation is defined in Section 4. Related work is discussed in Section 5, our experiments are presented in Section 6, and conclusions are in Section 7.

2 k-FIFO Queues

We introduce the notion of a k-FIFO queue where $k \geq 0$. Similar to a regular FIFO queue, a k-FIFO queue provides an enqueue and a dequeue operation but with a strictly more general semantics defined as follows. Let the tuple (q,l) denote the state of a k-FIFO queue where q is the sequence of queue elements and l is an integer, called lateness, that counts the number of dequeue operations since the most recent dequeue operation removed the oldest element. The initial, empty state of a k-FIFO queue is $(\varepsilon, 0)$. The enqueue operation of a k-FIFO queue is a function from queue states and queue elements to queue states. The dequeue operation is a function from queue states and queue elements or the *null* return value to queue states. The formal definition of the semantics (sequential specification) of a k-FIFO queue is shown in Figure 1. In order to keep the definition simple we assume, without loss of generality, that queue elements are unique, i.e., each element will only be enqueued once and discarded when dequeued.

A k-FIFO queue is a queue where an enqueue operation, as usual, adds an element to the queue tail. A dequeue operation, however, may either return nothing (*null*) al-

though there could be elements in the queue, or else remove one of the $k+1-l$ oldest elements from the queue with $l < k$ again being the number of invoked dequeue operations since the most recent dequeue operation that removed the oldest element from the queue. Retrieving the oldest element from the queue may require up to $k+1$ dequeue operations (bounded lateness), which may return *null* or elements not younger than the $k+1-l$ oldest elements in the queue (bounded age) and which may be interleaved with any number of enqueue operations. Thus k-FIFO queues are starvation-free for finite k and a 0-FIFO queue is a regular FIFO queue.

The standard definition of a regular FIFO queue can be obtained from Figure 1 by dropping lines (L2) and (L4). Just dropping line (L2) provides the definition of a k-FIFO queue without *null* returns if non-empty, which is a special case that we have implemented for some queues. Other combinations may also be meaningful, e.g. dropping conditions (C1), (C2), and (C3) defines the semantics of a POOL. In other words, a POOL is equivalent to a k-FIFO queue with unbounded k.

2.1 Implementations

We study different implementations of k-FIFO queues with k independent from any workload as well as k dependent on workload parameters such as the number of threads. In particular, k may or may not be configurable for a given implementation.

The following queues implement regular FIFO queues: a standard lock-based FIFO queue (LB), the lock-free Michael-Scott FIFO queue (MS) [13], and the flat-combining FIFO queue (FC) [11]. LB locks a mutex for each data structure operation. With MS each thread uses at least two compare-and-swap (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 Random Dequeue Queue (RD) [1] is a k-FIFO queue 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. If the element is not the oldest element in the queue it is marked as dequeued and returned but not removed from the queue. If the element is already marked as dequeued the process is repeated until a not-dequeued element is found or the queue is empty. If the element is the oldest element the queue head is set to the first not-dequeued element and all elements in between are removed. Hence RD may be out-of-FIFO order by at most r and always returns an element when the queue is not empty. RD was originally not defined as a k-FIFO queue but introduced in the context of relaxing the consistency condition linearizability [1].

The Segment Queue (SQ) [1] is a k-FIFO queue implemented by a non-blocking FIFO queue of segments. 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* if only one segment containing up to $s-1$ elements remains in the queue. The thread returns *null* if other threads dequeued the up to $s-1$ elements before the thread could find them. Hence, $k = s$ for SQ. SQ was originally not defined as a k-FIFO queue but introduced in the context of relaxing the consistency condition linearizability [1].

Next, we discuss new implementations of k-FIFO queues where k depends not only on constant numbers but also on the workload such as the number of threads or is even unbounded and may only be determined probabilistically.

3 Scal Queues

Scal is a framework for implementing k-FIFO queues as well as potentially other concurrent data structures such as relaxed versions of stacks and priority queues that may provide bounded out-of-order behavior. In this paper we focus on k-FIFO queues and leave other concurrent data structures for future work. In the sequel we refer to k-FIFO queues implemented with Scal as Scal queues.

