How to Relax Instantly: Elastic Relaxation of Concurrent Data Structures

Kåre von Geijer[1 [0009−0007−4823−6855]] and Philippas Tsigas[1 [0000−0001−9635−9154]]

Chalmers University of Technology, Gothenburg, Sweden

Abstract. The sequential semantics of many concurrent data structures, such as stacks and queues, inevitably lead to memory contention in parallel environments, thus limiting scalability. Semantic relaxation has the potential to address this issue, increasing the parallelism at the expense of weakened semantics. Although prior research has shown that improved performance can be attained by relaxing concurrent data structure semantics, there is no one-size-fits-all relaxation that adequately addresses the varying needs of dynamic executions.
In this paper, we first introduce the concept of *elastic relaxation* and consequently present the *Lateral* structure, which is an algorithmic component capable of supporting the design of elastically relaxed concurrent data structures. Using the *Lateral*, we design novel elastically relaxed, lock-free queues and stacks capable of reconfiguring relaxation during run-time. We establish linearizability and define upper bounds for relaxation errors in our designs. Experimental evaluations show that our elastic designs hold up against state-of-the-art statically relaxed designs, while also swiftly managing trade-offs between relaxation and operational latency. We also outline how to use the *Lateral* to design elastically relaxed lock-free counters and deques.

Keywords: concurrent data structures · lock-free · relaxed semantics

1 Introduction

As hardware parallelism advances with the development of multicore and multiprocessor systems, developers face the challenge of designing data structures that efficiently utilize these resources. Numerous concurrent data structures exist [14], but theoretical results such as [6] demonstrate that many common data structures, such as queues, have inherent scalability limitations as threads must contend for a few access points. One of the most promising solutions to tackle this scalability issue is to relax the sequential specification of data structures [27], which permits designs that increase the number of memory access points, at the expense of weakened sequential semantics.

The k *out-of-order* relaxation formalized in [13] is a popular model [26,10,18,30] that allows relaxed operations to deviate from the sequential order by up to k; for example, for the dequeue operation on a FIFO queue, any of the first $k + 1$ items can be returned instead of just the head. This error, the distance from

the head for a dequeue, is called the *rank error*. While other relaxations, such as quiescent consistency [5] are incompatible with linearizability [15], k out-of-order relaxation can easily be combined with linearizability, as it modifies the semantics of the data structure instead of the consistency. Despite extensive work on out-of-order relaxation [10,13,26,16,29,24,30], almost all existing methods are static, requiring a fixed relaxation degree during the data structures' lifetime.

In applications with dynamic workloads, such as bursts of activity with throughput constraints, it is essential to be able to temporarily sacrifice sequential semantics for improved performance. This is the problem tackled in this paper, to specify and design relaxed data structures where the relaxation is reconfigurable during run-time, which we term *elastic relaxation*. Elastically relaxed data structures enable the design of instance-optimizing systems, an area that is evolving extremely rapidly across various communities [19]. The trade-off between rank error and throughput is highlighted in [29] and [24], where their shortest-path benchmarks show that increased relaxation leads to higher throughput, but at the expense of additional required computation.

Several relaxed data structures are implemented by splitting the original concurrent data structure into disjoint *sub-structures*, and then using load-balancing algorithms to direct different operations to different sub-structures. In this paper, we base our elastic designs on the relaxed 2D framework presented in [26], which has excellent scaling with both threads and relaxation, as well as provable rank error bounds. The key idea of the 2D framework is to superimpose a window (Win) over the sub-structures, as seen in green in Figure 1 for the 2D queue, where operations inside the window can linearize out of order. The Win^{tail} shifts upward by *depth* when it is full, and Win^{head} shifts upward when emptied, to allow further operations. The size of the window dictates the rank error, as a larger one allows for more reorderings.

The algorithmic design concept we propose in this paper is the *Lateral* structure that can extend the 2D structures to encompass elastic relaxation. This *Lateral* is a strict concurrent version of the relaxed data structure, kept to the side of the sub-structures to keep track of the elastic changes, as shown in Figure 2. We show how to incorporate the *Lateral* into the window mechanism that the 2D framework introduced while achieving a deterministic rank error bound. Although we chose to use the 2D framework as a base for our designs, the *Lateral* can also accommodate other designs, such as the distributed queues from [10], the k-queue from [18], and the k-stack from [13].

Contributions. This work takes crucial steps toward designing reconfigurable relaxed concurrent data structures with deterministic error bounds, capable of adjusting relaxation levels during run-time.

- Firstly, we introduce the concept of *elastic relaxation*, allowing the rank error to change over time. Furthermore, we introduce the *Lateral* component for efficiently enhancing relaxed data structures with elasticity.
- We design and implement elastically relaxed queues and a stack using the *Lateral*, also establishing their correctness and rank error bounds. Additionally, we outline how to incorporate the *Lateral* into counters and deques.

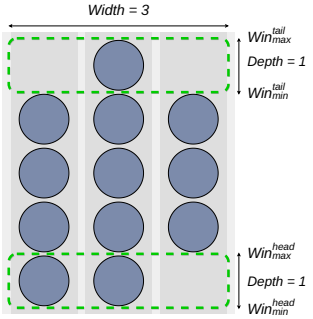


Fig. 1: The 2D queue has two windows defining the operable area for the enqueue and dequeue operations.

