Balanced Allocations over Efficient Queues: A Fast Relaxed FIFO Queue

Kåre von Geijer
Department of Computer Science and Engineering
Chalmers University of Technology and
University of Gothenburg
Gothenburg, Sweden
karev@chalmers.se

Philippas Tsigas
Department of Computer Science and Engineering
Chalmers University of Technology and
University of Gothenburg
Gothenburg, Sweden
tsigas@chalmers.se

Elias Johansson
Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden

Sebastian Hermansson
Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden

Abstract

Relaxed semantics have been introduced to increase the achievable parallelism of concurrent data structures in exchange for weakening their ordering semantics. In this paper, we revisit the balanced allocations d-choice load balancing scheme in the context of relaxed FIFO queues. Our novel load balancing approach distributes operations evenly across n sub-queues based on operation counts, achieving low relaxation errors independent on the queues size, as opposed to similar earlier designs. We prove its relaxation errors to be of $O(\frac{n \log \log n}{\log d})$ with high probability for a collection of possible executions. Furthermore, our scheme, contrary to previous ones, manages to interface and integrate the most performant linearizable queue designs from the literature as components. Our resulting relaxed FIFO queue is experimentally shown to outperform the previously best design using balanced allocations by more than four times in throughput, while simultaneously incurring less than a thousandth of its relaxation errors. In a concurrent breadth-first-search benchmark, our queue consistently outperforms both relaxed and strict state-of-the-art FIFO queues.

CCS Concepts: • **Computing methodologies** → **Parallel algorithms**; **Shared memory algorithms**; • **Mathematics of computing** → **Probabilistic algorithms**; • **Theory of computation** → *Data structures design and analysis*; *Program semantics.*

This work is licensed under Creative Commons Attribution International 4.0.
PPoPP '25, March 1–5, 2025, Las Vegas, NV, USA
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1443-6/25/03
https://doi.org/10.1145/3710848.3710892

Keywords: concurrent data structures, lock-free, relaxed semantics, load balancing, balls-into-bins

ACM Reference Format:
Kåre von Geijer, Philippas Tsigas, Elias Johansson, and Sebastian Hermansson. 2025. Balanced Allocations over Efficient Queues: A Fast Relaxed FIFO Queue . In *The 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP '25), March 1–5, 2025, Las Vegas, NV, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3710848.3710892

1 Introduction

The available hardware parallelism has seen tremendous growth the last two decades, which in turn has heightened the need for efficient concurrent data structures [34]. The design of efficient concurrent queues has proven to be a hot area, and there are now many efficient FIFO queues [26, 42], stacks [8, 17] and priority queues [5, 22]. However, the scalability of such sequential designs is limited, as all operations must form a total order and often contend for the same few memory locations [9].

A popular approach to further increase scalability of such inherently higly contented data structures has been to relax data structure semantics [34]. The most common type of relaxation is the *out-of-order* one, where a set of operations is allowed to deviate from the sequential linearization order. Relaxed queues often relax the ordering of their dequeue operations and use the terms *rank error* and *delay* to quantify the relaxation errors. Consider the dequeue of item x at time t in a FIFO queue. Then the *rank error* becomes the number of items older than x in the queue at t, and the *delay* the number of items enqueued after the enqueue of x that were dequeued before t.

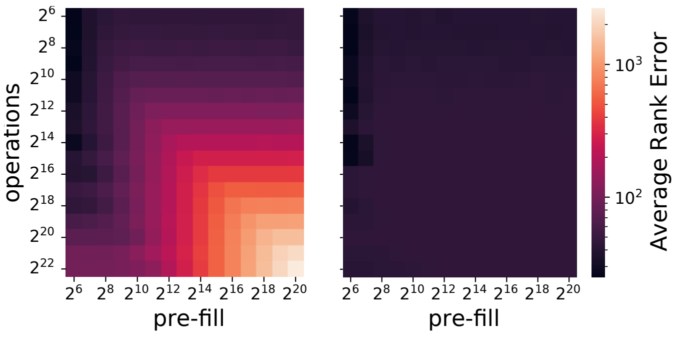
Queues are commonly relaxed by being split up into disjoint sub-queues [15, 32, 33]. Here, operations linearize by operating on one of these sub-queues, and the main algorithmic difference between the queues is what load balancing scheme they use. A large part of the research is focused on load balancing that guarantees deterministic worst-case rank error and delay bounds [18, 20, 33], which leads to designs that are often quite easy to work with and extend [40].On the other hand, several other designs leverage the power of randomization, often enabling them to achieve better tradeoffs between efficiency and observed relaxation [15, 29, 41], at the cost of losing their deterministic worst-case relaxation bounds.

A prevalent technique in randomized relaxation [15, 41] has been the use of the d-choice load balancer, first analyzed by Azar et al. [3]. Consider trying to balance the load of m balls over a set of n bins. If one inserts each new ball into the least loaded bin out of a random choice of d (where $d \geq 2$), it is proven that the heaviest bin will diverge at most $O(\log\log n/\log d)$ from the average load *with high probability* (w.h.p.) when $m = n$ [3] or $m \gg n$ [4].This has been used to relax queues by splitting them up into n sub-queues and operating on the preferred sub-queue out of the d randomly selected ones for each operation [20, 32].

Applying this d-choice load balancer to queues requires a preference between sampled sub-queues. The MultiQueue [29, 32, 41] relaxed priority queue has proven successful by choosing the sub-queue with the highest priority item for deletes. For relaxed FIFO queues, the d-RA queue [15, 20] demonstrated efficiency by enqueuing into the shortest sampled sub-queue and dequeueing from the longest. However, basing the d-choice on the length of the sub-queues comes at the expense of making the rank error dependent on the total queue size. This is demonstrated in Figure 1a, which visualizes the average rank error of the 2-RA with 64 sub-queues over 100 runs in a sequential setting. It shows empirically that the rank errors in the d-RA can degrade with increasing queue size. The prior analysis on the d-RA only bounds the difference in sub-queue size as a function of the number of sub-queues, missing the connection to the rank errors, and their experiments seem to only run on small queues where the relaxation looks good.

Another issue with contemporary relaxed FIFO queues is that essentially all of them [15, 20, 33] build upon suboptimal sub-queues such as the Michael-Scott (MS) queue [24]. The main reason for this is that the MS queue provides linearizable exact enqueue and dequeue counts, a crucial element of the respective relaxed designs. More scalable FIFO queue designs, such as the LCRQ [26] lack this property, making them more intricate to utilize. By using modern sub-queues based on arrays and fetch-and-add (FAA) instead of compare-and-swap (CAS), one can get better performance at similar relaxation errors.

In this paper, we introduce a d-choice sub-queue load balancer that gives both good throughput and low relaxation. For enqueues (dequeues) it samples d sub-queues at random and operates on the one with fewest total enqueues (dequeues) performed on it. By using results from balanced allocations



(a) Length-based **(b)** Operation-based

Figure 1. FIFO rank errors when using 2-choice load balancing over length and operations respectively, using 64 sub-queues.

for heavily loaded balls-into-bins [4] we bound the rank errors and delay of our load balancing scheme to $O(\frac{n\log\log n}{\log d})$ w.h.p. in execution with non-overlapping operations. Figure 1b demonstrates the difference in rank errors when balancing using sub-queue lengths and operation counts. It shows that the earlier length-based scheme, as used in the d-RA queue, has errors increasing with the size of the queue, while our operation-based balancing maintains low rank errors invariantly of the `pre-fill` and number of `operations`.

Furthermore, we show how to incorporate fast queue designs, such as the LCRQ by Morrison and Afek [26] and the wait-free queue by Yang and Mellor-Crummey [42], as sub-queues for such d-choice load balanced queues. We describe the interface a sub-queue needs to implement, which mainly entails noticing if the queue has changed between two points in time, for ensuring empty-linearizability and lock-freedom. We then describe how to implement this for a set of such fast queues by using internal counters that are incremented and used every operation. We also show how any queue can be used as sub-queue when the requirement for empty-linearizability is removed, as it is not standard in relaxed queues [33, 41].

Contributions. We introduce the randomly relaxed FIFO d-Choice Balanced Operations (d-CBO) queue that builds upon the theory of balanced allocations. The operation-based load balancer is shown analytically to give rank errors and delay of $O(\frac{n\log\log n}{\log d})$ w.h.p. in execution with non-overlapping operations. Our experimental evaluation finds that it gives orders of magnitude smaller errors in concurrent executions than previous designs, as well as suggesting that general concurrent executions scale as when assumeing non-overlapping operations. The queue further improves on earlier designs by incorporating state-of-the-art sub-queues for increased performance. The sub-queue interface is generic and leaves room for future queue designs, and our experimental evaluation shows that the fast sub-queues can give more than 10 times better throughput when using a low number of sub-queues. Our breadth-first-search (BFS) macro-benchmark demonstrates consistently and significantly faster running times than earlier strict and relaxed concurrent FIFO queues.