Scal is motivated by distributed systems where shared resources are distributed and access to them is coordinated globally or locally. For implementing k-FIFO queues Scal uses p instances of a regular FIFO queue, so-called partial FIFO queues, and a load balancer that distributes queueing operations among the p partial FIFO queues. Upon the invocation of a queueing operation the load balancer first selects one of the p partial FIFO queues and then calls the actual queueing operation on the selected queue. The value of p and the type of load balancer determine k, as discussed below, as well as the performance and scalability of Scal queues, i.e., how many queueing operations can potentially be performed concurrently and in parallel, and at which cost without causing contention. Moreover, in our Scal queue implementations selection and queueing are performed non-atomically for better performance and scalability. Thus k with Scal queues depends on the number of threads in the system since between selection and queueing of a given thread all other threads may run. The semantics of Scal queues may nevertheless be significantly closer to FIFO semantics than what the value of k may suggest because of the low probability of the worst case, as shown in Section 6. Note that p and the load balancer may be configured at compile time or dynamically at runtime with the help of performance counters. For example, a load balancer may be chosen with $p = 1$ under low contention and with increasing p as contention increases. Dynamic reconfiguration is future work.

Round-Robin Load Balancing We have implemented a round-robin load balancer (RR) for Scal that selects partial FIFO queues for enqueue and dequeue operations in round-robin fashion. Two global counters keep track on which of the p partial FIFO queues the last enqueue and the last dequeue operation was performed. The counters are accessed and modified using atomic operations, which can cause contention. However, scalability may still be achieved under low contention since the load balancer itself is simple. A Scal queue using RR implements a k-FIFO queue with $k = t \cdot (p-1)$ where t is an upper bound on the number of threads in the system. Note that k comes down to $p-1$ if selection and queueing are performed atomically.

Randomized Load Balancing Another approach is to use a randomized load balancer (RA) for Scal that randomly distributes operations over partial FIFO queues. Randomized load balancing [4, 16, 6] has been shown to provide good distribution quality if the random numbers are distributed independently and uniformly. However, generating such random numbers may be computationally expensive. Therefore, it is essential to find the right trade-off between quality and overhead of random number generation. We use an efficient random number generator that produces evenly distributed random numbers [15]. The value of k for RA Scal queues is unbounded but may be determined probabilistically as part of future work. A first step is to determine the maximum imbalance of the partial FIFO queues. Suppose that t threads have performed m operations each on p partial FIFO queues using RA. Then, with a probability of at least $1 - O\left(\frac{1}{p}\right)$, the maximum difference (imbalance) between the number of elements in any partial FIFO queue and the average number of elements in all partial FIFO queues is $\Theta\left(\sqrt{\frac{t \cdot m \cdot \log p}{p}}\right)$ [16] if selection and queueing are performed atomically. However, as previously mentioned, selection and queueing are performed non-atomically in our implementation. The presented maximum imbalance is anyway relevant for a comparison with a refined version of RA discussed next.

In order to improve the load balancing quality of RA, d partial FIFO queues with $1 < d \leq p$ may be chosen randomly. Out of the d partial FIFO queues the queue that contributes most to a better load balance is then selected. More precisely, enqueue and dequeue operations are performed on the partial FIFO queues that contain among the d partial FIFO queues the fewest and the most elements, respectively. We refer to such a load balancer as d-randomized load balancer (dRA). The runtime overhead of dRA increases linearly in d since the random number generator is called d times. Thus d allows us to trade off balancing quality and global coordination overhead. Here, again the value of k for dRA is unbounded. However, again with a probability of at least $1 - O(\frac{1}{p})$, the maximum difference (imbalance) between the number of elements in any partial FIFO queue and the average number of elements in all partial FIFO queues is now $\Theta\left(\frac{\log \log p}{d}\right)$ [6] if selection and queueing are performed atomically. Again, determining the maximum imbalance for the case when selection and queueing are performed non-atomically, as in our implementation, is future work. However, the presented maximum imbalance shows an important difference to RA Scal queues. It is independent of the state of the Scal queue, i.e., the history of enqueue and dequeue operations. In particular, $d = 2$ leads to an exponential improvement in the balancing quality in comparison to RA. Note that $d > 2$ further improves the balancing quality only by a constant factor [6] at the cost of higher computational overhead.