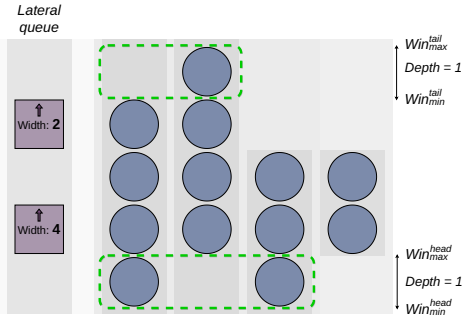


Fig. 2: By adding a *Lateral* to the 2D queue, changes in width at Win^{tail} can be tracked and adjusted to by Win^{head}.

– We present an extensive evaluation of our proposed data structures, benchmarking against both non-relaxed and relaxed data structures. These evaluations show that the elastic designs significantly outscale non-relaxed data structures. and perform as well as the statically relaxed ones, while also supporting elastic relaxation.
– Finally, we showcase the elastic capabilities of our design by implementing a minimalistic controller for a producer-consumer application. By dynamically adjusting the relaxation, it is able to control the producer latency during bursts of activity with minimal overhead.

Structure. Section 2 introduces related work, as well as gives a short description of the structures we base our elastic designs on. Section 3 introduces elastic relaxation and our novel data structures, which we then prove correct and provide worst-case bounds for in Section 4. Section 5 experimentally evaluates the new algorithms, both comparing them to earlier non-elastic data structures as well as testing their elastic capabilities. Section 6 contains a few closing remarks.

2 Related Work

One of the earliest uses of relaxed data structures is from 1993 by Karp and Zhang in [17]. However, they did not focus on the relaxation, and it was not until more recently that relaxed data structures emerged as a promising technique to boost concurrency [27]. They have demonstrated exceptional throughput on highly parallel benchmarks [26,29,9,10], have proven suitable in heuristics for graph algorithms [22,24], and been theoretically analyzed in e.g. [13,1,11,2].

Henzinger et al. specified *quantitative relaxation* in [13] to define relaxed data structures with a rank error bound. The paper also introduces the relaxed k-segment stack, which in turn builds upon the earlier relaxed FIFO queue from

[18]. Their theoretical framework is easy to extend to encompass elastic relaxation by allowing the rank error bound to vary over time, and it is simple to elastically extend their designs using our *Lateral*.

Instead of bounding the rank error, designs such as the MultiQueue [29,25] instead only give probibalistic guarantees. This MultiQueue is a relaxed priority queue, which, for example, has proven useful on shortest-path graph algorithms [24]. It enqueues items into random sub-queues, while dequeuing the highest priority item from a random selection of two sub-queues. This can be applied similarly to FIFO queues [10] and stacks. The probabilistic rank error guarantees of the MultiQueue were extensively analyzed in [2].

The SprayList from [3] is another probibalistically relaxed priority queue. Although experimentally outperformed by the MultiQueue in [29,25,24], the SprayList is the only relaxed data structure we found in the literature that can reconfigure its relaxation during run-time.

The relaxed queues introduced in [16] employ relaxation to reach significantly lower wait-times (especially in nearly empty queues) than the strict LCRQ [21] on which they are based. While offering a new perspective on concurrent queue design goals, these queues lack mechanisms to configure their relaxation.

2D Framework The 2D framework for k out-of-order relaxed data structures from [26] (hereby called the *static* 2D designs) outscales other implementations from the literature such as [18,13,10] with threads, while unifying designs across stacks, queues, deques, and counters. Furthermore, its throughput scales monotonically with k. It achieves this, as shown in Figure 1, by superimposing a *window* (Win) over multiple disjoint concurrent (nonrelaxed) sub-structures which defines which of them are valid to operate on.

The 2D *window* has a *width* (always the number of sub-structures in the static designs) and a *depth*, which together govern the relaxation profile. At any point, it is valid to insert an item on a *row* r where $r \leq Win_{max}$, or delete an item on row r where $r > Win_{min} \equiv Win_{max} - depth$, while keeping linearization order in the sub-structures. To maximize data locality, each thread tries to do as many operations as possible on the same sub-structure in a row, only moving due to contention or from reaching Win_{max} or Win_{min}.

If an operation cannot find a valid sub-structure, it will try to *shift* the window. For example, if a thread tries to insert an item into the 2D queue and sees all sub-queues at Win^{tail}_{max}, it will try to shift the window up by atomically incrementing Win^{tail}_{max} (and implicitly Win^{tail}_{min}) by *depth*, after which it restarts the search for a valid sub-queue.

The 2D stack is similar to the 2D queue, but as both insert (push) and delete (pop) operate on the same side of the data structure, only one window is used. This leads to Win_{max} no longer increasing monotonically, but instead decreasing under delete heavy workloads, and increasing under insert heavy ones. Furthermore, Win_{max} shifts by $depth/2$ instead of *depth*, which roughly makes it fair for future push and pop operations. The framework also covers deques and counters, using the same idea of a window to define valid operations.

