# How FIFO is Your Concurrent FIFO Queue? *

Andreas Haas, Christoph M. Kirsch, Michael Lippautz, Hannes Payer
University of Salzburg
firstname.lastname@cs.uni-salzburg.at

## Abstract

Designing and implementing high-performance concurrent data structures whose access performance scales on multicore hardware is difficult. Concurrent implementations of FIFO queues, for example, seem to require algorithms that efficiently increase the potential for parallel access by implementing semantically relaxed rather than strict FIFO queues where elements may be returned in some out-of-order fashion. However, we show experimentally that the on average shorter execution time of enqueue and dequeue operations of fast but relaxed implementations may offset the effect of semantical relaxations making them appear as behaving more FIFO than strict but slow implementations. Our key assumption is that ideal concurrent data structure operations should execute in zero time. We define two metrics, element-fairness and operation-fairness, to measure the degree of element and operation reordering, respectively, assuming operations take zero time. Element-fairness quantifies the deviation from FIFO queue semantics had all operations executed in zero time. With this metric even strict implementations of FIFO queues are not FIFO. Operation-fairness helps explaining element-fairness by quantifying operation reordering when considering the actual time operations took effect relative to their invocation time. In our experiments, the effect of poor operation-fairness of strict but slow implementations on element-fairness may outweigh the effect of semantical relaxation of fast but relaxed implementations.

***Categories and Subject Descriptors*** D.1.3 [*Concurrent Programming*]: Parallel Programming

***General Terms*** Measurement

***Keywords*** Zero-Time Linearization, Element-Fairness, Operation-Fairness

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


*RACES'12,* October 21, 2012, Tucson, Arizona, USA.

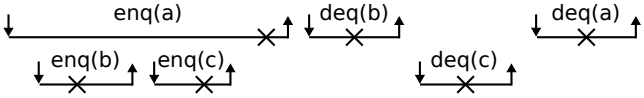




**Figure 1.** A concurrent history of queue operations which is linearizable with respect to FIFO queue semantics [4].

## 1. Introduction

Increasing performance and multicore scalability of concurrent data structures is difficult. A recent trend seems to suggest that fast and scalable concurrent algorithms require reducing the need for synchronization and the overhead that comes with it by weakening data structure semantics [10]. For example, relaxed implementations of FIFO queues may enqueue and dequeue elements in some out-of-order fashion which allows efficient, parallel queue access [3, 5]. However, using FIFO queues as example, we show in a number of experiments on two different machines that relaxed implementations may not only perform and scale better than strict implementations of FIFO queues but can also be seen as actually providing semantics closer to FIFO than strict implementations.

Consider the concurrent history of executing FIFO queue operations illustrated in Figure 1. The horizontal lines represent the execution times of operations, the $\downarrow$ and $\uparrow$ mark the invocation and response times of operations, respectively, and the $\times$ indicates the time when an operation actually takes effect. The operations $enq(a)$ and $deq(a)$, for example, enqueue and dequeue an element $a$, respectively. The operation $enq(a)$ is invoked earlier than the operation $enq(b)$ and significantly earlier than the operation $enq(c)$ but takes effect later then both $enq(b)$ and $enq(c)$. The concurrent history is anyway still linearizable [4] with respect to FIFO queue semantics because the execution of the enqueue operations overlap in time and may therefore be reordered arbitrarily, and the dequeue operations return elements in FIFO order.

From the perspective of the caller of $enq(a)$ the reordering of $enq(a)$ with $enq(b)$ and $enq(c)$ can nevertheless be seen as undesirable. In fact, we argue here that concurrent operations should ideally execute in zero time. A convenient consequence of zero-time operations is that there are no overlapping operations anymore giving rise to what we call the zero-time linearization of a concurrent history where

all operations are assumed to take effect immediately upon invocation. Note that zero-time linearizations of concurrent histories obtained with relaxed but also strict FIFO queue implementations may deviate from FIFO queue semantics.

In order to quantify semantical deviation we measure the degree of reordering of elements in a queue. We say that an element $e$ overtakes an element $e'$ if, in the zero-time linearization, the enqueue operation of the element $e'$ precedes the enqueue operation of the element $e$ and the dequeue operation of the element $e$ precedes the dequeue operation of the element $e'$. For example, element $a$ is overtaken by the elements $b$ and $c$ in Figure 1. We call the number of elements which have overtaken an element $e$ the element-lateness of $e$. In Figure 1 the element-lateness of $a$ is 2 while the element-lateness of $b$ and $c$ is 0. The complementary metric is the element-age of an element $e$, which is the number of elements that are overtaken by $e$. In Figure 1 the element-age of $a$ is 0 while the element-age of $b$ and $c$ is 1. The average element-lateness (or equivalently element-age) of all elements in a run is called element-fairness which is $2/3$ in the example. Note that smaller quantities of element-fairness mean better element-fairness. We show experimentally that some fast but relaxed implementations provide better element-fairness than strict but slow implementations.