Hierarchical Load Balancing With hierarchical load balancing p partial FIFO queues are partitioned into $0 < h \leq p$ non-overlapping subsets. In this paper we use a two-level hierarchy where the high-level load balancer chooses the subset and the low-level load balancer chooses one of the partial FIFO queues in the given subset. For partitioning we take the cache architecture of the system into account by making the subsets processor-local, i.e., h is here the number of processors of the system. For the high-level load balancer we use a weighted randomized load balancer where the thread running on pro-

cessor i chooses the processor-local subset i with a given probability w while one of the remaining subsets is chosen with probability $1-w$. This allows us to increase cache utilization and reduce the number of cache misses. On the lower level we use a randomized (H-RA) or 2-randomized (H-2RA) load balancer to choose the actual partial FIFO queue. Note that in principle multiple hierarchies of load balancers could be used and in each hierarchy a different load balancer could run. The value of k for H-RA and H-2RA Scal queues is again unbounded but may be determined probabilistically similar to the value of k for RA and 2RA Scal queues, respectively.

Backoff Algorithm We have implemented two so-called backoff algorithms for dequeue operations to avoid *null* returns on non-empty Scal queues. In particular, we have implemented a perfect backoff algorithm (no *null* returns if queue is non-empty) based on the number of elements in a Scal queue as well as a heuristic backoff algorithm (no *null* returns, if queue is non-empty, with high probability given a sufficiently high retry threshold).

In the perfect backoff algorithm a global counter holds the number of elements in a Scal queue. The counter is incremented after a successful enqueue operation and decremented after a successful dequeue operation. If a dequeue operation ends up at an empty partial FIFO queue the backoff algorithm inspects the counter. If it indicates that the Scal queue is not empty the load balancer selects another partial FIFO queue. Updating and inspecting the global counter requires synchronization and can lead to cache conflicts, which may limit performance and scalability.

The heuristic backoff algorithm may simply retry a given number of times determined at compile-time before having the dequeue operation return *null*. The average number of retries depends on different factors such as the application workload. In the experiments in Section 6 we use a heuristic backoff algorithm with a maximum retry threshold set high enough to avoid *null* returns on non-empty Scal queues.

4 Semantical Deviation

We are interested in what we call the semantical deviation of a k-FIFO queue from a regular FIFO queue when applied to a given workload. Semantical deviation captures how many dequeue operations it took to return oldest elements (lateness) and what the age of dequeued elements was. Since semantical deviation cannot be measured efficiently without introducing prohibitive measurement overhead we propose lower and upper bounds of which the lower bounds can be computed efficiently from time-stamped runs of k-FIFO queue implementations. Our experimental results show that the lower bounds at least enable a relative, approximative comparison of different implementations in terms of their actual semantical deviation. Computing the upper bounds remains future work.

We represent a workload applied to a queue by a so-called (concurrent) history H, which is a finite sequence of invocation and response events of enqueue and dequeue operations [10]. We work with complete histories, i.e., histories in which each operation has a corresponding invocation and response event and the invocation is before the response event. By $\langle op$ and by $op\rangle$ we denote the invocation and response events of the

operation op, respectively. Two operations op_1 and op_2 in a history H are overlapping if the response event $op_1\rangle$ is after the invocation event $\langle op_2$ and before the response event $op_2\rangle$, or vice versa. An operation op_1 precedes another operation op_2 in a history H, if the response event $op_1\rangle$ is before the invocation event $\langle op_2$. Two histories are equivalent if the one is a permutation of the other in which precedences are preserved (only events of overlapping operations may commute). A history H is sequential if the first event of H is an invocation event and each invocation event is immediately followed by a matching response event [10]. Equivalently, a sequential history is a sequence of enqueue and dequeue operations.

Given a sequential specification C (here FIFO, k-FIFO, or POOL), an execution sequence corresponding to a sequential history $H_S = op_1 \dots\ op_m$ is a sequence of states $C(H_S) = s_0 s_1 \dots\ s_m$ starting from the initial state s_0 with $s_{j+1} = op_{j+1}(s_j)$ for $j = 0, \dots, m-1$. The sequential history H_S is valid with respect to the sequential specification C if no s_i in $C(H_S)$ is the error state *error*.