3 Design of Elastic Algorithms

Static k out-of-order relaxation is formalized in [13] by defining and bounding a *transition cost* (rank error) of the "get" methods within the linearized concurrent history. Elastic relaxation allows the relaxation configuration to change over time, which will naturally change the bound k as well. Therefore, we define *elastic out-of-order* data structures as static out-of-order, but allow the rank error bound to be a function of the relaxation configuration history during the lifetime of the accessed item. In the simplest case, such as the elastic queue from Section 3.1, the rank error bound for every dequeued item is a function of the *width* and *depth* during which the item is dequeued.

To elastically extend the 2D algorithms, we want the *width* and *depth* of the 2D windows to be adjustable during run-time, which would enable fine-grained control of the relaxation. Our designs let these parameters change every window shift, and then update them atomically with Win_{max}. Changing *depth* is practically simple by including it as a window variable Win_{depth} (although it affects the error bound). Varying the *width* efficiently requires more attention. During initialization, we create a fixed number of sub-structures, and then utilize the *Lateral* to track the width of populated rows, as illustrated in Figure 2.

Definition 1. *A Lateral to a relaxed data structure is a set of nonoverlapping adjacent ranges of rows, where each range has a corresponding width bound.*

Furthermore, we call the *Lateral consistent* if the *width bound* of each node properly bounds the width of the corresponding rows in the main structure. The exact implementation of the *Lateral* will vary depending on what performance properties are desired, but the overarching challenge is to keep it consistent while also being fast to read and update, as well as promising good rank error bounds. For our 2D designs, we found that it is essential to at most update the *Lateral* once every window.

3.1 Elastic Lateral-as-Window 2D Queue

This first elastic Lateral-as-Window queue (LaW queue) merges the window into the *Lateral*, and its ideas can be applied to most data strucutures from the 2D framework [26] with small changes. The pseudocode is shown in Algorithm 1. First, we add a *Lateral* queue that is implemented as a Michael-Scott queue [20], for which the code omits the standard failable *Enqueue* and *Dequeue* methods. The *Lateral* nodes are windows, where each window contains Win_{max}, Win_{depth}, and Win_{width}. The Win^{tail} and Win^{head} then become the head and tail nodes in the *Lateral*. Every shift of Win^{tail} enqueues a new window in the *Lateral* (line 1.11), and every shift of Win^{head} dequeues a window (line 1.19).

As shown in *ShiftTail* (line 1.11), Win_{depth} and Win_{width} can be updated from shared variables every shift, which enables the elasticity. The main drawback of this design is that the relaxation can only change at Win^{tail}, and must propagate through the queue to reach Win^{head}.

Algorithm 1: Pseudocode for the *Lateral* in the elastic LaW queue

```
1.1  struct Window
1.2  |   Window* next;
1.3  |   uint max;
1.4  |   uint depth;
1.5  |   uint width;
1.6  global struct Lateral
1.7  |   Window* head;
1.8  |   Window* tail;
     // Try to atomically enqueue new directly
        after expected
1.9  method Lateral.Enqueue(expected, new);
     // Try to atomically dequeue expected if it
        is the head
1.10 method Lateral.Dequeue(expected);
1.11 method Lateral.ShiftTail(old_window)
1.12 |   depth ← depth_shared;
1.13 |   new_window ← {
1.14 |     width: width_shared,
1.15 |     depth: depth,
1.16 |     max: old_window.max + depth
1.17 |   };
1.18 |   Lateral.Enqueue(old_window, new_window);
1.19 function ShiftHead(current_head)
1.20 |   Lateral.Dequeue(current_head);
```

Other than the pseudocode in Algorithm 1, the remaining logic from the static 2D queue require only small adjustments. Mainly, items must always be inserted within the window, so each item gets enqueued at $max(Win^{tail}_{max} - Win^{tail}_{depth}, last\_item.row) + 1$, which create the needed gaps in sub-queues as seen in Figure 2. Furthermore, Win^{head} cannot pass Win^{tail}, and dequeues can simply return empty if the *Lateral* is empty.

3.2 Elastic Lateral-plus-Window 2D Queue

The elastic Lateral-plus-Window 2D queue (elastic LpW queue) solves two shortcomings of the previous elastic LaW queue. Firstly, it allows the head to change relaxation independently of the tail by letting both windows change Win_{depth} at window shifts. Second, it does not have to allocate a new *Lateral* node every Win^{tail} shift, and instead only creates *Lateral* nodes when Win^{tail}_{width} changes. However, it comes at the expense of having to decouple the window and *Lateral* components, and uses a 128-bit shared atomic struct for each window.

The pseudocode for the *Lateral* and windows is presented in Algorithm 2 and shows that the Win^{head} and Win^{tail} structs now are global variables, both containing *max*, *depth*, and *width*. The *Lateral* is again implemented as a Michael-Scott queue [20] where we omit the *Enqueue* and *Dequeue* implementations.

Shifting Win^{tail} occurs once a thread observes all sub-structures at or above Win^{tail}_{max} (line 2.27). It reads the desired *depth* and *width* from shared variables, which are used in the next window. However, the *width* is not used immediately, but instead written to a *next_width* field, which is then used in the successive shift as the new *width* (line 2.31). This delay is introduced to ensure that a *Lateral* node will be enqueued with the new *width* before this *width* is used in an enqueue. Enqueueing such a *Lateral* node is done before the window shift when $next\_width \neq width$ (line 2.28), ensuring that the head of the queue will be aware of changes in width before they occur.