Outline. We present the novel algorithm in Section 3. Based on this algorithm description, we analyze the expected errors from its load balancing scheme in Section 4, where we also prove its linearizability and lock-freedom. We describe how to integrate our algorithm with state-of-the-art sub-queues in Section 5. Our experimental evaluation in Section 6 evaluates throughput and rank errors in micro-benchmarks, as well as the possible speedup of relaxed queues in a concurrent BFS benchmark. Finally, the paper concludes in Section 7.

2 Related Work

The Michael-Scott (MS) queue [24] was introduced in 1996 and has since become a foundational design in lock-free data structures due to its simplicity and extensibility, as evidenced by its adaptation in the Harris linked list [16] and many relaxed queues [15, 20, 33]. However, the design faces scalability issues due to two highly contended CAS retry loops at the head and tail, respectively. In the following years, there were several lock-free queue designs [12, 19, 25, 37] that improved on the MS queue in different scenarios.

Morrison and Afek proposed the LCRQ in 2013 [26] which since has remained among the most scalable and flexible lock-free queue designs. In utilizes a linked list of circular ring buffers with epoch counters inside cells for linearizability. The key is that they use FAA on head and tail counters in each buffer to assign operations to cells. This leads to dequeues occasionally reaching the cell before the corresponding enqueue, poisoning the cell and forcing the future enqueue to retry. However, the performance increase from using FAA over CAS far dwarfs these retry costs in most cases.

There has also been work on wait-free queues, but until recently, the fastest wait-free queue was orders of magnitude slower than that of lock-free queues [10]. However, Kogan and Petrank [21] presented the *fast-path-slow-path* methodology in 2012, that was later used by Yang and Mellor-Crumney in 2016 to create an efficient wait-free queue [42] (WFQ). Similarly to the LCRQ, the WFQ is a linked list of bounded buffers, where FAA is used to assign operations to cells. The difference is that after a certain set of failed attemps at linearization, a slow operation gets help from others until it succeeds. However, in the fast path, which is most cases, it performs very similarly to the LCRQ. A similar design to the WFQ, but without the wait-free helping, called the FAAArrayQueue [31] was described by Ramalhete in 2016.

Another area with a lot of interest over the years is balanced allocations. In the balls-into-bins problem, we are given m balls to insert into n bins with the goal of minimizing the maximal bin load, which is the same as minimizing the $gap = \max_{i=1...n}\{bin_i - \frac{m}{n}\}$. The problem assumes that one cannot always inspect all bins and is divided into the *lightly loaded* case where $n = m$ and the *heavy loaded* case where $m \gg n$. In the simplest *one-choice* process where balls are allocated uniformly at random, the gap w.h.p. is $O(\frac{\log n}{\log\log n})$ [14] in the lightly loaded case, and w.h.p. $O(\sqrt{\frac{m}{n}\log n})$ [30] in the heavily loaded case.

Simply extending the strategy to a *two-choice* process, where each ball is allocated to the least loaded of two random bins, proves a huge improvement over the one-choice process. This was first analyzed by Azar et al. [3] who showed that the process w.h.p. only has a gap of $\log_2\log n + O(1)$ in the lightly loaded case. Berenbrink et al. [4] extended the analysis to d-choice processes (allocate each ball to the lightest of d sampled bins) where $d \geq 2$ in the heavily loaded case, for which they found the same bound as in the light setting $\log_d \log n + O(1)$. This heavily loaded bound was later also proved by Talwar and Wieder [36], but using a simpler proof technique that was generalizable for the case of weighted balls.

The two-choice process has proven useful within the design of relaxed priority queues. Perhaps the most successful design yet is the MultiQueue [32, 41] which is a relaxed priority queue which inserts items into random partial priority queues, and removes items from the the partial with the highest priority item out of a random choice of two. It has proven to be effective in practice [41], for example as a priority scheduler [29]. Its relaxation has also been analyzed using techniques from balls-into-bins [1, 2, 29].

Furthermore, the d-choice has been applied to relaxed FIFO queues with the d-RA queue [20], where items are enqueued into the shortest (and dequeued from the longest) of d sampled queues. Although showing comparatively good performance to other designs, the relaxation error of the queue is not properly analyzed and it is based on the sub-optimal MS queue [24] for sub-queues, depending on the exact enqueue and dequeue counters this provides.

There are also relaxed FIFO queues that guarantee worst-case bounds on the rank error. The k-segment queue [20] is designed as a MS queue [24] where every node is an array with space for k items. The tail (head) is only advanced when completely filled (emptied), which gives a maximal rank error of $k-1$. Similarly, the 2D queue [33] superimposes a window with a *width* and *depth* over its partial queues at its head and tail, that define operable areas for the dequeues and enqueues respectively. This leads to a rank error bound of $(width - 1) \cdot depth$. Although the 2D queue improves on the k-segment queue, they both share the issue of threads searching for valid sub-queues the final operations before the segment/window advances.

The b-round-robin queue [15] is another relaxed FIFO queue. In it, every thread is attached to one of b counters for enqueues and dequeues respectively, and for every operation is uses fetch-and-add on the counter, modulo the number of partial queues, to get which queue to operate on. Letting the relaxation of a queue become unbounded, but requiring empty-linearizability, corresponds to concurrent pools. Sundell et al. [35] presented an efficient pool for workloads where each thread both insert and removes items, and Gidron et al. [13] presented an efficient design for producer-consumer workloads.

The potential of relaxing the sequential semantics of concurrent data structures was first highlighted by Shavit [34] and first formalized by the concept of *quantitative relaxation* by Henzinger et al. [18]. However, their definition only covers designs with worst-case bounds. Alistarh et al. [1] defined *distributional linearizability* as a correctness condition for concurrent designs with randomized relaxation, such as the MultiQueue [32]. The concept of *elastic relaxation* has also been recently introduced by von Geijer et al. [40] as a correctness condition for designs in which the relaxation bounds can be changed during run-time to match dynamic executions.

3 Algorithmic Description

The pseudocode of our d-Choice Balanced Operations (d-CBO) queue is shown in Algorithm 1. It starts by presenting the generic functions required for the sub-queue to implement (line 1.1). As for all FIFO queues, the sub-queues must implement *Enqueue* and *Dequeue*. Furhermore, the sub-queue must implement *EnqCount* and *DeqCount* which should return how many items have been enqueued and dequeued from the sub-queue. Finally, *EnqVersion* returns an integer that is unique for each current tail item.

For linearizability, the sub-queues must be linearizable, and the *EnqVersion* must change during the linearization of each enqueue, never returning the same count for two different tail items. The *EnqCount* and *DeqCount* should be exact during sequential executions, but are allowed to deviate in concurrent executions as long as the deviation is on the same order of magnitude for all sub-queues. To guarantee lock-freedom, the sub-queue must be lock-free, and *EnqVersion* is only allowed to change a finite number of times before an enqueue linearizes. Most lock-free queues can implement this interface with small additions, and we show how to implement it for a few modern sub-queues in Section 5.

Moving on to the d-CBO methods, its *Enqueue* (line 1.7) samples d sub-queues at random (with replacement), selects the optimal sampled sub-queue as the one with lowest *EnqCount*, and linearizes by enqueueing into it. As outlined

Algorithm 1: d-Choice Balanced Operations Queue

```
1.1  generic SubQueue
1.2      method SubQueue.Enqueue(item)
1.3      method SubQueue.Dequeue() → Item | NULL
1.4      method SubQueue.EnqCount() → uint
1.5      method SubQueue.DeqCount() → uint
1.6      method SubQueue.EnqVersion() → uint
1.7  function Enqueue(sub_queues, d, item)
1.8      samples ← randomly sample $(q_1, \dots, q_d)$ where
           $q_i \in$ sub_queues
1.9      optimal ← $\operatorname{argmin}_{q \in samples}$ EnqCount(q)
1.10     optimal.Enqueue(item)
1.11 function Dequeue(sub_queues, d)
1.12     samples ← randomly sample $(q_1, \dots, q_d)$ where
           $q_i \in$ sub_queues
1.13     optimal ← $\operatorname{argmin}_{q \in samples}$ DeqCount(q)
1.14     dequeued ← optimal.Dequeue()
1.15     if dequeued ≠ NULL then
1.16         return dequeued
1.17     else
1.18         return DoubleCollect(sub_queues)
1.19 function DoubleCollect(sub_queues)
1.20     versions ← [0, . . . , 0]
1.21     repeat
1.22         for $q_i \in$ sub_queues do
1.23             versions[i] ← EnqVersion($q_i$)
1.24             dequeued ← $q_i$.Dequeue()
1.25             if dequeued ≠ NULL then
1.26                 return dequeued
1.27         if $\forall q_i \in$ sub_queues, EnqVersion($q_i$) =
               versions[i] then
1.28             return NULL
```

above, this will also change the *EnqVersion* and increment the *EnqCount* of the sub-queue.

The d-CBO *Dequeue* (line 1.11) similarly tries to dequeue from the sub-queue with the lowest *DeqCount* out of the d sampled ones. However, if the sub-queue *Dequeue* returns `NULL`, the d-CBO *Dequeue* cannot immediately return `NULL`. Instead it must enter the *DoubleCollect* function to guarantee empty-linearizability.