In particular, for a sequential history $H_S = op_1 \dots\ op_m$, FIFO(H_S) is the sequence of FIFO queue states obtained from the sequential specification of Figure 1 without lines (L2) and (L4), where $s_0 = (\varepsilon, 0)$ and $s_j = (q_j, l_j)$ with $l_j = 0$ for $j = 0, \dots, m$; k-FIFO(H_S) is the sequence of k-FIFO queue states obtained from the sequential specification of Figure 1, where $s_0 = (\varepsilon, 0)$. If H_S is valid with respect to FIFO, FIFO-valid for short, i.e., if no queue state in FIFO(H_S) is the error state *error*, then each dequeue operation in H_S returns the head of the queue or *null* if the queue is empty. Similarly, if H_S is valid with respect to k-FIFO, k-FIFO-valid for short, then each dequeue operation in H_S returns one of the $k+1-l$ oldest elements in the queue (or *null*) and queue heads are always returned in H_S in at most $k+1$ steps. Every FIFO-valid sequential history is k-FIFO-valid.

We next define the notion of semantical deviation of a sequential history and characterize validity in terms of it. In order to do that we need the sequential specification of a POOL. Given a sequential history $H_S = op_1 \dots\ op_m$,

$$\text{POOL}(H_S) = (q_0, l_0)(q_1, l_1) \dots\ (q_m, l_m)$$

is the sequence of POOL states obtained from the sequential specification of Figure 1 without the conditions (C1), (C2), and (C3), where $(q_0, l_0) = (\varepsilon, 0)$.

From POOL(H_S) we quantify bounded fairness through (maximum) lateness of H_S, denoted $L(H_S)$, which is the maximum number of dequeue operations it ever took in H_S to return an oldest element, i.e.,

$$L(H_S) = max_{1 \le j \le m}(l_j).$$

The average lateness $ML(H_S)$ is the mean of the number of dequeue operations it took to return all oldest elements in H_S, i.e.,

$$ML(H_S) = mean(\{l_{j-1} \mid l_j = 0, j \in \{1, \dots, m\}\}).$$

From POOL(H_S) we also define the sequence of (inverse) ages $a_0 a_1 \dots\ a_m$ of H_S by

$$a_j = \begin{cases} i-1 & \text{if } q_j = e_1 \dots\ e_n, q_{j+1} = e_1 \dots\ e_{i-1} e_{i+1} \dots\ e_n \\ 0 & \text{otherwise} \end{cases}$$

The (minimum inverse) age of H_S, denoted $A(H_S)$, is the (inverse) age of the youngest element ever returned in H_S, i.e.,

$$A(H_S) = max_{1 \leq j \leq m}(a_j).$$

The average (inverse) age $MA(H_S)$ is the mean of the (inverse) ages of all elements returned in H_S, i.e.,

$$MA(H_S) = mean_{1 \leq j \leq m}(a_j).$$

Finally, the (maximum) semantical deviation of H_S, denoted $SD(H_S)$, is the maximum of the sums of the lateness and (inverse) age pairs obtained from POOL(H_S), i.e.,

$$SD(H_S) = max_{1 \leq j \leq m}(l_j + a_j).$$

Similarly, the average semantical deviation is

$$MSD(H_S) = mean_{1 \leq j \leq m}(l_j + a_j).$$

We are now ready to present the characterization of k-FIFO validity in terms of lateness, age, and semantical deviation.

Proposition 1 *A sequential history H_S is k-FIFO-valid if and only if $L(H_S) \leq k$, $A(H_S) \leq k$, and $SD(H_S) \leq k+1$.*

Finally, we recall the notion of linearizability [10] before introducing the remaining concepts. Given a history H and a sequential specification C, $lin(H,C)$ denotes the set of all sequential histories that are equivalent to H and valid with respect to C. If $lin(H,C)$ is not empty, H is said to be linearizable with respect to C [10]. Hence, H is linearizable with respect to FIFO if there is a sequential history H_S equivalent to H that is FIFO-valid; it is linearizable with respect to k-FIFO if there is a sequential history H_S equivalent to H that is k-FIFO-valid. Note that every history linearizable with respect to FIFO is linearizable with respect to k-FIFO as well. A concurrent implementation of a sequential specification is said to be linearizable if all histories that can be obtained with the implementation are linearizable [10]. Linearizability is thus a consistency condition for specifying the semantics of objects in the presence of concurrency. The implementations of all (k-FIFO) queues discussed in this paper are linearizable.