Similarly, Win^{head} is shifted during a dequeue call when all sub-queues in the *width* has reached the window *max*. The shift starts by dequeueing all *Lateral* nodes below the current Win^{head}_{max} (loop at line 2.38), as they represent stale

Algorithm 2: Lateral and window code for the elastic LpW queue

```
2.1  global struct TailWindow
2.2      uint64 max;
2.3      uint16 depth;
2.4      uint16 width;
2.5      uint16 next_width;
2.6  global struct HeadWindow
2.7      uint64 max;
2.8      uint16 depth;
2.9      uint16 width;
2.10 struct LateralNode
2.11     LateralNode* next;
2.12     uint row;
2.13     uint width;
2.14 global struct Lateral
2.15     LateralNode* head;
2.16     LateralNode* tail;
     // Try to atomically enqueue new directly
        after expected
2.17 method Lateral.Enqueue(expected, new);
     // Try to atomically dequeue expected if it
        is the head
2.18 method Lateral.Dequeue(expected);
2.19 method Lateral.SyncTail(window)
2.20     tail ← Lateral.tail;
2.21     if tail.row ≤ window.max then
2.22         new_tail ← {
2.23             row: window.max + 1,
2.24             width: window.next_width
2.25         };
2.26         Lateral.Enqueue(tail, new_tail);
2.27 function ShiftTail(old_window)
2.28     if window.width ≠ window.next_width
           Lateral.SyncTail(old_window) ;
2.29     depth ← depth_shared;
2.30     new_window ← {
2.31         width: old_window.next_width,
2.32         next_width: width_shared,
2.33         depth: depth,
2.34         max: old_window.max + depth
2.35     };
2.36     CAS(&TailWindow, old_window,
           new_window);
2.37 function ShiftHead(old_window)
2.38     while true do
2.39         head ← Lateral.head();
2.40         if head.max > old_window.max break;
2.41         Lateral.Dequeue(head);
2.42     new_window ← {
2.43         width: old_window.width,
2.44         max: min(TailWindow.max,
2.45                  old_window.max + depth_shared)
2.46     };
2.47     if head.row = old_window.max + 1 then
2.48         new_window.width ← head.width;
2.49         head ← head.next;
2.50     if head.row < new_window.max then
2.51         new_window.max ← head.row - 1;
2.52     new_window.depth ← new_window.max -
           old_window.max;
2.53     CAS(&HeadWindow, old_window,
           new_window);
```

changes in width. The shift tries to increment Win_{max}^{head} by a shared *depth* variable (line 2.45), but sometimes has to shift it less as to not overtake Win_{max}^{tail} (line 2.44) or the next *Lateral* node (line 2.50). If the *Lateral* head is on the bottom row of the new window (line 2.48), then the new window will adapt to the width change encoded by the head. Otherwise, the new Win_{max}^{head} is limited to not overlap a *Lateral* node (line 2.50). This ensures that all nodes within a window were pushed within the same *width*, which is used at dequeues to calculate which row the dequeued node was at.

At the cost of separating the *Lateral* and the window, this LpW queue is able to change *depth* independently for the head and the tail. However, the *width* is still only ever changed at Win^{tail} which Win^{head} has to adapt to by using the *Lateral*. We have designed this to be efficient on modern x86-64 machines where CAS only has hardware support for up to 128 bits, which then becomes the upper size limits for our window structs. One can allocate the sizes differently depending on the need of the application, but if 128 bits is not enough, or a machine without 128-bit CAS support is used, the elastic LaW queue might be more suitable.

3.3 Elastic Lateral-plus-Window 2D Stack

This elastic Lateral-plus-Window 2D stack (elastic LpW stack) uses the same idea as the elastic LpW queue to encompass elasticity. Due to space limitations, we here give the key ideas and refer the reader to our extended version [8] for the complete algorithm description. The stack maintains a global shared window struct, which is updated with CAS at window shifts, and a Treiber stack [28] as a *Lateral* for all changes in *width*. Unlike the queue, the nonmonotonic nature of its Win_{max} means that the width bound of a row can change repeatedly, and thus the *Lateral* must be continuously stabilized.

The push and pop operations are very similar to the static 2D stack, except they operate within Win_{push_width} and Win_{pop_width} respectively, as well as that items are always pushed to at least row Win_{min}. We set Win_{pop_width} to the shared desired *width*, and set Win_{push_width} to the largest *width* of any *Lateral* node overlapping the new window.

The window shift updates the *widths*, reads Win_{depth} from a shared variable, and sets the Win_{max} accordingly. In addition, it stores the last *push_width* and the direction of the shift (UP or DOWN) in the window, which are used to stabilize the *Lateral*. Due to the extra variables, the shift linearizes with a 16-byte CAS. To support a computer without hardware support for such a wide CAS, one could use tagged pointers to 16 byte dynamically allocated windows.