The *DoubleCollect* function (line 1.18) uses the double-collect [28] mechanism to try to get an atomic snapshot where all sub-queues are empty. It does this by first iterating over the sub-queues (line 1.22), recording their *EnqVersion* and trying to dequeue an item from them. To avoid sub-queue bias, this iteration should start at a random index. If it at any point succeeds in dequeuing an item, it successfully returns it. However, if it found all sub-queues empty in the first iteration, it then does a second iteration (line 1.27) where it checks if any *EnqVersion* has changed, signalling that an item might have been enqueued, in which case it restarts. If no *EnqVersion* has changed, no item can have been enqueued between the end of the first iteration and the start of the second, at which point all sub-queues were empty and it is safe to return *NULL*.

4 Analysis

4.1 Probabilistic Relaxation Guarantees

This section analyzes the relaxation errors in the d-CBO queue. We consider the set of exections where enqueues and dequeues are non-overlapping or equivalently occur atomically, a common approach used in the related literature [2, 20, 41], and analyze the load balancing scheme in this setting.

On a high level, our analysis starts by formalizing the sub-queue selection as an enqueue and dequeue multivariate process over the *EnqCount* and *DeqCount*, similarly to the balls-into-bins process. The rank error and delay are then formulated in terms of those two processes. Without empty sub-queues, the enqueue and dequeue processes would be equivalent to the Greedy d-choice [3], and we could re-use its theory [4]. This is however not the case, but Lemma 4.4 and Lemma 4.5 use the short memory of the d-choice process to show that this difference does not affect the rank error bound, as the deviation only takes place when the queue is small. These two lemmas naturally prove Theorem 4.6 and Theorem 4.7, which bound the rank error and delay to $O(\frac{n \log\log n}{\log d})$ w.h.p.

Let $\mathcal{E}_i(t)$ denote the *EnqCount* of sub-queue i at time t. The enqueue operation, as described on line 1.7, inserts its item into the sub-queue with the smallest EnqCount among the d sampled queues, in the process incrementing that *EnqCount* by 1. In terms of $\mathcal{E}$, an enqueue samples d $\mathcal{E}_i(t)$ and increments the smallest of them.

Similarly, let $\mathcal{D}_i(t)$ denote the *DeqCount* of sub-queue i at time t. A dequeue operation as described on line 1.11 tries to dequeue an item from the sub-queue with the smallest *DeqCount* out of the k samples sub-queues, incrementing its *DeqCount*. However, if the sampled sub-queueue with smallest *DeqCount* is empty, the *DoubleCollect* mechanism at line 1.18 is used to dequeue an item from another queue or return *NULL* if the whole queue is empty. This can be seen as sampling d counts in $\mathcal{D}_i(t)$ and incrementing the smallest such $\mathcal{D}_j(t)$ if $\mathcal{D}_j(t) < \mathcal{E}_j(t)$. But if $\mathcal{D}_j(t) \geq \mathcal{E}_j(t)$, the next (in round-robin order) $\mathcal{D}_i(t)$ where $\mathcal{D}_i(t) < \mathcal{E}_i(t)$ is incremented. If no such $\mathcal{D}_i(t)$ exists, the dequeue has no effect and returns *NULL*.

Definition 4.1. The *gap* of a n-dimensional vector $\mathcal{X}$, such as $\mathcal{E}$ or $\mathcal{D}$, is defined as the difference between its largest value and its mean.

$$gap(\mathcal{X}) = max(\mathcal{X}) - \overline{\mathcal{X}}$$

Definition 4.2. The *rank error* of the dequeue of item x, which was enqueued at time t_e and dequeued at t_d, is the number of other items that were enqueued before t_e and are still in the queue at time t_d. In terms of $\mathcal{D}$ and $\mathcal{E}$, this can be expressed as

$$\begin{aligned} rank\ error &= \sum_{i=1}^{n} max(0, \mathcal{E}_i(t_e) - \mathcal{D}_i(t_d)) \\ &\leq \sum_{i=1}^{n} |\mathcal{E}_i(t_e) - \mathcal{D}_i(t_d)|. \end{aligned} \tag{1}$$

Definition 4.3. The *delay* of item x, which was enqueued at time t_e and dequeued at time t_d (where $t_d = \infty$ if x is never dequeued), is defined as the number of items enqueued after t_e but dequeued before t_d. This is formulized in terms of $\mathcal{E}$ and $\mathcal{D}$ as

$$\begin{aligned} delay &= \sum_{i=1}^{n} max(0, \mathcal{D}_i(t_d) - \mathcal{E}_i(t_e)) \\ &\leq \sum_{i=1}^{n} |\mathcal{E}_i(t_e) - \mathcal{D}_i(t_d)|. \end{aligned} \tag{2}$$

Equations 1 and 2 show the duality of the rank error and delay, where they simply swap the sign inside the *max* at each sub-queue. Bounding this sum $\sum_{i=1}^{n} |\mathcal{E}_i(t_e) - \mathcal{D}_i(t_d)|$ now becomes the key to bounding both the rank error and delay, and is bounded in the following two lemmas.

Lemma 4.4. *At any time t, it holds that $gap(\mathcal{E}(t)) = O(\frac{\log\log n}{\log d})$ and $gap(\mathcal{D}(t)) = O(\frac{\log\log n}{\log d})$.*

Proof. As the updates of $\mathcal{E}$ matches the d-choice balanced allocation (Greedy[d]) process for balls-into-bins, the analysis of Berenbrink et al. [4] gives, $\forall c > 0$, $\exists \gamma(c)$ such that $Pr[gap(\mathcal{E}(t)) \geq \frac{\log\log n}{\log d} + \gamma] \leq n^{-c}$, for any t. Therefore, the lemma holds for $\mathcal{E}$.

As earlier outlined, the use of double-collect in dequeues makes $\mathcal{D}$ deviate from the classical d-choice process. Instead of incrementing the lowest sampled $\mathcal{D}_i(t)$ when $\mathcal{D}_i(t) = \mathcal{E}_i(t)$, it instead increments another $\mathcal{D}_j(t)$ where $\mathcal{D}_j(t) < \mathcal{E}_j(t)$. However, as $\mathcal{D}_i(t) = \mathcal{E}_i(t)$, we have $\overline{\mathcal{E}(t)} - \overline{\mathcal{D}(t)} = O(\frac{\log\log n}{\log d})$ w.h.p. Furthermore, as the double-collect enforces that the incremented count $\mathcal{D}_j(t) < \mathcal{E}_j(t)$, and $\overline{\mathcal{E}(t)} - \overline{\mathcal{D}(t)} = O(\frac{\log\log n}{\log d})$, a series of such double-collect increments can only increase $gap(\mathcal{D}(t))$ by $gap(\mathcal{E}(t)) = O(\frac{\log\log n}{\log d})$.

So, assume $gap(\mathcal{D}) = O(\frac{\log\log n}{\log d})$ at time t. Then at time $t + T$ the effects of the double-collect will not have an effect on the order of the gap, and the short-term memory of the Greedy[d] process, as proven by Berenbrink et al. [4], guarantee that the successful d-choice dequeues maintain

$gap(\mathcal{D}(t)) = O(\frac{\log\log n}{\log d})$ even if the distributions of the counts in $\mathcal{D}(t)$ take on a different pattern than the normal d-choice process. ◻

Lemma 4.5. *For any item enqueued at time t_e and dequeued at time t_d, it holds that $\sum_{i=1}^{n} |\mathcal{E}_i(t_e) - \mathcal{D}_i(t_d)| = O(\frac{n\log\log n}{\log d})$.*

Proof. Assume the item was dequeued from queue i. Then $\mathcal{D}_i(t_d) = \mathcal{E}_i(t_e)$ due to the total order in the sub-queue. Thus, Lemma 4.4 implies that $|\overline{\mathcal{E}(t_e)} - \overline{\mathcal{D}(t_d)}| = O(\frac{\log\log n}{\log d})$ w.h.p., which when summed over all sub-queues w.h.p. gives $\sum_{i=1}^{n} |\mathcal{E}_i(t_e) - \mathcal{D}_i(t_d)| = O(\frac{n\log\log n}{\log d})$. ◻

With Lemmas 4.4 and 4.5 we now have all we need to create simple proofs bounding the rank error and the delay of the d-CBO queue.

Theorem 4.6. *A d-CBO queue w.h.p. has rank errors of $O(\frac{n\log\log n}{\log d})$, in executions with non-overlapping operations.*

Proof. The rank error of a dequeue is in equation 1 bounded by $\sum_{i=1}^{n} |\mathcal{E}_i(t_e) - \mathcal{D}_i(t_d)|$, where t_e and t_d are the timestamps when the item was enqueued and dequeued respectively. That sum in turn is in turn found to be $O(\frac{n\log\log n}{\log d})$ in Lemma 4.5. ◻

Theorem 4.7. *Items in a d-CBO queue w.h.p. have delays of $O(\frac{n\log\log n}{\log d})$, in executions with non-overlapping operations.*

Proof. The delay of an item is in equation 2 bounded by $\sum_{i=1}^{n} |\mathcal{E}_i(t_e) - \mathcal{D}_i(t_d)|$, where t_e is the time the item was enqueued and t_d the time it was dequeued (or ∞ if never dequeued). Lemma 4.5 in turn finds that sum to be of order $O(\frac{n\log\log n}{\log d})$. ◻

4.2 Correctness Guarantees

In this part we prove the linearizability and lock-freedom of the d-CBO queue. The analysis utilizes the sub-queue interface, which guarantees that they should be linearizable, lock-free, that *EnqVersion* must be uniquely changed for every tail, and that at least one enqueue must linearize after the *EnqVersion* has changed a certain finite number of times.