In general, $lin(H,C)$ may contain more than one sequential history if H is linearizable with respect to C. However, we are only interested in the sequential history H_A in $lin(H,C)$ that represents the run that was actually performed. In particular, we are interested in the actual semantical deviation $SD(H_A)$ of H (ASD for short), and similarly in $L(H_A)$ and $A(H_A)$. Unfortunately, H_A cannot be determined on existing hardware without introducing prohibitive overhead. In practice, only H can be obtained efficiently by time-stamping the invocation and response events of all operations. We therefore propose to approximate H_A by computing two sequential histories H_L and H_H in $lin(H,C)$ such that

$$\begin{aligned} L(H_L) &= min(\{L(H_S) | H_S \in lin(H,C)\}) \\ A(H_L) &= min(\{A(H_S) | H_S \in lin(H,C)\}) \\ SD(H_L) &= min(\{SD(H_S) | H_S \in lin(H,C)\}) \end{aligned}$$

and, similarly for H_H with *min* replaced by *max*, holds.

The following proposition is a consequence of Proposition 1 and the definition of a k-FIFO queue.

Proposition 2 *For all histories H of a linearizable implementation of a k-FIFO queue we have that*

$$\begin{aligned} L(H_L) \leq\ L(H_A)\ \leq\ L(H_H)\ \leq k \\ A(H_L) \leq\ A(H_A)\ \leq\ A(H_H)\ \leq k \\ SD(H_L) \leq SD(H_A) \leq SD(H_H) \leq k+1 \end{aligned}$$

and $SD(H_H) = 0$ for $k = 0$.

We therefore call k the worst-case lateness (WCL) and worst-case age (WCA), and $k+1$ for $k > 0$ and 0 for $k = 0$ the worst-case semantical deviation (WCSD) of a k-FIFO queue.

4.1 Computing H_L

We have designed and implemented a tool that computes H_L from a given history H without enumerating $lin(H,C)$ explicitly (assuming that the sequential specification C is POOL not knowing any k in particular). The tool scans H for invocation events of dequeue operations in the order of their appearance in H to construct H_L in a single pass (and POOL(H_L) to keep track of the queue states and lateness). For each invocation event $\langle op$ of a dequeue operation op the following computation is performed until a linearization point for op has been created: (1) if op returns *null* remember the (inverse) age for op as zero, otherwise compute and remember the (inverse) age for op assuming that the linearization point of the enqueue operation that matches op is as far in the past as possible under the precedence constraints in H, (2) repeat (1) for all dequeue operations that overlap with op and are not preceded by any other dequeue operations that also overlap with op, (3) among the dequeue operations considered in (1) and (2) find the dequeue operation op' that returns an element other than *null* and has the minimum remembered (inverse) age (any such op' will do if multiple exist), or else if only dequeue operations that return *null* have been considered in (1) and (2) then take any of those as op', and finally (4) create a linearization point in H_L for the enqueue operation that matches op' and move that point under the precedence constraints in H as far into the past as possible and create a linearization point for op' in H_L right before the invocation event $\langle op$. Note that after creating a linearization point for an operation its invocation and response events are not considered anymore in subsequent computations. The key insight for correctness is that bringing operations forward with minimum (inverse) age also minimizes lateness and thus produces H_L. In contrast to H_L, computing H_H may require exploring all possible permutations of overlapping operations, which is computationally expensive, in particular for histories obtained from k-FIFO queue implementations with large or even unbounded k.

5 Related Work

We relate the notions of a k-FIFO queue and semantical deviation as well as the concept, design, and implementation of Scal queues to other work.

The topic of this paper is part of a recent trend towards scalable but semantically weaker concurrent data structures [17] acknowledging the intrinsic difficulties of implementing deterministic semantics in the presence of concurrency [3]. The idea is to address the multicore scalability challenge by leveraging non-determinism in concurrent data structure semantics for better performance and scalability. In the context of concurrent FIFO queues, the notion of a k-FIFO queue is an attempt to capture the degree of non-determinism and its impact on performance and scalability in a single parameter. Many existing implementations of concurrent FIFO queues (with or without relaxed semantics) are instances of k-FIFO queues. The implementations we consider here [13, 11, 1] are only a subset of the available choices [9]. Other implementations such as work stealing queues which may return the same element multiple times before removing it are not instances of k-FIFO queues but are anyway related in high-level objective and principle [14]. The concept of Scal queues can be seen as an example of best-effort computing [7] where inaccuracies introduced on a given level in a system may lead to better overall performance but must then be dealt with on a higher level.