Next, we aim at analyzing the factors that influence element-fairness by measuring the degree of reordering of enqueue and dequeue operations when considering the actual time the operations take effect. In particular, we quantify, relative to the zero-time linearization, the degree of operation reordering in what we call the actual-time linearization of a concurrent history where operations are ordered according to the actual time they take effect. Note, however, that the time an operation takes effect may in general only be approximated. We therefore focus in our experiments on benchmarks where actual-time linearizations may be determined exactly.

Similar to element-lateness and element-age we define the operation-lateness and operation-age of an operation $m$ as the number of operations which have overtaken $m$ and which are overtaken by $m$, respectively. An operation $m$ has overtaken an operation $n$ if $m$ is invoked after but takes effect before $n$. Analogous to element-fairness, operation-fairness is then the average operation-lateness (or equivalently operation-age) of all operations in a run. In Figure 1 the operation-lateness of the operation $enq(a)$ is 2, the operation-lateness of all other operations is 0. The operation-age of the operations $enq(b)$ and $enq(c)$ is 1, the operation-age of all other operations is 0. The operation-fairness is therefore $1/3$ in the example.

Figure 2 illustrates the reason why strict but slow implementations may be less operation-fair than fast but relaxed implementations. Slow implementations typically require more attempts to complete an operation under contention than fast implementations. If an attempt fails because

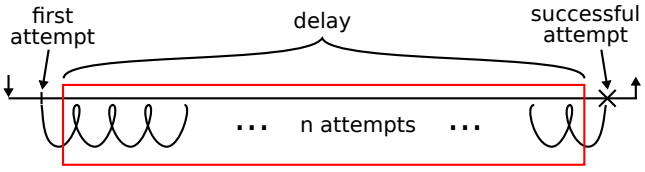



**Figure 2.** An operation that makes $n$ attempts to take effect.

of interference with a concurrent operation a new attempt is started. For example, an attempt to acquire a spin lock consists of reading the lock state and trying to set the lock. A thread then spins on the lock trying to acquire the lock until an attempt finally succeeds. The operation takes effect in the last attempt. Thus, with an increasing number of attempts, the time an operation takes effect deviates more and more from the time it was invoked, resulting in decreased operation-fairness. Fast but relaxed implementations aim at reducing the number of attempts at the alleged expense of data structure semantics, e.g. by allowing elements to be enqueued and dequeued in parallel but out of FIFO order. The result is that the time operations take effect may deviate less from their invocation times, which improves operation-fairness.

We analyze the element- and operation-fairness of strict implementations (a lock-based queue, the Michael-Scott queue [8], and the Flat Combining queue [2]) and relaxed implementations ($k$-FIFO queues [6] and Scal queues [7]). In our experiments poor operation-fairness of strict implementations may outweigh the effect of semantical relaxations. Conversely, some relaxed implementations may provide better element-fairness than strict implementations.

The contributions of this paper are: (1) the notions of zero- and actual-time linearization as well as element- and operation-fairness, and (2) the benchmarks analyzing element- and operation-fairness of various FIFO queue implementations in different contention scenarios.

## 2. Model and Metrics

We first recall the notions of sequential and concurrent history, linearization of a concurrent history, sequential specification, and linearizability [4], and then define zero-time and actual-time linearizations as well as element- and operation-fairness.

Let $O$ be a concurrent object and let $\Sigma$ be a sequential alphabet of operations with data on the concurrent object $O$. For a queue the sequential alphabet is the set of enqueue and dequeue operations $\{enq(e)|e \in A\} \cup \{deq(e)|e \in A \cup \{null\}\}$ where $A$ is the set of elements which can be enqueued into the queue. The concurrent alphabet of operations on the concurrent object $O$ is then the set $\hat{\Sigma} = \{m_i|m \in \Sigma\} \cup \{m_r|m \in \Sigma\}$ of invocations and responses of the operations in $\Sigma$.