Theorem 4.8. *The d-CBO queue is linearizable with respect to the semantics of a concurrent pool.*

Proof. An enqueue operation always linearizes with a linearizable enqueue to a sub-queue at line 1.10. Similarly, a non-empty dequeue also linearizes on a sub-queue at either line 1.16 or line 1.26.

Finally, dequeues returning *NULL* linearize with the double-collect [28] mechanism by repeating two itertions over the sub-queues in the *DoubleCollect* function (line 1.18). The first (line 1.22) collects *EnqVersion* counts for every sub-queue, after which it ensures that the queue is empty (as it otherwise returns an dequeued item at line 1.26). The *DoubleCollect* call only returns *NULL* if the sub-queues have unchanged *EnqVersion* counts during the second iteration (line 1.27). As the version counts are guaranteed to be incremented during the linearization of an enqueue, this means that no enqueue has linearized between the end of the first iteration and the start of the second. Finally, as all queues were found empty in the first iteration, the operation can linearize by returning *NULL* with the linearization point between the two iterations. ◻

Theorem 4.9. *A d-CBO queue is lock-free.*

Proof. Assume a period of time, with pending operations, where no enqueue linearizes for the d-CBO. We will prove that this period must be finite in length, as an enqueue or dequeue otherwise must complete. Inspecting enqueuers, we note that each d-CBO enqueue call (line 1.7) only performs d calls to *EnqCount* and one sub-queue *Enqueue* call before returning.

A d-CBO dequeue can invoke more sub-queue calls, but before it enters the *DoubleCollect* call it will similarly have performed d calls to *DeqCount* and one *Dequeue* at a sub-queue. After it enters *DoubleCollect* (line 1.18), it repeatedly (line 1.21) performs two loops containing up to n *Dequeue* and $2n$ *EnqVersion* calls. However, the *DoubleCollect* function will return *NULL* on line 1.28 in one of these iterations if no sub-queue *EnqVersion* has changed. The correctness condition on *EnqVersion* guarantees that it can only change a finite number of times before an enqueue linearizes.

Therefore, in this setting without enqueue progress, each d-CBO operation can only invoke a finite number of instructions and sub-queue calls before returning. As the sub-queues are lock-free, this duration without enqueue progress must either be finite in time or include dequeue linearizations, implying that the d-CBO queue is lock-free. ◻

5 Implementations

5.1 Integrating Fast Sub-Queues

Section 3 outlined the interface sub-queues must implement, and this section describes how to uphold those guarantees for four selected lock-free queues. We start with the simple MS queue [24] which is commonly used in earlier relaxed FIFO designs [15, 20, 33], and then move on to state-of-the-art lock-free queues designed around FAA.

An optimization used for all sub-queues is for their dequeues to linearize by returning *NULL* if *EnqCount* $\leq$ *DeqCount*. This reduces the number of increments of *DeqCount* and *EnqCount* which do not result in linearizations, improving the accuracy of the counters for the d-choice.

MS queue. Naturally, the MS queue [24] supports lock-free linearizable enqueue and dequeue methods. Furthermore, the by using counted head and tail pointers[1], as in the original

[1]This is a necessary slight overhead, as the counted pointers are not required inside the MS queue when using memory management such as hazard

design [24], we can implement exact *EnqCount* and *DeqCount* functions. Finally, as those counts are incremented during the linearization of operations, we can let *EnqVersion* = *EnqCount*. Note also that the *EnqCount* is *only* updated during the linearization of an enqueue.

FAAArrayQueue. The FAAArrayQueue [31] is an efficient, yet very simple, lock-free queue designed around FAA. It uses a linked list of queue segments where each segment has enqueue and dequeue counts used to assign operations to cells within the segment. By instrumenting each segment with a `segment_number`, increasing by one for each new segment, we can approximate the *EnqCount* as `tail.segment_number * SEGMENT_SIZE + min(tail.enq_count, SEGMENT_SIZE)`, where `tail` is the newest queue segment. Similarly, the *DeqCount* can be approximated by `head.segment_number * SEGMENT_SIZE + min(head.deq_count, SEGMENT_SIZE)`, where `head` is the oldest nonempty queue segment.

By limiting `enq_count` and `deq_count` from exceeding `SEGMENT_SIZE`, we ensure that each *EnqCount* corresponds to a unique tail item. Therefore, we can use *EnqCount* as *EnqVersion*. Furthermore, the queue is operation-wise lock-free as an enqueue will succeed system-wide every `SEGMENT_SIZE` attempts when enqueueing a new segment. Therefore it fulfills the requirement of one enqueue linearizing for some finite number of changes to *EnqVersion*. The *EnqCount* and *DeqCount* are not exact, as cells can be poisoned when the queue is close to empty, but the balanced operations spread these errors indiscriminately across the sub-queues.

WFQ. The wait-free queue (WFQ) [42] is similar to the FAAArrayQueue, with the addition that threads can help each other achieve wait-freedom. The WFQ has one central enqueue count and dequeue count for assigning operations to cells across segments. These can be used direcly as *EnqCount* and *DeqCount*, with very similar behavior as the FAAArrayQueue. By letting *EnqVersion* = *EnqCount*, the same argument as for the FAAArrayQueue guarantees a unique tail for every *EnqVersion*. The wait-freedom, together with the fact that the *EnqCount* is only incremented by enqueues, ensures that for some finite number of changes to *EnqVersion*, at least one enqueue must have linearized.

LCRQ. The LCRQ [26] is similar to the FAAArrayQueue, with the non-trivial optimization that segments are cyclic. This cyclic nature can improve performance in some cases, but also makes it more difficult to integrate it in *d*-CBO. Firstly, as the maximum size of each segment is unknown – as opposed to the FAAArrayQueue – we instrument each segment with a `starting_count`, which initialized to the `enq_count` of the previous segment. The *EnqCount* is then estimated as `tail.starting_count + tail.enq_count`, and *DeqCount* as `head.starting_count + head.deq_count`, where `tail` and `head` are the newest and oldest non-empty segments respectively.

pointers [23] or epoch-based reclamation [11]. Such counts are not added for the other sub-queues, as one then can use internal pre-existing counters.

A problem with the `starting_count` is that the `enq_count` of the previous segment can be incremented after the initialization of the new `starting_count`. This is a potential source of error when choosing the optimal sub-queue and also leads to the same *EnqCount* being reported for different tail items. Therefore, *EnqCount* cannot be used directly as *EnqVersion* and we instead let *EnqVersion* be the bitwise concatenation of `tail.enq_count` and `tail.starting_count`.

To fulfill the requirement that *EnqVersion* can only change a finite number of times before an enqueue linearizes, the optional *fixState* mechanism must be disabled. This is an optimization enabling dequeue operations to increase `enq_count`, which can cause dead-locks in the double-collect of an empty *d*-CBO. It is clear that each *EnqVersion* corresponds to a unique tail item, only being changed during the linearization of an enqueue. As the base LCRQ is lock-free, it now implements the sub-queue interface.

Memory Overhead. The memory usage of the *d*-CBO depends on the sub-queues used. Array-based queues such as the LCRQ, WFQ, and FAAArrayQueue incur a memory overhead due to the arrays not always being filled. This is compounded when using them as sub-queues, as the overhead is multiplied by the number of sub-queues used. In the worst case, each sub-queue has empty *head* and *tail* array segments, translating to `NBR_SUBQUEUES * 2 * SEGMENT_SIZE` empty cells, which bounds the *d*-CBO's memory overhead (disregarding potential overhead in used cell), as there is no replication of items. In memory-constrained environments, one should consider using smaller array segments, when using several sub-queues, compared to the array segments used in the original strict queues.

5.2 Relaxing Empty-Linearizability

As shown above, especially for the LCRQ, integrating new sub-queues for the *d*-CBO requires some care and understanding of the sub-queue properties. Even then, the *EnqCount* and *DeqCount* are not guaranteed to be accurate. Here we introduce the *Simplified d-Choice Balanced Operations queue* (Simple *d*-CBO), which relaxes the empty-linearity guarantee of the *d*-CBO in favor of a simplified design that can be used for any sub-queue, inheriting its properties.

Replacing the double-collect with a single-collect, where a dequeue returns *NULL* iff it has dequeued *NULL* from all sub-queues, removes the need for *EnqVersion* by sacrificing the empty-linearizability. For problems such as graph-traversals, where threads iteratively both dequeue and enqueue from the queue and can suspend after dequeueing *NULL*, the queue must be empty when all threads have dequeued *NULL* and suspended. This leads to the same termination conditions as if the queue was empty-linearizable.