The notion of semantical deviation is a metric for quantifying the difference in semantics between a queue with relaxed FIFO semantics and a regular FIFO queue. Another approach for relaxing the sequential specification is quasi-linearizability [1]. As opposed to relaxing the sequential specification of a concurrent object one can also relax the consistency condition. An example of a more relaxed consistency condition than linearizability is quiescent consistency [2], for which concurrent data structure implementations exist which may provide superior performance in comparison to their linearizable counterparts. A comprehensive overview of variants of weaker and stronger consistency conditions than linearizability can be found in [9].

6 Experiments

We evaluate performance, scalability, and semantics of the k-FIFO queue implementations described in Section 2.1 and Section 3. 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 study the LB, MS, FC, RD, SQ, and Scal queues (with the RR, RA, 2RA, H-RA, and H-2RA load balancers without backoff as well as the RR-B, RA-B, 2RA-B, H-RA-B, and H-2RA-B load balancers with backoff). The partial FIFO queues of the Scal queues are implemented with MS. For the RD, SQ, and Scal queues we use $r = s = p = 80$. For the hierarchical load balancers we use $h = 4$ (number of processors) and $w = 0.9$.

All benchmarked algorithms are implemented in C and compiled using gcc 4.3.3 with -O3 optimizations. In all experiments the benchmark threads are executed with real-time priorities to minimize system jitter. The threads are scheduled by the default Linux scheduler and not pinned to cores. Each thread pre-allocates and touches a large block of memory to avoid subsequent demand paging, and then allocates and deallocates thread-locally all queue elements from this block to minimize cache misses and to avoid potential scalability issues introduced by the underlying memory allocator.

6.1 Microbenchmarks

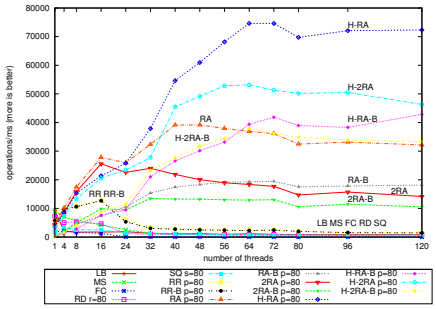
We designed and implemented a framework to microbenchmark and analyze different queue implementations under configurable contention. The framework emulates a multi-threaded producer-consumer setup where each thread enqueues and dequeues in alternating order an element to a shared queue. The framework allows to specify the number of threads, the number of elements each thread enqueues and dequeues, how much computation is performed between queueing operations, and which queue implementation to use. We focus on two microbenchmark configurations: 1. a high contention configuration where no computational load in between queueing operations is performed and 2. a low contention configuration where in between any two queueing operations additional computational load is created by executing an iterative algorithm that calculates in 500 loop iterations an approximation of π which takes in total on average 1130ns.

We evaluate each queue implementation with an increasing number of threads and determine its performance, scalability, and semantics. Performance is shown in number of operations performed per millisecond. Scalability is performance with an increasing number of threads. Semantics is average semantical deviation as computed by our tool described in Section 4.

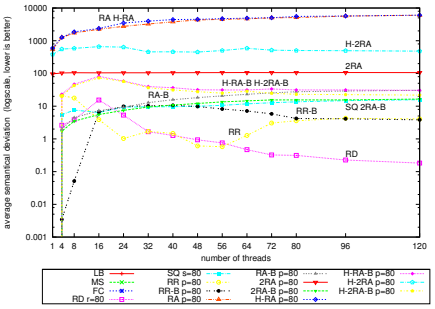
Figure 2(a) depicts the performance result of the high contention benchmark. The throughput of LB, MS, FC, RD, and SQ decreases with an increasing number of threads. RR does not scale but performs better then the non-Scal queues. The non-backoff Scal queues provide better performance then their backoff counterparts. This is due to the fact that in the non-backoff case a Scal queue may return *null* if no element is found in the selected partial FIFO queue whereas in the backoff case it has to retry. The best performance is provided by the hierarchical Scal queues which scale up to the number of hardware threads in the system, due to better cache utilization. For all other instances the number of L3 cache misses significantly increases between 20 and 32 threads.