A sequential history for a concurrent object $O$ is a sequence of operations $s \in \Sigma^*$ accessing the object $O$. Simi-

larly, a concurrent history is a sequence of invocations and responses of operations $\hat{s} \in \hat{\Sigma}^*$. A concurrent history $\hat{s}$ is complete if for every operation invocation $m_i$ in $\hat{s}$ there also exists the corresponding operation response $m_r$ in $\hat{s}$ and it appears after $m_i$. In this work we only consider complete concurrent histories.

In the following we use the concurrent history

$$enq(a)_i\ enq(b)_i\ enq(b)_r\ enq(c)_i\ enq(c)_r\ enq(a)_r$$
$$deq(b)_i\ deq(b)_r\ deq(c)_i\ deq(c)_r\ deq(a)_i\ deq(a)_r$$

shown in Figure 1 as running example.

Two operations $m,n \in \Sigma$ are said to overlap in a concurrent history $\hat{s}$ if $m_i$ precedes $n_r$ and $n_i$ precedes $m_r$ in $\hat{s}$. In Figure 1 the operation $enq(a)$ overlaps with the operations $enq(b)$ and $enq(c)$.

A sequential history $s \in \Sigma^*$ is called a linearization of a concurrent history $\hat{s} \in \hat{\Sigma}^*$ if

1. For any operation invocation $m_i$ in $\hat{s}$ the operation $m$ appears in $s$, and no operation $m$ appears in $s$ if the operation invocation $m_i$ does not appear in $\hat{s}$.
2. For any operations $m,n \in \Sigma$, if the operation response $m_r$ precedes the operation invocation $n_i$ in $\hat{s}$, then the operation $m$ precedes the operation $n$ in $s$.

For the concurrent history in Figure 1 three linearizations are possible:

$$enq(a)\ enq(b)\ enq(c)\ deq(b)\ deq(c)\ deq(a)$$

$$enq(b)\ enq(a)\ enq(c)\ deq(b)\ deq(c)\ deq(a)$$

$$enq(b)\ enq(c)\ enq(a)\ deq(b)\ deq(c)\ deq(a)$$

REMARK 1. *Note that the linearizations of concurrent histories only differ in the order of overlapping operations. If operations $m,n \in \Sigma$ do not overlap in a concurrent history $\hat{s}$, and the operation response $m_r$ precedes the operation invocation $n_i$ in the concurrent history $\hat{s}$, then the operation $m$ precedes the operation $n$ in any linearization $s$ of $\hat{s}$.*

Next we introduce informally the correctness condition linearizability [4]. A sequential specification of a concurrent object is a prefix-closed set of all valid sequential histories of operations accessing the concurrent object. Based on a sequential specification linearizability defines that an implementation is correct with respect to that sequential specification if for every concurrent history created by the implementation there exists a linearization which is contained in the sequential specification. As all linearizations of a concurrent history differ only in the order of overlapping operations, linearizability forces operations which do not overlap to take place in the order they were invoked, but allows overlapping operations to be executed in any order, see Remark 1.

The sequential specification of a FIFO queue is defined informally as the set of sequential histories where elements are enqueued in the same order as they are dequeued. Additionally a dequeue operation $deq(null)$ only exists if the queue is empty at the time of the $deq(null)$, i.e., if every element which gets enqueued before the $deq(null)$ also gets dequeued before the $deq(null)$. Without loss of generality we assume that elements are unique and get enqueued at most once. Note that only the linearization

$$enq(b)\ enq(c)\ enq(a)\ deq(b)\ deq(c)\ deq(a)$$

of the three previously mentioned linearizations is contained in the sequential specification of a FIFO queue.

### 2.1 Zero-Time Linearization

Intuitively, the zero-time linearization of a concurrent history $\hat{s} \in \hat{\Sigma}^*$ is the linearization of $\hat{s}$ that contains all operations invoked in $\hat{s}$ ordered by their invocation times in $\hat{s}$.

DEFINITION 1. *A linearization $z \in \Sigma^*$ of a concurrent history $\hat{s} \in \hat{\Sigma}^*$ is the zero-time linearization of $\hat{s}$ if an operation $m$ precedes an operation $n$ in $z$ if and only if the invocation $m_i$ precedes the invocation $n_i$ in $\hat{s}$.*

Thus the previously mentioned linearization

$$enq(a)\ enq(b)\ enq(c)\ deq(b)\ deq(c)\ deq(a)$$

is the zero-time linearization of the concurrent history in Figure 1 which is not contained in the sequential specification of a FIFO queue. In fact, all queue implementations that we considered in our experiments, including those which are linearizable with respect to FIFO queue semantics, also create concurrent histories with zero-time linearizations that are not contained in the sequential specification of a FIFO queue.