Once the *EnqVersion* is made redundant, the d-CBO can be made completely generic over its sub-queue by also removing the need for sub-queue *EnqCount* and *DeqCount* implementations. Instead, one can create a wrapper for each sub-queue, with additional *EnqCount* and *DeqCount* fields that are updated with FAA after enqueues and non-empty dequeues. This sacrifices a bit of performance due to added contention at these counters, but will cause the counters to be more exact.

The relaxation analysis in Section 4.1 trivially still holds for this Simple d-CBO, and the only relaxation difference is that we allow relaxed empty returns. Relaxed empty returns are also part of the *quantitative relaxation* [18] and *distributed linearizability* specifications of out-of-order relaxations, when such empty returns are within the error bound. Furthermore, the Simple d-CBO inherits lock-free and linearizable properties directly from its sub-queue.

6 Experimental Results

We have implemented the d-CBO [38], using the sub-queues presented in the last section, in a C benchmark suite for relaxed data structures built on top of the ASCYLIB suite [6]. To evaluate our implementations, we first present a variety of synthetic benchmarks to evaluate the scalability of throughput and rank errors. Secondly, we have implemented a concurrent BFS algorithms that accommodates relaxed queues, which we use to compare the d-CBO against state-of-the-art relaxed and strict queues in a more realistic setting. All experiments use $d = 2$, as that has been shown to be a good choice in the literature [3, 15, 29, 32, 41], and was also verified in our experiments.

When comparing against state-of-the-art, we have selected the FAAArrayQueue[2] [31], WFQ[3] [42], and LCRQ[3] [26] as representatives of strict FIFO queues. For relaxed queues, we have selected the 2D queue[4] [33, 40], and the d-RA queue[2] [15, 20], as these are the best bounded and randomized designs we know of in the literature. The 2D queue always uses `depth=16` in our benchmarks, as it showed good trade-offs between performance and relaxation errors. All these implementations have been incorporated into our benchmarking suite [38], using SSMEM [6] for consistent and fast epoch-based memory management.

System Description. All experiments run on an AMD EPYC 9754 running at 2.25 GHz with 128 cores using two-way SMT, 256 MB L3 cache, and 755 GB RAM. The machine runs OpenSUSE Tumbleweed with the Linux 6.9.9 kernel. All experiments are written in C, using pthreads for concurrency and compiling with gcc 13.3.0 at optimization level O3. Software threads are pinned to hardware threads in a round-robin fashion between core clusters, starting to use SMT after 128 threads.

6.1 Benchmarking Scalability

To evaluate the performance of the d-CBO, we here evaluate its throughput and average rank error. Each data point shows the average and standard deviation of ten runs, each running for half a second. We mainly show results of using 128 sub-queues, as that empirically gave good results in the BFS benchmark, but also show sub-queue scalability in Figure 4. Furthermore, most benchmarks pre-fill the queues with 10^6 items to avoid empty returns for dequeues, which can cause deceivingly low relaxation errors or high throughput. Each experiment is run in two settings:

- **Random Enqueue/Dequeue**: Here, each thread repeatedly flips a fair coin and enqueues or dequeues an item based on the outcome.
- **Producer-Consumer**: Here half the threads are producers, repeatedly enqueueing items, and the other half consumers, repeatedly dequeueing items.

We measure rank errors with similar methodology as previous works [15, 41] by timestamping each operation and afterwards using the timestamps to re-create a sequential history. The rank errors can then be determined from the sequential history. This timestamping incurs some overhead, and relaxation is therefore measured in separate executions from the throughput measurements.

Efficient Sub-Queue Designs. We begin by evaluating the performance benefits of our design's ability to integrate the most effective queue designs. Figure 2 compares the scalability of our new d-CBO designs, including the simplified implementations. Unsurprisingly, the MS queue has the lowest throughput scalability, while the other sub-queues scale relatively similarly. The simplified versions are a bit slower than their standard versions, due to the extra work of updating the external counters. The simplified designs also demonstrate worse average rank error, especially at higher levels of concurrency. This is due to the external counters being updated *after* the sub-queue operations linearize, leading to slightly stale information for the d-choice sub-queue selection. The queues show very similar scalability in both the *Random Enqueue/Dequeue* and *Producer-Consumer* settings.

Load Balancing Schemes. To evaluate our new load balancing scheme, we have implemented the d-Choice Balanced-Lenghts (d-CBL) queue, which uses the previous d-RA load balancer [20] to select sub-queues based on their length. As shown in Figure 3, d-RA often leads to slightly lower throughput, as sampling must read both the enqueue and dequeue counts for each sub-queue. However, the standout differentiator is rank errors where the operation-based balancing is an order of magnitude better.

[2]Implemented ourselves.
[3]Code from https://github.com/chaoran/fast-wait-free-queue [42].
[4]Code from the relaxed 2D benchmarking suite [39].

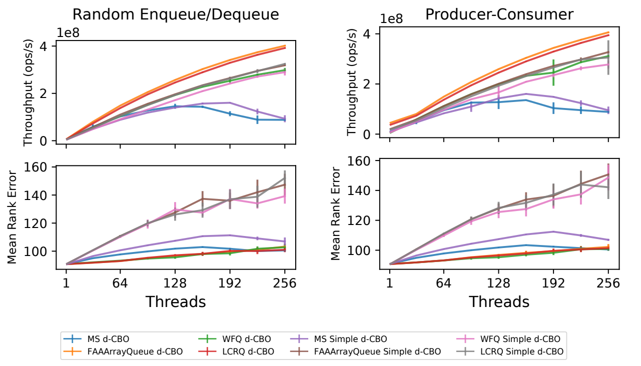


Figure 2. Evaluating the different d-CBO designs, using 128 sub-queues and 10^6 pre-fill.

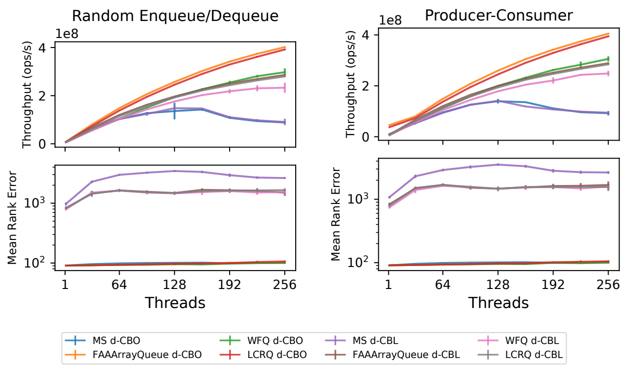


Figure 3. Comparing our operation-based load balancing with the earlier length-based balancing scheme, using 128 sub-queues and 10^6 pre-fill.

Scalability with Sub-Queues. Figure 4 shows how the d-CBO scales with increasing number of sub-queues. The MS queue is significantly slower – though its more accurate counters result in slightly better rank errors – at lower widths where sub-queue contention is higher. Our analysis bounds the rank error to $O(n \log\log n)$ w.h.p. in executions with non-overlapping operations. The figure includes a line of $k * n \log\log n + m$, where m and k are constants, which correlates very well with the measured rank errors. This suggests that the scalability in general concurrent executions also scale as $O(n \log\log n)$.

State-of-the-art. Experiments comparing the d-CBO with state-of-the-art designs [20, 26, 31, 33, 42] are shown in Figure 5, presenting results for both large and small pre-fill. For clarity, we only included the FAAArrayQueue d-CBO as it performed the best in the in previous comparisons. Both the simplified and standard d-CBO significantly outscale earlier designs in throughput while also maintaining a far lower rank error average than the 2D and d-RA designs. The d-RA manages to achieve relatively low rank errors in the presence of low pre-fill, but its heuristic is bad for large queues as also

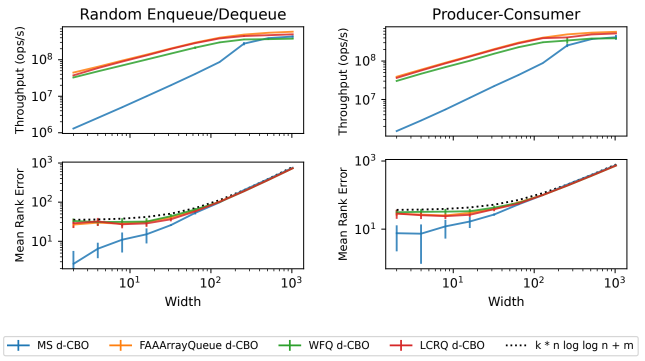


Figure 4. Evaluating how the d-CBO scales with the number of sub-queues, using 256 threads and 10^6 pre-fill. Includes line of $O(n \log\log n)$ for comparing with rank error scalability.