The core of this LpW stack is how the *Lateral* is kept consistent over successive windows. This is done by maintaining the invariant that for any *Lateral* node l, all rows in the 2D stack between $l.row$ and $l.next.row$ will have smaller or equal width as $l.width$. This is enforced in two phases, which together create a local top candidate for the *Lateral* stack, and linearize with a CAS to update the top of the *Lateral*. Only one such update can linearize during each window.

The first phase *lowers Lateral* nodes above Win_{min}. If such a node l has $l.width \leq Win_{push\_width}$, then it is lowered to row Win_{min}, as newly pushed nodes could have invalidated its invariant. Similarly, if $l.width > Win_{last\_push\_width}$ and the last shift was downwards, l can safely be lowered to the previous Win_{min}. A node l is lowered by replacing it with a thread-local clone whose *row* is the lower row, and l is removed if its new *row* is smaller or equal to $l.next.row$.

In the second phase, a new *Lateral* node with $width = Win_{last\_push\_width}$ is pushed if $Win_{push\_width} \neq Win_{last\_push\_width}$. Depending on if Win_{push_width} has increased or decreased, the new top is pushed at row Win_{min} or above all populated rows in the 2D stack respectively. These phases result in a thread-local top *Lateral* node, which the thread atomically tries to swap with the previous *Lateral* top.

3.4 Elastic Extension Outlines: 2D Counter and Deque

The 2D counter can easily be made elastic by adding a *Lateral* counter. The key is to let the *Lateral* track the difference between the the sum of all counters within Win_{width} and the total count. A simple way to implement this is to add a small delay before changing Win_{width}, as in the LpW queue, and iterate over

the counters between the next and current Win_{width}, updating them and the *Lateral* to get a consistent offset.

To derive an elsatic 2D deque, we use a similar deque as [26], but with only one window at each side of the deque, much like the 2D queue. However, as with the 2D stack, these windows can shift both up and down. This can be made elastic in a similar fashion to the LaW queue, where a *Lateral* deque is kept with the sequence of all windows. If the windows shift with *depth* rows each time, as in the queue, a similar approach to the k-stack from [13] can be used to make sure a window is not removed while non-empty. If the window should shift by *depth*/2 as the stack, successive windows would overlap, requiring extra care. A solution to this could be to split each window in two, letting the threads operate on the two topmost windows, and still use the confirmation technique from [13] for the top *Win* when shifting downwards.

4 Correctness

Here we present the correctness and rank error bound guarantees of our elastic designs. Due to space constraints, we have moved the proofs to the extended version [8]. For simplicity, we only relax non-empty remove operations and assume a double-collect [23] approach is used to get linearizable empty returns, as is done in [10]. Furthermore, all our elastic designs have the same error bounds as the static 2D structures, $depth(width-1)$ for the queue [26] and $2.5\,depth(width-1)$ for the stack [8], if no elastic changes are used.

Theorem 1. *The elastic LaW queue is linearizeable with respect to a FIFO queue with elastic k out-of-order relaxed dequeues, where* $k = (Win_{width}^{head}-1)\,Win_{depth}^{head}$.

Theorem 2. *The elastic LpW queue is linearizeable with respect to a FIFO queue with elastic k out-of-order relaxed dequeues, where for every dequeue of item x,* $k = (Win_{width}^{enq\ x}-1)(Win_{depth}^{enq\ x}+Win_{depth}^{deq\ x}-1)$. *Here,* $Win^{enq\ x}$ *and* $Win^{deq\ x}$ *signify the windows during which x was enqueued and dequeued, respectively.*

Theorem 3. *The elastic LpW stack is linearizeable with respect to a stack with k out-of-order relaxed pops, where k is bounded for every item x as* $k = (Win_{width}^{max\ x}-1)(3\,Win_{depth}^{max\ x}-1)$. *Here,* $Win_{field}^{max\ x}$ *signifies the maximum value of* Win_{field} *during the lifetime of x on the stack.*

5 Evaluation

We experimentally evaluate the scalability and elastic capabilities of our elastically relaxed LaW queue, LpW queue, and LpW stack on a 128-core 2.25GHz AMD EPYC 9754 with two-way SMT, 256 MB L3 cache, and 755 GB RAM. The machine runs openSUSE Tumbleweed 20240303 which uses the 7.4.1 Linux kernel. All experiments are written in C and compiled with gcc 13.2.1 at its highest optimization level, using pthreads for parallelization. Threads are pinned in a round-robin fashion between core clusters, starting to use SMT after 128 threads.

Our elastic 2D implementations build on an optimized version of the static 2D framework [26]. We use SSMEM from [4] for memory management, which includes an epoch-based garbage collector for our dynamic memory.

5.1 Static Relaxation

To understand the performance of our data structures under static relaxation, we compare their scalability against that of state-of-the-art k out-of-order and strict concurrent designs. For the queues, we select the static 2D queue [26] and the k-segment queue [18] as k out-of-order designs. Furthermore, we selected the wait-free (WF) queue from [31] as the state-of-the-art linearizable FIFO queue, as well as the Michael-Scott (MS) queue [20] as a baseline. For the stacks, we selected the 2D stack [26] and the k-segment stack [13] as k out-of-order designs, the lock-free elimination-backoff stack from [12] as an efficient strict design, and the Treiber stack [28] as a baseline. All data structures were implemented in our framework using SSMEM [4] for memory management, with the exception of the WF queue, for which we used the authors' implementation with hazard pointers.