### 2.2 Element-Fairness

Given a concurrent history $\hat{s} \in \hat{\Sigma}^*$ of a queue and the zero-time linearization $z \in \Sigma^*$ of $\hat{s}$, we define element-lateness, element-age, and element-fairness as follows.

DEFINITION 2. *The element-lateness of an element $e$ with $enq(e)$ and $deq(e)$ in $z$ is the number of elements $e'$ such that $enq(e)$ precedes $enq(e')$ in $z$ and $deq(e')$ precedes $deq(e)$ in $z$.*

Thus the element-lateness of $e$ is the number of elements that overtake $e$ out of FIFO order, i.e., the number of elements that are enqueued after $e$ but dequeued before $e$.

DEFINITION 3. *The element-age of an element $e$ with $enq(e)$ and $deq(e)$ in $z$ is the number of elements $e'$ such that $enq(e')$ precedes $enq(e)$ in $z$ and $deq(e)$ precedes $deq(e')$ in $z$.*

Similarly, the element-age of $e$ is thus the number of elements that $e$ overtakes out of FIFO order, i.e., the number of elements that are enqueued before $e$ but dequeued after $e$.

DEFINITION 4. *The element-fairness of $\hat{s}$ is the average element-lateness (or element-age) of all elements with $enq(e)$ and $deq(e)$ in $z$.*

Note that the average element-lateness of all elements is equal to the average element-age of all elements because every time an element $e$ is overtaken by an element $e'$ the element-lateness of $e$ as well as the element-age of $e'$ increase by one. Moreover, elements that are enqueued but not dequeued are ignored in the three metrics. We have therefore designed our benchmarks such that all elements that are enqueued are also dequeued.

### 2.3 Operation-Fairness

We first introduce the notion of actual-time linearization before defining operation-lateness, operation-age, and operation-fairness. Intuitively, the actual-time linearization of a concurrent history $\hat{s} \in \hat{\Sigma}^*$ is the linearization of $\hat{s}$ that contains all operations invoked in $\hat{s}$ ordered by their linearization points [4], i.e., the times the operations took effect in the run from which $\hat{s}$ was obtained. In our experiments involving actual-time linearizations we focus on benchmarks with strict FIFO queue implementations where elements are enqueued concurrently but dequeued sequentially as in our running example. In this case linearization points can be determined exactly which may otherwise only be possible through approximations.

We informally define that a linearization $x \in \Sigma^*$ of a concurrent history $\hat{s} \in \hat{\Sigma}^*$ is the actual-time linearization of $\hat{s}$ if operation $m$ precedes an operation $n$ in $x$ if and only if the linearization point of $m$ precedes the linearization point of $n$ in the run from which $\hat{s}$ was obtained. With $\times$ indicating linearization points in Figure 1, the previously mentioned linearization

$$enq(b)\ enq(c)\ enq(a)\ deq(b)\ deq(c)\ deq(a)$$

is the actual-time linearization of the concurrent history in our running example.

Given a concurrent history $\hat{s} \in \hat{\Sigma}^*$ of a queue, the zero-time linearization $z \in \Sigma^*$ of $\hat{s}$, and the actual-time linearization $x \in \Sigma^*$ of $\hat{s}$, we define operation-lateness, operation-age, and operation-fairness as follows.

DEFINITION 5. *The operation-lateness of an operation $m$ invoked in $\hat{s}$ is the number of operations $m'$ such that $m$ precedes $m'$ in $z$ and $m'$ precedes $m$ in $x$.*

The operation-lateness of $m$ is therefore the number of operations which overtake $m$ in $\hat{s}$, i.e., the number of operations which get invoked after but take effect before $m$.

DEFINITION 6. *The operation-age of an operation $m$ invoked in $\hat{s}$ is the number of operations $m'$ such that $m'$ precedes $m$ in $z$ and $m$ precedes $m'$ in $x$.*

Complementary to operation-lateness the operation-age of $m$ is the number of operations overtaken by $m$ in $\hat{s}$, i.e., the number of operations which get invoked before but take effect after $m$.

DEFINITION 7. *The operation-fairness of $\hat{s}$ is the average operation-lateness (or operation-age) of all operations invoked in $\hat{s}$.*

Note that although element-fairness and operation-fairness are related they are not equivalent. For example, if an enqueue operation overtakes a dequeue operation, then the operation-age of the enqueue operation increases but the element-age of the enqueued element does not change.