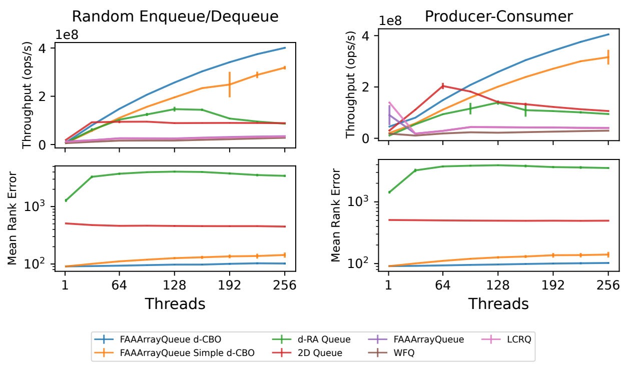
seen in Figure 3 and Figure 1. The reason for the high errors in larger queues is that the enqueue/dequeue counts for two sub-queues can drift relative to each other over time, as long as the difference between the counts for all queues stays the same.

Generally, the experiments with low pre-fill are more volatile, as *NULL* dequeue returns can significantly alter the performance. The number of d-CBO operations entering the double-collect in the benchmarks varied by sub-queue due to differences in dequeue and enqueue latencies. The MS queue was the only one with faster dequeues than enqueues, thus entering double-collect in around 20% of dequeues for the producer-consumer scenario. For the random enqueue/dequeue setting, we saw between four and ten percent of dequeues entering the double-collect with low pre-fill, and none at high pre-fill. In general, relaxed queues similar to the d-CBO are not as effective when the number of empty dequeues are significantly higher than this and you want low-latency empty dequeues, due to the double-collects. If one wants faster empty dequeues, the double-collect can be removed, or changed to a single-collect, at the cost of empty-linearizability.

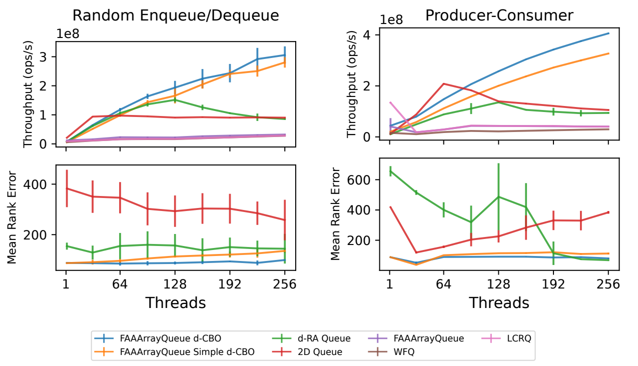
6.2 Concurrent BFS Benchmark

To evaluate the usability of the d-CBO, we utilize a concurrent BFS benchmark. In the BFS, as shown in Algorithm 2, threads iteratively, through a series of so-called relaxations (line 2.12), decrease the known fastest path to vertices. The algorithm continues until the queue is empty and all threads are idle, where the fastest path to each vertex is found, assuming unweighted edges.

We define the *work* of an execution as the number of times the tentative distance to a node is changed (line 2.12). In the sequential setting, the work will equal the number of nodes (assuming the graph is strongly connected), as one always processes the wavefronts in order. However, interleavings



(a) 10^6 pre-fill



(b) 10^3 pre-fill

Figure 5. Comparison against state-of-the-art, using 128 sub-queues for all relaxed queues, and `depth` = 16 for the 2D queue.

Algorithm 2: Concurrent BFS Algorithm

```
2.1  graph ← ReadToCSR()
2.2  distances ← [$\infty, \dots, \infty$]
2.3  queue ← [source]
2.4  distances[source] ← 0
2.5  concurrently while One thread has work do
2.6      at ← queue.Dequeue()
2.7      if at ≠ NULL then
2.8          at_dist ← distances[at]
2.9          for neigh ∈ neighbors(at, graph) do
2.10             neigh_dist ← distances[neigh]
2.11             while neigh_dist < at_dist + 1 do
2.12                 if CAS(distances[neigh], neigh_dist,
                       at_dist + 1) then
2.13                     queue.Enqueue(neigh)
2.14                     break
```

Graph	Nodes	Edges	Avg. Source Dist.
europe_osm	51M	108M	5028
road_usa	24M	58M	2762
road_central	14M	34M	1901
asia_osm	11M	25M	11619
hugebubbles-00000	18M	55M	3322
delaunay_n24	17M	101M	926
tx2010	914T	4.4M	123
coPapersDBLP	540T	30M	5.82

Table 1. Graphs used in BFS benchmark. All available in the SuiteSparse Matrix Collection [7].

and relaxation errors in concurrent executions can cause sub-optimal paths to be explored, leading to additional work.

For evaluation, we selected real-world graphs following previous research [27, 29] as shown in Table 1. We selected a set of road networks, one graph from numerical simulation, one from Delaunay triangulation, one from census data, and one research citation graph. They are all strongly connected and we arbitrarily select node 1 as the source. Table 1 also includes the average distance from the source, which is related to the diameter and heavily affects the BFS execution characteristic.

The results from running the BFS algorithm with our *d*-CBO and the selected state-of-the-art queues, is shown in Figure 6. Each data point shows the speedup and work increase compared to the sequential algorithm with a strict queue, averaged over 10 runs, including standard deviation.

The results vary significantly for different graphs, but the *d*-CBO consistently achives the highest speedup for all the graphs at high thread counts, with the FAAArrayQueue *d*-CBO often slightly outperforming the other FAA-based designs, as also shown in the synthetic benchmarks. The *d*-CBO often has to process more work than the sequential designs, but compensates with its superior throughput. The 2D queue consistently has among the highest work increases, especially at low thread counts, but manages to scale rather well and for the most part outperforms the strict queues. The *d*-RA performs similar to the MS *d*-CBO, as the queue likely often is relatively small, leading to lesser rank errors as also shown in Figure 5.

Interestingly, the strict queues can have rather high work increases as well, purely due to concurrency interleavings. Especially in the citation graph, with its low diameter, several of the strict queues do more work than the relaxed queues. This shows that the drawback of relaxed semantics, namely the relaxed order, in some cases does not pose a significant difference to normal concurrent designs. This is especially noticeable for relaxed queues such as the *d*-CBO with relatively small relaxation errors.

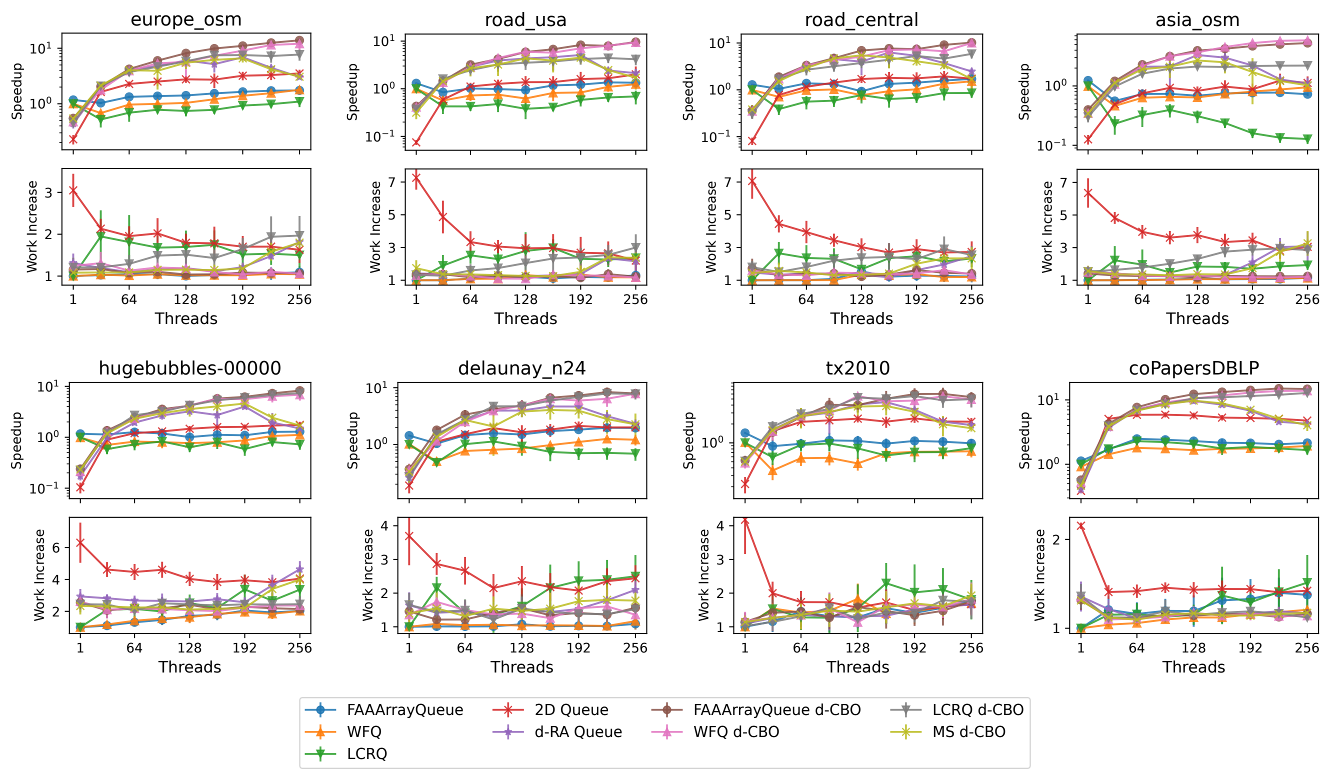


Figure 6. Comparing queues with a concurrent label-correcting BFS algorithm, using 128 sub-queues for all relaxed queues, and depth=32 for the 2D queue.