We use a benchmark in which threads repeatedly perform insert or remove operations at random, each with equal probability, for a duration of one second. Each data structure is pre-filled with 2^{19} items to avoid empty returns, which significantly alter the performance profile. Test results are aggregated over 10 runs, with standard deviation included in the plots. The test bounds the rank error of the data structures, and as the optimal choice of Win_{width} and Win_{depth} is not known [26], we set $Win_{width} = 2 \times$ `nbr_threads`, and use the maximum *depth* to stay within the bound, which is simple and gives acceptable scalability.

Measuring rank errors without altering their distribution is an open problem, and we adapt the method used in [26,3,25]. It encapsulates the linearization points of all methods by a global lock, enabling us to keep a totally ordered data structure to the side, protected by the lock. After each removal operation returning x, the distance between x and the top item gives the rank error.

Figure 3 shows how the queues and stacks scale with both threads and relaxation. The results show that the elastic designs scale essentially as well as the static 2D framework and out-scale the other data structures, meaning the *Lateral* induces minimal overhead. This is because the additional work from the *Lateral* is mostly confined to small checks during window shifts, while the algorithms otherwise function similarly to the static 2D structures.

5.2 Elastic Relaxation - Dynamic adjustments

In this section, we demonstrate the elastic capabilities of our designs by implementing a simple controller that dynamically adjusts the Win_{width} in a producer-consumer scenario with a varying workload. Consider a shared queue where tasks can be added and removed in FIFO order. We let one third (42) of the 128 cores act as consumers of this queue, constantly trying to dequeue tasks from it. The remaining (86) cores are designated as producers, repeatedly adding tasks to the queue, though not always active, as illustrated in the top graphs of Figure 4.

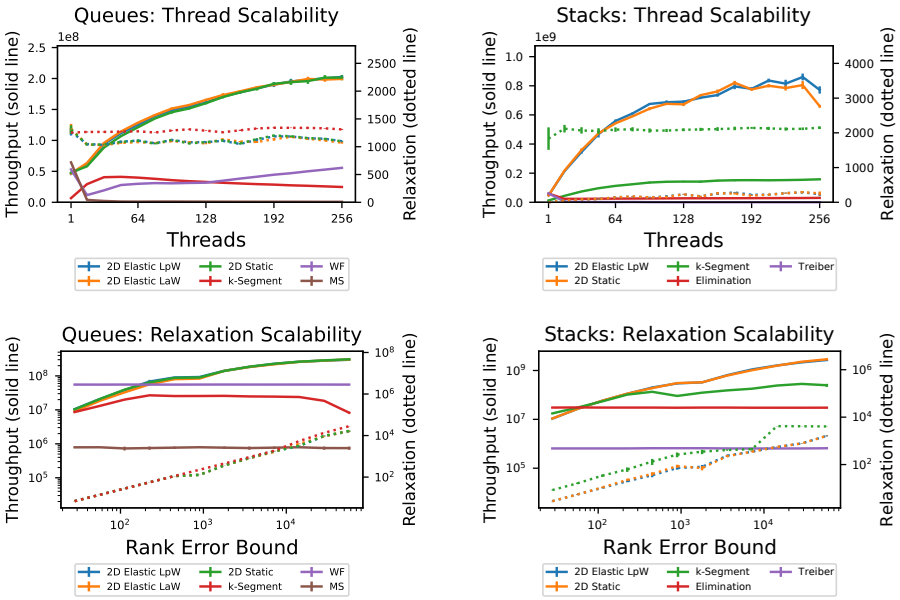


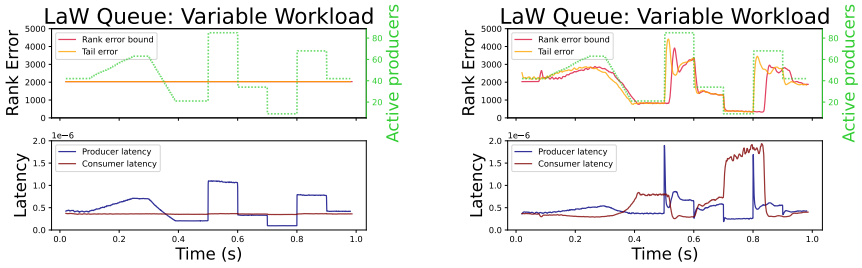
Fig. 3: Scalability of throughput and rank error during static relaxation. When scaling with threads (top row), the error bound is fixed as $k = 5 \times 10^3$. When scaling with error bound, 256 threads are used.

This simulates a task queue in a highly contended server, where the consumers are internal workers working at a constant pace, and the load of the producers vary depending on external factors. Our goal is to control the relaxation to cope with the dynamic nature of this producer workload.