## 3. Experiments

We benchmark a lock-based queue (LB), the Michael-Scott (MS) queue [8], the Flat Combining (FC) queue [2], the unbounded-size (US) $k$-FIFO queue [6, 9], and the round-robin (RR) and 2-random (2-RA) Scal queues [7, 9]. In all implementations we use cache- and page-aligned memory to avoid artifacts in the experiments that are unrelated to the actual queue implementations.

The LB, MS, and FC queues are strict FIFO queue implementations. LB uses a single pthread mutex to synchronize enqueue and dequeue operations. MS uses Compare-and-Swap (CAS) instructions to enqueue and dequeue elements without blocking. If CAS fails because of contention on the queue MS retries until it succeeds. FC uses a lock-based FIFO queue and an array of intended queue operations where each thread has its own slot. A thread accesses the queue by first writing its intended operation into the slot of the array assigned to the thread. Only then the thread tries to acquire the lock of the queue. The thread which actually acquires the lock iterates over the array and executes the intended operations of all threads. The threads which do not get the lock spin over their array slot, check if their operation has already been executed by another thread, and otherwise try again to acquire the lock.

The US $k$-FIFO queue as well as the RR and 2-RA Scal queues are relaxed queue implementations. US $k$-FIFO is based on a queue of segments of size $k$. Elements may be enqueued anywhere in the tail segment. If the tail segment gets full a new segment is appended. Elements are dequeued from the head segment in any order. Empty head segments are removed from the queue of segments. A Scal queue consist not just of one but of $p$ so-called partial queues. To enqueue or dequeue an element a load balancer (here RR or 2-RA) selects one of the partial queues and executes the operation on that particular queue. The implementation of partial queues is based on MS. The RR load balancer uses two global round-robin counters, one for the enqueue operations and one for the dequeue operations. With RR, the first enqueue operation, for example, enqueues into the first partial queue, the second enqueue operation enqueues into the second partial queue, and so on. The 2-RA load balancer first selects two partial queues randomly and then has elements

enqueued in the partial queue which contains less elements and elements dequeued from the partial queue which contains more elements. Selecting two queues randomly provides exponentially better load balancing than selecting just a single queue [1].

To measure element-fairness we use a microbenchmark with a sequentially-alternating access pattern [7]. Each thread executes a loop of ten thousand iterations where in each iteration one element is enqueued and one element is dequeued. Moreover, each thread enqueues two hundred elements before the loop and dequeues two hundred elements after the loop to avoid that the queue ever gets empty during the benchmark. To obtain zero-time linearizations we instrument the benchmark code such that the invocations of queue operations are time-stamped using the globally synchronized Time Stamp Counter of x86 processors. In the rare case when two operations get the same time stamp we consider the operation that is invoked by the thread with the lower thread ID as invoked first.

To measure operation-fairness we need to obtain actual-time linearizations of concurrent histories. We designed a benchmark with a parallel-enqueue-sequential-dequeue access pattern such that actual-time linearizations can be determined for concurrent histories obtained with strict FIFO queue implementations. With this benchmark all threads first enqueue ten thousand elements into a queue in parallel. Afterwards a single thread dequeues all elements from the queue sequentially. In this case, the order in which the dequeued elements appear is the order in which they were originally enqueued which fully determines the actual-time linearization. Note, however, that this way we only obtain non-trivial operation-fairness of enqueue operations. Operation-fairness of dequeue operations can be obtained separately by first enqueueing elements sequentially and then dequeueing them in parallel. Measuring operation-fairness for relaxed queue implementations remains future work.

### 3.1 Experimental Setup

We used two machines for our experiments: an Intel-based server with four 10-core 2GHz Intel Xeon processors (40 cores, 2 hyper-threads per core), 24MB shared L3-cache, and 128GB of memory running Linux 2.6.39, and an AMD-based server with four 6-core 2.1GHz AMD Opteron processors (24 cores), 6MB shared L3-cache, and 48GB of memory running Linux 2.6.32. All queues are implemented in C++ and compiled using gcc 4.3.3 with -O3 optimizations.

The benchmarking threads compute $\pi$ iteratively in between queue operations to simulate different levels of contention on the benchmarked queue. A computational load of one thousand iterations, for example, takes on average 2.3 microseconds on the 40-core machine [9]. The segment size of $k$-FIFO queues and the number of partial queues of Scal queues are set to 80 on the 40-core machine (the number of available hyperthreads) and to 24 on the 24-core machine.