7 Conclusion

We introduced the relaxed d-CBO queue, that significantly improves on state-of-the-art designs in throughput and relaxation errors simultaneously. This is achieved by providing a new algorithmic design that i) is capable of interfacing against efficient sub-queues and ii) introduces a new load balancing scheme for balancing operations to sub-queues based on operation counts. Besides the experimental results that depict its performance improvements, we prove that the relaxation error is of $O(\frac{n \log\log n}{\log d})$ in executions with non-overlapping operations. This bound also correlates well with our experiments in all general concurrent executions. Finally, our concurrent BFS benchmark demonstrated the utility of relaxed queues, where d-CBO again consistently performed the best.

As future work, we would like to extend the relaxation error bounds to encompass all concurrent executions. Another interesting direction is to extend the algorithmic design to be able to elastically change its relaxation during runtime.

Acknowledgments

This work was supported by the Swedish Research Council under Grant No. 2021-05443.

References

[1] Dan Alistarh, Trevor Brown, Justin Kopinsky, Jerry Z. Li, and Giorgi Nadiradze. 2018. Distributionally Linearizable Data Structures. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures* (Vienna, Austria) *(SPAA '18, Vol. test)*. Association for Computing Machinery, New York, NY, USA, 133–142. https://doi.org/10.1145/3210377.3210411

[2] Dan Alistarh, Justin Kopinsky, Jerry Li, and Giorgi Nadiradze. 2017. The Power of Choice in Priority Scheduling. In *Proceedings of the ACM Symposium on Principles of Distributed Computing* (Washington, DC, USA) *(PODC '17)*. Association for Computing Machinery, New York, NY, USA, 283–292. https://doi.org/10.1145/3087801.3087810

[3] Yossi Azar, Andrei Z Broder, Anna R Karlin, and Eli Upfal. 1999. Balanced Alocations. *SIAM J. Comput.* 29, 1 (1999), 180–200. A preliminary version appeared in *Proceedings of the 26th Annual ACM Symposium on Theory of Computing*, pages 593–602, Montreal, Quebec, Canada, May 23–25, 1994. ACM Press, New York, NY..

[4] Petra Berenbrink, Artur Czumaj, Angelika Steger, and Berthold Vöcking. 2006. Balanced Allocations: The Heavily Loaded Case. *SIAM J. Comput.* 35, 6 (2006), 1350–1385. https://doi.org/10.1137/S009753970444435X

[5] Anastasia Braginsky, Nachshon Cohen, and Erez Petrank. 2016. CBPQ: High Performance Lock-Free Priority Queue. In *Euro-Par 2016: Parallel Processing*, Pierre-François Dutot and Denis Trystram (Eds.). Springer International Publishing, Cham, 460–474.

[6] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. 2015. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures. *SIGARCH Comput. Archit. News* 43, 1 (2015), 631–644. https://doi.org/10.1145/2786763.2694359

[7] Timothy A. Davis and Yifan Hu. 2011. The university of Florida sparse matrix collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (Dec. 2011), 25 pages. https://doi.org/10.1145/2049662.2049663

[8] Mike Dodds, Andreas Haas, and Christoph M. Kirsch. 2015. A Scalable, Correct Time-Stamped Stack. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) *(POPL '15)*. Association for Computing Machinery, New York, NY, USA, 233–246. https://doi.org/10.1145/2676726.2676963

[9] Faith Ellen, Danny Hendler, and Nir Shavit. 2012. On the Inherent Sequentiality of Concurrent Objects. *SIAM J. Comput.* 41, 3 (2012), 519–536. https://doi.org/10.1137/08072646X

[10] Panagiota Fatourou and Nikolaos D. Kallimanis. 2011. A highly-efficient wait-free universal construction. In *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures* (San Jose, California, USA) *(SPAA '11)*. Association for Computing Machinery, New York, NY, USA, 325–334. https://doi.org/10.1145/1989493.1989549

[11] Keir Fraser. 2003. *Practical lock-freedom.* Ph. D. Dissertation. University of Cambridge.

[12] Anders Gidenstam, Håkan Sundell, and Philippas Tsigas. 2010. Cache-Aware Lock-Free Queues for Multiple Producers/Consumers and Weak Memory Consistency. In *Principles of Distributed Systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, 302–317. https://doi.org/10.1007/978-3-642-17653-1_23

[13] Elad Gidron, Idit Keidar, Dmitri Perelman, and Yonathan Perez. 2012. SALSA: scalable and low synchronization NUMA-aware algorithm for producer-consumer pools. In *Proceedings of the Twenty-Fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures* (Pittsburgh, Pennsylvania, USA) *(SPAA '12)*. Association for Computing Machinery, New York, NY, USA, 151–160. https://doi.org/10.1145/2312005.2312035

[14] Gaston H. Gonnet. 1981. Expected Length of the Longest Probe Sequence in Hash Code Searching. *J. ACM* 28, 2 (apr 1981), 289–304. https://doi.org/10.1145/322248.322254

[15] Andreas Haas, Michael Lippautz, Thomas A. Henzinger, Hannes Payer, Ana Sokolova, Christoph M. Kirsch, and Ali Sezgin. 2013. Distributed queues in shared memory: multicore performance and scalability through quantitative relaxation. In *Proceedings of the ACM International Conference on Computing Frontiers* (Ischia, Italy) *(CF '13)*. Association for Computing Machinery, New York, NY, USA, Article 17, 9 pages. https://doi.org/10.1145/2482767.2482789

[16] Timothy L. Harris. 2001. A Pragmatic Implementation of Non-blocking Linked-Lists. In *Proceedings of the 15th International Conference on Distributed Computing (DISC '01)*. Springer-Verlag, Berlin, Heidelberg, 300–314.

[17] Danny Hendler, Nir Shavit, and Lena Yerushalmi. 2010. A Scalable Lock-Free Stack Algorithm. *J. Parallel and Distrib. Comput.* 70, 1 (2010), 1–12. https://doi.org/10.1016/j.jpdc.2009.08.011

[18] Thomas A. Henzinger, Christoph M. Kirsch, Hannes Payer, Ali Sezgin, and Ana Sokolova. 2013. Quantitative Relaxation of Concurrent Data Structures. *SIGPLAN Not.* 48, 1 (2013), 317–328. https://doi.org/10.1145/2480359.2429109

[19] Moshe Hoffman, Ori Shalev, and Nir Shavit. 2007. The Baskets Queue. In *Principles of Distributed Systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, 401–414. https://doi.org/10.1007/978-3-540-77096-1_29

[20] Christoph M. Kirsch, Hannes Payer, Harald Röck, and Ana Sokolova. 2012. Performance, scalability, and semantics of concurrent FIFO queues. In *Proceedings of the 12th International Conference on Algorithms and Architectures for Parallel Processing - Volume Part I* (Fukuoka, Japan) *(ICA3PP'12)*. Springer-Verlag, Berlin, Heidelberg, 273–287. https://doi.org/10.1007/978-3-642-33078-0_20

[21] Alex Kogan and Erez Petrank. 2012. A methodology for creating fast wait-free data structures. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New Orleans, Louisiana, USA) *(PPoPP '12)*. Association for Computing Machinery, New York, NY, USA, 141–150. https://doi.org/10.1145/2145816.2145835

[22] Jonatan Lindén and Bengt Jonsson. 2013. A Skiplist-Based Concurrent Priority Queue with Minimal Memory Contention. In *Principles of Distributed Systems*. Springer International Publishing, Cham, 206–220.