To cope with this dynamic nature, our relaxation controller strives to keep the operational latency approximately constant for all producers. To minimize overhead, this controller is thread-local and only tracks failed and successful producer CAS linearizations. Its pseudocode is included in the extended paper [8], but is built around incrementing or decrementing a *contention* count by `SUCC_INC` or `FAIL_DEC` depending on if the CAS linearization on a sub-queue succeeds or fails. If $|contention| >$ `CONT_TRESHOLD`, the thread resets *contention*, adds a local vote for increasing or decreasing the next Win^{tail}_{width} by `WIDTH_DIFF`, and depending on its votes tries to change `width_shared`. When the window shifts, the local vote count is reset. These values can be tuned, but are in our experiments set as `SUCC_INC` $= 1$, `FAIL_DEC` $= 75$, `CONT_TRESHOLD` $= 5000$, `WIDTH_DIFF` $= 5$.

Figure 4 shows the average thread operational latency, as well as the error bound $(Win^{head}_{width} - 1) \times Win^{head}_{depth}$ averaged over 50 runs for our elastic LaW queue. It also shows $tail\ error = (Win^{tail}_{width} - 1) \times Win^{tail}_{depth}$ which can be interpreted as a rank error for enqueues, and shows that the controller adapts quickly. However, it notes occasional delays between Win^{tail}_{width} and Win^{head}_{width}, as the change has to

propagate through the queue. The figure shows a scenario without the dynamic controller, one short test over 1 second, and one long test over 1 minute.



(a) Static Relaxation (b) Dynamic Relaxation

Fig. 4: Producer-consumer system with a variable number of producers over time for the elastic LaW queue, running for one second. In the right plot, a relaxation controller is used to keep the producer's latency stable.

The test without the controller shows how the producers' latency clearly scales with their contention. However, using the controller, the producers' latency is much more stable, only temporarily spiking with increases in contention. While producers enjoy stable latency, consumers must accommodate the significant variations in relaxation. Furthermore, it is evident that the controller quickly adjusts Win^{tail}_{width}, but that it at some points takes a while for this change to propagate through the queue and reach Win^{head} and the rank error bound.

This experiment shows how even a simple thread-local controller for the *width* is enough to get good dynamic trade-offs between relaxation and latency. Similar, but a little more sophisticated, controllers could easily be created to target different use-cases, such as ones where we care about the performance of both the producers and consumers. For example, using the elastic LaW queue, a controller could control Win^{head}_{depth} and Win^{tail}_{depth} separately in combination with the Win_{width}, which would lead to more flexible adaptation. To fully leverage such a controller, it would be helpful to design a model for the queue performance, so that the choices of *depth* and *width* could be made with more information.

6 Conclusion

We have presented the concept of elastic relaxation for concurrent data structures and extended the 2D relaxed framework from [26] to encompass elasticity. The *Lateral* structure is used to track the history of elastic changes and can be used to extend other k out-of-order data structures. Our designs have established worst-case bounds, and demonstrate as good performance during periods of constant relaxation as state-of-the-art designs, while also being able to reconfigure their relaxation on the fly. Our simple controller, based on thread-local contention,

demonstrated that the elasticity can be utilized to effectively trade relaxation for latency. We believe elastic relaxation is essential for relaxed data structures to become realistically viable and see this paper as a first step in that direction.

As further work, we find constructing a model over the data structure performance interesting, which could aid in designing more sophisticated relaxation controllers. Another direction is applying the idea of elasticity to other data structures, such as relaxed priority queues.

Acknowledgment and Artifact Availability This work was supported by the Swedish Research Council project with Registration No.: 2021-05443. We would like to thank Adones Rukundo for sharing the code used in the evaluation part of [26]. Our code is available in the Zenodo repository [7].

References

1. Alistarh, D., Brown, T., Kopinsky, J., Li, J.Z., Nadiradze, G.: Distributionally Linearizable Data Structures. Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures pp. 133–142 (2018). https://doi.org/10.1145/3210377.3210411
2. Alistarh, D., Kopinsky, J., Li, J., Nadiradze, G.: The power of choice in priority scheduling. In: Proceedings of the ACM Symposium on Principles of Distributed Computing. p. 283–292. PODC '17, ACM (2017). https://doi.org/10.1145/3087801.3087810
3. Alistarh, D., Kopinsky, J., Li, J., Shavit, N.: The spraylist: A scalable relaxed priority queue. SIGPLAN Not. **50**(8), 11–20 (2015). https://doi.org/10.1145/2858788.2688523
4. David, T., Guerraoui, R., Trigonakis, V.: Asynchronized concurrency: The secret to scaling concurrent search data structures. SIGARCH Comput. Archit. News **43**(1), 631–644 (2015). https://doi.org/10.1145/2786763.2694359
5. Derrick, J., Dongol, B., Schellhorn, G., Tofan, B., Travkin, O., Wehrheim, H.: Quiescent consistency: Defining and verifying relaxed linearizability. In: FM 2014: Formal Methods. pp. 200–214. Springer (2014). https://doi.org/10.1007/978-3-319-06410-9_15
6. Ellen, F., Hendler, D., Shavit, N.: On the inherent sequentiality of concurrent objects. SIAM Journal on Computing **41**(3), 519–536 (2012). https://doi.org/10.1137/08072646X
7. von Geijer, K., Tsigas, P.: Artifact of the paper: How to Relax Instantly: Elastic Relaxation of Concurrent Data Structures (Jun 2024). https://doi.org/10.5281/zenodo.11547063
8. von Geijer, K., Tsigas, P.: How to Relax Instantly: Elastic Relaxation of Concurrent Data Structures (2024), https://arxiv.org/abs/2403.13644
9. Haas, A., Hütter, T., Kirsch, C.M., Lippautz, M., Preishuber, M., Sokolova, A.: Scal: A benchmarking suite for concurrent data structures. In: Networked Systems. pp. 1–14. Springer (2015). https://doi.org/10.1007/978-3-319-26850-7_1
10. Haas, A., Lippautz, M., Henzinger, T.A., Payer, H., Sokolova, A., Kirsch, C.M., Sezgin, A.: Distributed queues in shared memory: Multicore performance and scalability through quantitative relaxation. In: Proceedings of the ACM International Conference on Computing Frontiers. CF '13, ACM (2013). https://doi.org/10.1145/2482767.2482789