### 3.2 Performance and Scalability

Figures 3(a) and 3(b) show performance and scalability on the 40-core and 24-core machine, respectively, with a computational load of two thousand iterations. The relaxed queue implementations outperform and outscale the strict FIFO queue implementations in this benchmark.

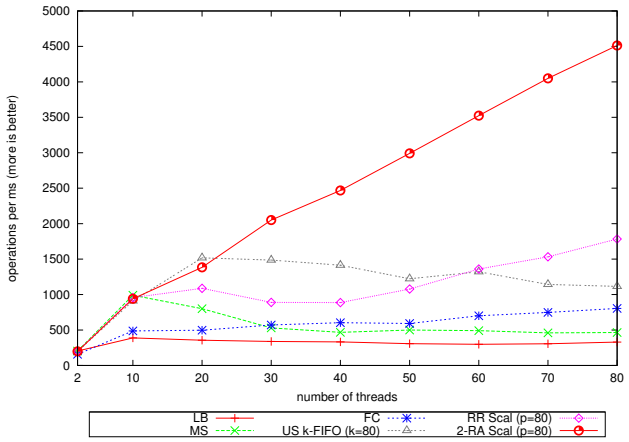
### 3.3 Element-Fairness

Figures 3(c) and 3(d) show element-fairness for the sequentially-alternating benchmark with increasing computational load on the 40-core and 24-core machine with 80 threads and 24 threads, respectively. The 2-RA Scal queue shows the worst element-fairness on both machines. On the 40-core machine the FC queue has the best element-fairness with high contention on the queue. With a computational load of 16000 iterations or higher the element-fairness of the RR Scal queue becomes better than the element-fairness of the FC queue. On the 24-core machine the element-fairness of the RR Scal queue is better than the element-fairness of the FC queue except with a computational load of 64000 iterations. This means that relaxed queue implementations may outperform strict FIFO queue implementations even in terms of element-fairness. However, the element-fairness of the US $k$-FIFO queue is significantly worse than the element-fairness of the FC queue yet still close to the element-fairness of the LB queue and the MS queue with high contention on the queue. Depending on the computational load the US $k$-FIFO queue can even have better element-fairness than the MS queue.

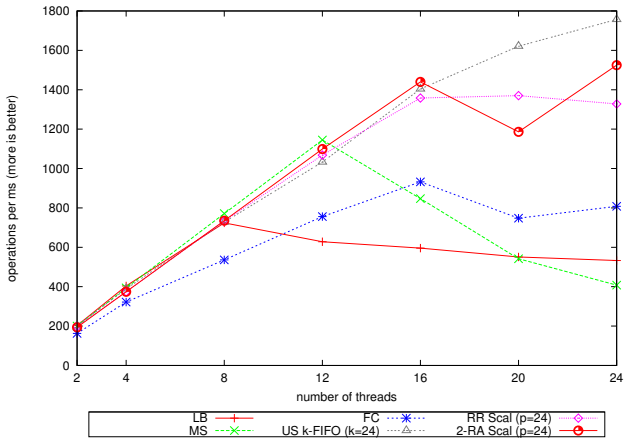
### 3.4 Operation-Fairness

We discuss our results in terms of operation-lateness and operation-age rather than operation-fairness for a more detailed analysis. Figures 4(a) and 4(b) show the maximum operation-lateness and operation-age of all enqueue operations, respectively, for the parallel-enqueue-sequential-dequeue benchmark on the 40-core machine. The maximum operation-lateness of the LB and MS queues is significantly higher than their maximum operation-age. The maximum operation-lateness of the FC queue, however, is even lower ($< 65$) then its maximum operation-age, except for the outlier with a computational load of 8000 iterations. Note that the maximum operation-age of the FC queue is close to eighty, which is the number of threads in the experiment.