The average semantical deviation of the experimental runs or the high contention benchmark are depicted in Figure 2(b). Note that the graphs for regular FIFO queues, i.e., LB, MS, and FC, are not visible since their semantical deviation is always zero. Using a backoff algorithm for Scal queues generally improves the average semantical deviation by an order of magnitude, except for RR and RR-B. Scal queues that show higher average semantical deviation clearly outperform the other queues in terms of performance and scalability. The Scal queue with H-2RA load balancing appears to offer the best trade-off between performance, scalability, and semantics on the workload and hardware considered here.

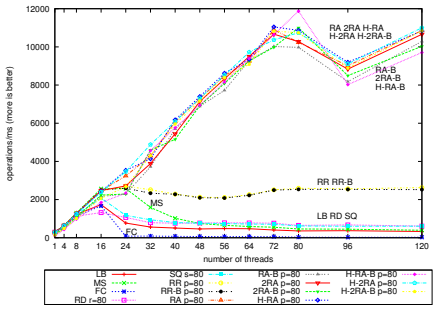
Figures 2(c) and 2(d) depict the performance result of the low contention benchmark. The LB, MS, FC, RD, and SQ queues scale for up to 8 threads. Between 8 to 16 threads throughput increases only slightly. With more than 16 threads scalability is negative. The RR Scal queue scales for up to 16 threads and then maintains throughput. The other Scal queues provide scalability up to the number of hardware threads in the system. The performance difference between backoff and non-backoff is less significant in the presence of additional computational load. The best performance and scalability is still provided by the hierarchical Scal queues but the difference to the non-hierarchical versions is significantly smaller. The number of L3 cache misses are similar



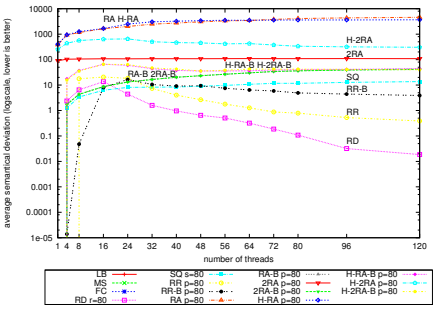
(a) Performance and scalability of high contention benchmark



(b) Average semantical deviation of high contention benchmark



(c) Performance and scalability of low contention benchmark



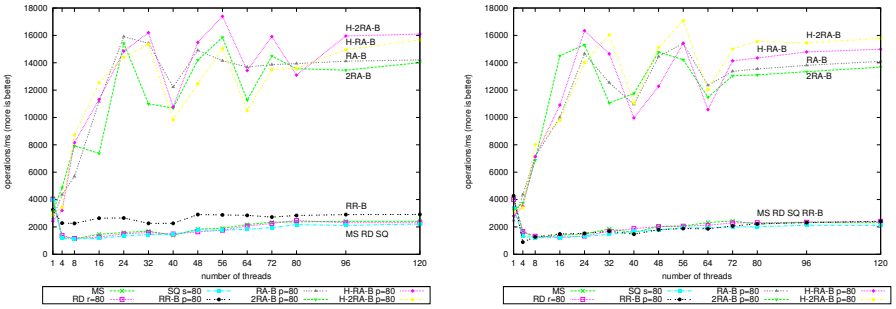
(d) Average semantical deviation of low contention benchmark

Fig. 2. High and low contention microbenchmarks with an increasing number of threads on a 40-core (2 hyperthreads per core) server

to the high contention case. Again, the hierarchical Scal queues have the lowest number of L3 cache misses. The additional computational load between queueing operations does not change average semantical deviation significantly on the workload considered here, see Figure 2(d).