11. Hendler, D., Khattabi, A., Milani, A., Travers, C.: Upper and Lower Bounds for Deterministic Approximate Objects. 2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS) **00**, 438–448 (2021). https://doi.org/10.1109/icdcs51616.2021.00049
12. Hendler, D., Shavit, N., Yerushalmi, L.: A scalable lock-free stack algorithm. Journal of Parallel and Distributed Computing **70**(1), 1–12 (2010). https://doi.org/10.1016/j.jpdc.2009.08.011
13. Henzinger, T.A., Kirsch, C.M., Payer, H., Sezgin, A., Sokolova, A.: Quantitative relaxation of concurrent data structures. SIGPLAN Not. **48**(1), 317–328 (2013). https://doi.org/10.1145/2480359.2429109
14. Herlihy, M., Shavit, N., Luchangco, V., Spear, M.: The Art of Multiprocessor Programming. Elsevier Science (2020)
15. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems (TOPLAS) **12**(3), 463–492 (1990). https://doi.org/10.1145/78969.78972
16. Kappes, G., Anastasiadis, S.V.: A family of relaxed concurrent queues for low-latency operations and item transfers. ACM Trans. Parallel Comput. **9**(4) (2022). https://doi.org/10.1145/3565514
17. Karp, R.M., Zhang, Y.: Randomized parallel algorithms for backtrack search and branch-and-bound computation. J. ACM **40**(3), 765–789 (1993). https://doi.org/10.1145/174130.174145
18. Kirsch, C.M., Payer, H., Röck, H., Sokolova, A.: Performance, Scalability, and Semantics of Concurrent FIFO Queues. In: Algorithms and Architectures for Parallel Processing, vol. 7439, pp. 273–287. Springer Berlin Heidelberg (2012). https://doi.org/10.1007/978-3-642-33078-0_20
19. Kraska, T.: Towards instance-optimized data systems, keynote. 2021 International Conference on Very Large Data Bases (2021)
20. Michael, M.M., Scott, M.L.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing. pp. 267–275 (1996)
21. Morrison, A., Afek, Y.: Fast concurrent queues for x86 processors. In: Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. p. 103–112. PPoPP '13, ACM (2013). https://doi.org/10.1145/2442516.2442527
22. Nguyen, D., Lenharth, A., Pingali, K.: A lightweight infrastructure for graph analytics. In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. p. 456–471. SOSP '13, ACM (2013). https://doi.org/10.1145/2517349.2522739
23. Peterson, G.L., Burns, J.E.: Concurrent reading while writing II: The multi-writer case. In: 28th Annual Symposium on Foundations of Computer Science (Sfcs 1987). pp. 383–392. IEEE (1987). https://doi.org/10.1109/SFCS.1987.15
24. Postnikova, A., Koval, N., Nadiradze, G., Alistarh, D.: 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. p. 353–367. PPoPP '22, ACM (2022). https://doi.org/10.1145/3503221.3508432
25. Rihani, H., Sanders, P., Dementiev, R.: Multiqueues: Simple relaxed concurrent priority queues. In: Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures. p. 80–82. SPAA '15, ACM (2015). https://doi.org/10.1145/2755573.2755616

26. Rukundo, A., Atalar, A., Tsigas, P.: 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, pp. 31:1–31:15. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik (2019). https://doi.org/10.4230/LIPIcs.DISC.2019.31
27. Shavit, N.: Data structures in the multicore age. Commun. ACM **54**(3), 76–84 (2011). https://doi.org/10.1145/1897852.1897873
28. Treiber, R.: Systems programming: Coping with parallelism. Tech. rep., International Business Machines Incorporated, Thomas J. Watson Research Center (1986)
29. Williams, M., Sanders, P., Dementiev, R.: Engineering MultiQueues: Fast Relaxed Concurrent Priority Queues. In: 29th Annual European Symposium on Algorithms (ESA 2021). Leibniz International Proceedings in Informatics (LIPIcs), vol. 204, pp. 81:1–81:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik (2021). https://doi.org/10.4230/LIPIcs.ESA.2021.81
30. Wimmer, M., Gruber, J., Träff, J.L., Tsigas, P.: The lock-free k-lsm relaxed priority queue. In: Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. p. 277–278. PPoPP 2015, ACM (2015). https://doi.org/10.1145/2688500.2688547
31. Yang, C., Mellor-Crummey, J.: 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. PPoPP '16, ACM (2016). https://doi.org/10.1145/2851141.2851168