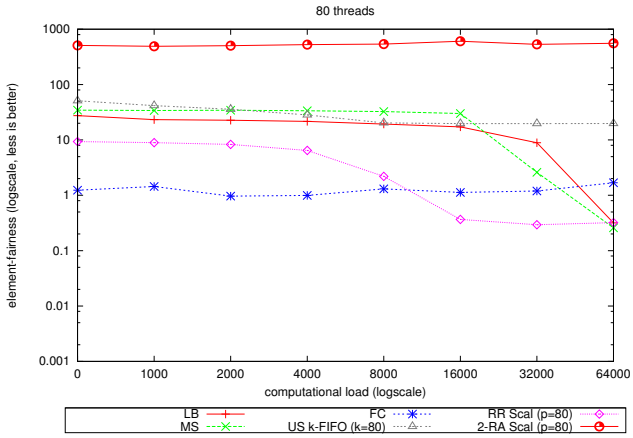
Figures 4(c) and 4(d) show the percentage of enqueue operations with an operation-lateness and operation-age greater than zero, respectively. For the LB and MS queues nearly all enqueue operations have an operation-age greater than zero, i.e., nearly all enqueue operations overtake at least one other enqueue operation. For the MS queue also the number of enqueue operations which are overtaken by at least one other enqueue operation is high. Only one out of five enqueue operations is not overtaken by another enqueue operation. For the LB queue only every third enqueue oper-



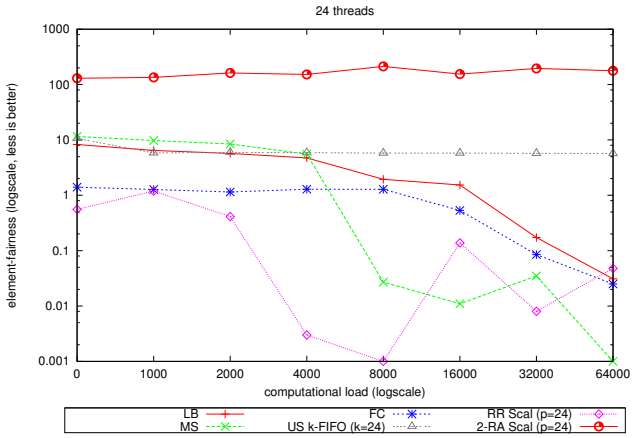

(a) Performance and scalability of the sequentially-alternating benchmark with an increasing number of threads and a computational load of two thousand iterations on the 40-core machine.




(b) Performance and scalability of the sequentially-alternating benchmark with an increasing number of threads and a computational load of two thousand iterations on the 24-core machine.




(c) Element-fairness for the sequentially-alternating benchmark with increasing computational load and 80 threads on the 40-core machine.




(d) Element-fairness for the sequentially-alternating benchmark with increasing computational load and 24 threads on the 24-core machine.

**Figure 3.** Performance, scalability, and element-fairness of the sequentially-alternating benchmark on the 40-core and 24-core machines.

ation is overtaken. For the FC queue the number of enqueue operations which overtake other enqueue operations is low, only one out of five enqueue operations overtakes another enqueue operation under high contention. However, more than every second enqueue operation is overtaken.

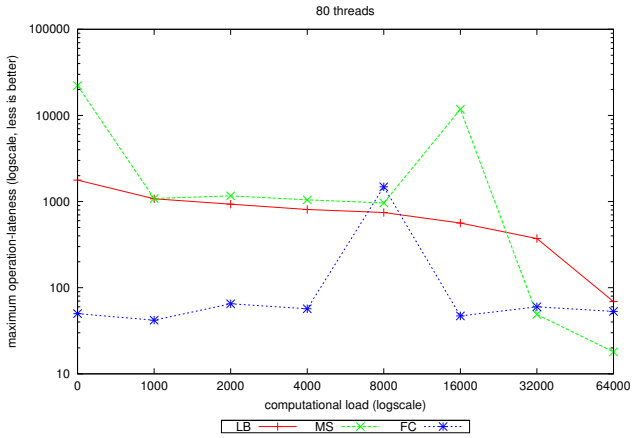
## 4. Conclusions

We have introduced two metrics: (1) element-fairness for quantifying the deviation of the supposedly ideal zero-time linearization of a concurrent history of queue operations from FIFO queue semantics, and (2) operation-fairness for quantifying the difference in the degree of operation reordering between the zero-time and actual-time linearizations of a concurrent history. Operation-fairness helps explaining the application-relevant notion of element-fairness. We have evaluated and compared the performance, scalability, element-fairness, and operation-fairness of several strict and relaxed implementations of concurrent FIFO queues. The experiments show that some relaxed implementations may not only perform and scale better than strict implementations but can also be seen as actually providing semantics closer to FIFO than strict implementations.