6.2 Macrobenchmarks

We ran two macrobenchmarks with parallel versions of transitive closure and spanning tree graph algorithms [5] using random graphs consisting of 1000000 vertices where 1000000 unique edges got randomly added to the vertices. All threads start operating on the graph at different 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). If a neighboring vertex already got processed by a different thread then the vertex is ignored. Vertices to be processed are kept in a global queue (which we implemented with a representative selection of our queue implementations). When a vertex is processed, then it is removed from the queue and all its unprocessed neighbors are added to the queue. The graph algorithm terminates when the global queue is empty. Thus we need to use backoff in these experiments to guarantee correct



(a) Spanning tree (b) Transitive closure

Fig. 3. Performance and Scalability of macrobenchmarks on a random graph with 1000000 vertices and 1000000 edges with an increasing number of threads on a 40-core (2 hyperthreads per core) server

termination. Note that both algorithms tolerate any particular order of elements in the global queue.

The macrobenchmark results are presented in Figure 3. Each run was repeated 10 times. We present the average number of operations per milliseconds of the 10 runs as our performance metric. The Scal queues with RA-B, 2RA-B, H-RA-B, and H-2RA-B clearly outperform MS, RD, and SQ. The RR-B Scal queue provides a small performance improvement in the spanning tree case and no improvement in the transitive closure case. Both algorithms may produce high cache-miss rates since accessing the neighbors of a vertex may result in disjoint cache line accesses. A graph representation that takes hardware features into account may improve scalability further.

7 Conclusions

We have introduced the notion of a k-FIFO queue which may dequeue elements out of FIFO order up to k. Several existing queue implementations are instances of k-FIFO queues for different k. We have also introduced Scal queues, which aim at improving performance and scalability of FIFO queue implementations through load balancing by distributing queueing operations across multiple, independent queue instances. Load balancing directly determines performance, scalability, and semantics of Scal queues, in particular how close the queueing behavior is to FIFO. In order to quantify the difference between actual and regular FIFO semantics, we have introduced the notion of semantical deviation, which captures how many dequeue operations it took to return oldest elements (lateness) and what the age of dequeued elements was. Our experiments show that Scal queues with a memory-hierarchy-aware combination of randomized and queue-size-based load balancing (H-2RA) offer the best trade-off between performance, scalability, and semantics on the considered workloads and hardware.

We see many interesting directions for future work. Which applications tolerate semantical deviation to what extent? Is the parameter k the right choice of information that should be exposed to application programmers for performance-oriented multicore

programming (rather than, e.g. the memory hierarchy)? Can concurrent data structures other than FIFO queues be relaxed in a similar way? In recent collaboration [8] we have developed a framework for quantitative relaxation of concurrent data structures which covers FIFO queues but also stacks, priority queues, and other examples.

References

1. 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.
2. J. Aspnes, M. Herlihy, and N. Shavit. Counting networks. *Journal of the ACM*, 41:1020–1048, 1994.
3. 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.
4. Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal. Balanced allocations (extended abstract). In *Proc. Symposium on Theory of computing (STOC)*, pages 593–602. ACM, 1994.
5. D. Bader and G. Cong. A fast, parallel spanning tree algorithm for symmetric multiprocessors (smps). *Journal of Parallel and Distributed Computing*, 65:994–1006, 2005.
6. P. Berenbrink, A. Czumaj, A. Steger, and B. Vöcking. Balanced allocations: The heavily loaded case. *SIAM Journal on Computing*, 35(6):1350–1385, 2006.
7. S. Chakradhar and A. Raghunathan. Best-effort computing: re-thinking parallel software and hardware. In *Proc. Design Automation Conference*, pages 865–870. ACM, 2010.
8. T. Henzinger, C. Kirsch, H. Payer, A. Sezgin, and A. Sokolova. Quantitative relaxation of concurrent data structures. Technical Report 2012-03, Department of Computer Sciences, University of Salzburg, May 2012.
9. M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., 2008.
10. 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.
11. 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.
12. 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)*. ACM, 2011.
13. 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.
14. M. Michael, M. Vechev, and V. Saraswat. Idempotent work stealing. In *Proc. Principles and Practice of Parallel Programming (PPoPP)*, pages 45–54. ACM, 2009.
15. S. Park and K. Miller. Random number generators: good ones are hard to find. *Communications of the ACM*, 31(10):1192–1201, 1988.
16. M. Raab and A. Steger. "Balls into Bins" - A Simple and Tight Analysis. In *Proc. Workshop on Randomization and Approximation Techniques in Computer Science (RANDOM)*, pages 159–170. Springer, 1998.
17. N. Shavit. Data structures in the multicore age. *Communications ACM*, 54:76–84, March 2011.