[23] Maged M. Michael. 2004. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Trans. Parallel Distrib. Syst.* 15, 6 (jun 2004), 491–504. https://doi.org/10.1109/TPDS.2004.8

[24] Maged M. Michael and Michael L. Scott. 1996. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing* (Philadelphia, Pennsylvania, USA) *(PODC '96)*. Association for Computing Machinery, New York, NY, USA, 267–275. https://doi.org/10.1145/248052.248106

[25] Mark Moir, Daniel Nussbaum, Ori Shalev, and Nir Shavit. 2005. Using elimination to implement scalable and lock-free FIFO queues. In *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures* (Las Vegas, Nevada, USA) *(SPAA '05)*. Association for Computing Machinery, New York, NY, USA, 253–262. https://doi.org/10.1145/1073970.1074013

[26] Adam Morrison and Yehuda Afek. 2013. Fast concurrent queues for x86 processors. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Shenzhen, China) *(PPoPP '13)*. Association for Computing Machinery, New York, NY, USA, 103–112. https://doi.org/10.1145/2442516.2442527

[27] Maxim Naumov, Alysson Vrielink, and Michael Garland. 2017. Parallel Depth-First Search for Directed Acyclic Graphs. In *Proceedings of the Seventh Workshop on Irregular Applications: Architectures and Algorithms* (Denver, CO, USA) *(IA3'17)*. Association for Computing Machinery, New York, NY, USA, Article 4, 8 pages. https://doi.org/10.1145/3149704.3149764

[28] Gary L. Peterson and James E. Burns. 1987. Concurrent reading while writing II: The multi-writer case. In *Proceedings of the 28th Annual Symposium on Foundations of Computer Science (SFCS '87)*. IEEE Computer Society, USA, 383–392. https://doi.org/10.1109/SFCS.1987.15

[29] Anastasiia Postnikova, Nikita Koval, Giorgi Nadiradze, and Dan Alistarh. 2022. Multi-queues can be state-of-the-art priority schedulers. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Seoul, Republic of Korea) *(PPoPP '22)*. Association for Computing Machinery, New York, NY, USA, 353–367. https://doi.org/10.1145/3503221.3508432

[30] Martin Raab and Angelika Steger. 1998. "Balls into Bins" — a Simple and Tight Analysis. In *Randomization and Approximation Techniques in Computer Science*, Michael Luby, José D. P. Rolim, and Maria Serna (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 159–170.

[31] Pedro Ramalhete. 2016. FAAArrayQueue - MPMC Lock-Free Queue. http://concurrencyfreaks.blogspot.com/2016/11/faaarrayqueue-mpmc-lock-free-queue-part.html Accessed: 2024-07-29.

[32] Hamza Rihani, Peter Sanders, and Roman Dementiev. 2015. MultiQueues: Simple Relaxed Concurrent Priority Queues. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures* (Portland, Oregon, USA) *(SPAA '15)*. Association for Computing Machinery, New York, NY, USA, 80–82. https://doi.org/10.1145/2755573.2755616

[33] Adones Rukundo, Aras Atalar, and Philippas Tsigas. 2019. Monotonically Relaxing Concurrent Data-Structure Semantics for Increasing Performance: An Efficient 2D Design Framework. In *33rd International Symposium on Distributed Computing (DISC 2019) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 146)*, Jukka Suomela (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 31:1–31:15. https://doi.org/10.4230/LIPIcs.DISC.2019.31

[34] Nir Shavit. 2011. Data Structures in the Multicore Age. *Commun. ACM* 54, 3 (2011), 76–84. https://doi.org/10.1145/1897852.1897873

[35] Håkan Sundell, Anders Gidenstam, Marina Papatriantafilou, and Philippas Tsigas. 2011. A lock-free algorithm for concurrent bags. In *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures* (San Jose, California, USA) *(SPAA '11)*. Association for Computing Machinery, New York, NY, USA, 335–344. https://doi.org/10.1145/1989493.1989550

[36] Kunal Talwar and Udi Wieder. 2014. Balanced Allocations: A Simple Proof for the Heavily Loaded Case. In *Automata, Languages, and Programming*. Springer Berlin Heidelberg, Berlin, Heidelberg, 979–990.

[37] Philippas Tsigas and Yi Zhang. 2001. A simple, fast and scalable non-blocking concurrent FIFO queue for shared memory multiprocessor systems. In *Proceedings of the Thirteenth Annual ACM Symposium on Parallel Algorithms and Architectures* (Crete Island, Greece) *(SPAA '01)*. Association for Computing Machinery, New York, NY, USA, 134–143. https://doi.org/10.1145/378580.378611

[38] Kåre von Geijer. 2025. Artifact for: Balanced Allocations over Efficient Queues: A Fast Relaxed FIFO Queue. https://doi.org/10.5281/zenodo.14223312

[39] Kåre von Geijer and Philippas Tsigas. 2024. Artifact of the paper: How to Relax Instantly: Elastic Relaxation of Concurrent Data Structures. https://doi.org/10.5281/zenodo.11547063

[40] Kåre von Geijer and Philippas Tsigas. 2024. How to Relax Instantly: Elastic Relaxation of Concurrent Data Structures. In *Euro-Par 2024: Parallel Processing*. Springer Nature Switzerland, Cham, 119–133. https://doi.org/10.1007/978-3-031-69583-4_9

[41] Marvin Williams, Peter Sanders, and Roman Dementiev. 2021. Engineering MultiQueues: Fast Relaxed Concurrent Priority Queues. In *29th Annual European Symposium on Algorithms (ESA 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 204)*, Petra Mutzel, Rasmus Pagh, and Grzegorz Herman (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 81:1–81:17. https://doi.org/10.4230/LIPIcs.ESA.2021.81

[42] Chaoran Yang and John Mellor-Crummey. 2016. A wait-free queue as fast as fetch-and-add. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Barcelona, Spain) *(PPoPP '16)*. Association for Computing Machinery, New York, NY, USA, Article 16, 13 pages. https://doi.org/10.1145/2851141.2851168

A Artifact Overview

This section gives an overview of the complementary artifact to this paper, which is available as a separate Zenodo archive [38]. Here we give a brief overview of the artifact, how to set it up, and rerun the experiments from the paper, and we refer to the full description in the artifact for further details.

The artifact consists of two repositories:

- **relaxed-queue-simulations** contains Rust code for simulating the rank errors in sequential executions of *choice-of-two* queues, encompassing the d-CBO and d-RA [20]. This is used to recreate Figure 1.
- **relaxed-benchmarking-suite** contains a library of concurrent data structures in C. This suite contains all the benchmarks needed to recreate the experiments from Section 6. These implementations require an x86-64 machine.

Both repositories use Docker[5] to set up their environments. Refer to https://docs.docker.com/engine/install/ for setup instructions. For brevity, we will here only include instructions for pulling the pre-built images from DockerHub.

A.1 Relaxed Queue Simulations

To rerun the experiment from Figure 1, you can simply pull the following image, containing the repository and an environment, and run it.

```
$ docker pull khorium/relaxed-queue-simulations:ppopp
$ docker run --rm -v $(pwd)/results:/app/results    \
    khorium/relaxed-queue-simulations:ppopp
```

This will generate two PDFs in `results/` containing the two heatmaps. To save time, the image is set up to only run one run for every ocnfiguration, instead of averaging over 100 as in Figure 1. The trends should still be clearly visible, but one can also start the container interactively with the `-it` flag, modify the script `recreate-ppopp.sh` to do 100 runs, and then run it.

A.2 Relaxed Benchmarking Suite

This repository is built around the x86-64 architecture and Linux. It will likely run on other systems, but could give incorrect or misleading results. First, pull the image with the following command.

```
$ docker pull khorium/relaxed-benchmarking-suite:ppopp
```

The repository contains `scripts/recreate-ppopp.sh`, which can be run to rerun all benchmarks from Section 6. This in turn runs several Python scripts, which compile and run the C benchmarks. The shell script contains values for controlling the evaluation, which you likely want to adapt based on your hardware (the ones used in the paper are included as comments in the script):

- `nbr_threads` sets the maximum number of threads. Adjust this based on your number of available hardware threads.
- `duration` is the number of milliseconds to run the throughput benchmmarks for.
- `relaxation_duration` is the number of milliseconds to run the relaxation benchmarks for. It can take quite a long time after the execution to compute the errors, and it is therefore suggested to set this a bit lower for the experiments to not take too long.
- `runs` is the number of runs to average over for each configuration.
- `step` is the number (plus one) of different thread configurations to run.

The benchmarks pin software threads to hardware threads to achieve stable and representative results. However, this has to be configured individually for each machine, as the hardware thread numbering varies between systems. The pinning strategy used depends on the hostname, which is

[5]They were developed using Docker version 26.1.5-ce, build 411e817ddf71.

why we set the hostname to `example-pinning` when running the container below, which is a configuration that allocates hardware threads in sequential order. This is not always desired, and we then refer to the full artifact description [38].

So, to recreate the evaluation from the paper, start the container interactively, configure the execution, and run it as follows.

```
$ docker run -it --rm --hostname=example-pinning  \
  -v $(pwd)/results:/app/results                  \
  khorium/relaxed-benchmarking-suite:ppopp bash
# Edit pinning, and scripts/recreate-ppopp.sh
$ bash scripts/recreate-ppopp.sh
```

This creates a subfolder in `./results/` for each experiment, including the PDF of the plot, as well as the collected data. The experiments are expected to take about an hour with the given settings. To simplify comparing the generated figures to the ones in the paper, we here present a mapping from the paper figures to the names of the generated folders in `results/`.

- **Figure 2**: *simple-enq-deq* (left) and *simple-prod-con* (right).
- **Figure 3**: *dcbl-enq-deq* (left) and *dcbl-prod-con* (right).
- **Figure 4**: *subqueue-scalability-enq-deq* (left) and *subqueue-scalability-prod-con* (right).
- **Figure 5a**: *sota-enq-deq-large* (left) and *sota-prod-con-large* (right).
- **Figure 5b**: *sota-enq-deq-small* (left) and *sota-prod-con-small* (right).
- **Figure 6**: Eight folders named *bfs-<graph_name>*.