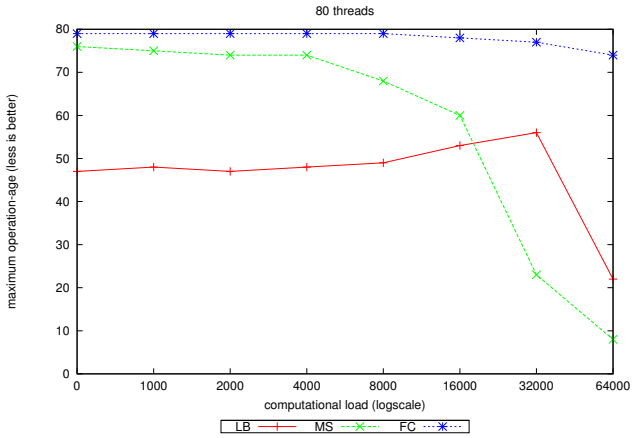
Interesting directions for future work are the problem of measuring operation-fairness even for relaxed queue implementations and studying element-fairness and operation-fairness in the context of other concurrent data structures such as stacks and priority queues.

## Acknowledgments

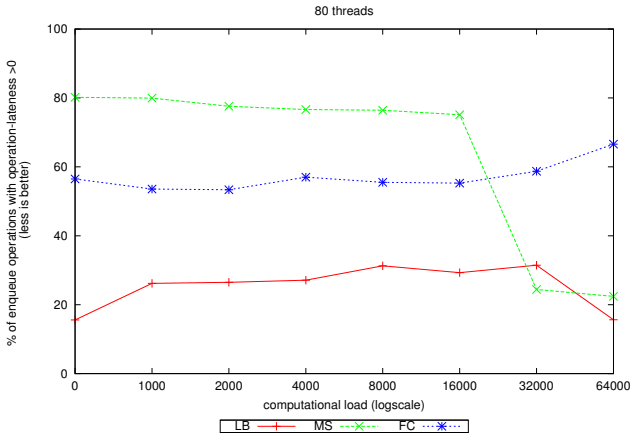
We thank Ana Sokolova for her helpful feedback especially on the theoretical parts of the paper.



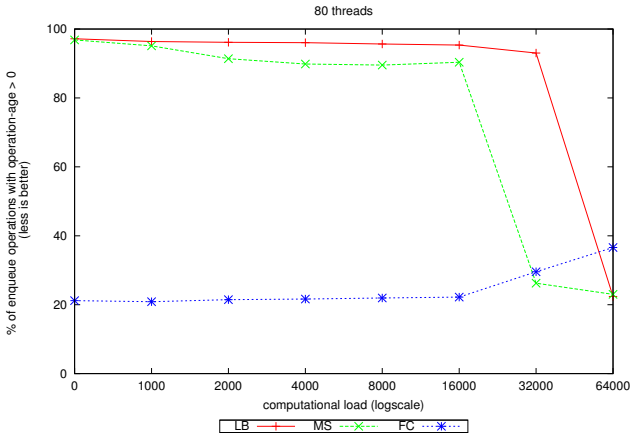

(a) Maximum operation-lateness of all enqueue operations.




(b) Maximum operation-age of all enqueue operations.




(c) Percentage of enqueue operations with an operation-lateness $> 0$.




(d) Percentage of enqueue operations with an operation-age $> 0$.

**Figure 4.** Operation-lateness and operation-age for the parallel-enqueue-sequential-dequeue benchmark with increasing computational load and 80 threads on the 40-core machine.

## References


[1] P. Berenbrink, A. Czumaj, A. Steger, and B. Vöcking. Balanced allocations: The heavily loaded case. *SIAM Journal on Computing*, 35(6):1350–1385, 2006.

[2] D. Hendler, 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.

[3] T. Henzinger, C. Kirsch, H. Payer, A. Sezgin, and A. Sokolova. Quantitative relaxation of concurrent data structures. In *Proc. Symposium on Principles of Programming Languages (POPL)*. ACM, 2013.

[4] 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.

[5] 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.

[6] C. Kirsch, M. Lippautz, and H. Payer. Fast and scalable k-fifo queues. Technical Report 2012-04, Department of Computer Sciences, University of Salzburg, June 2012.

[7] C. Kirsch, H. Payer, H. Röck, and A. Sokolova. Performance, scalability, and semantics of concurrent FIFO queues. In *Proc. International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP)*, LNCS. Springer, 2012.

[8] 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.

[9] H. Payer. *Multicore Scalability of Concurrent Objects*. Phd thesis, University of Salzburg, Salzburg, Austria, 2012.

[10] N. Shavit. Data structures in the multicore age. *Communications ACM*, 54:76–84, March 2011.