Elastic Relaxation of Concurrent Data Structures

Kåre von Geijer, Philippas Tsigas

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, stacks, a counter, and a deque, capable of reconfiguring relaxation during run-time. We establish linearizability and define worst-case bounds for relaxation errors in our designs. Experimental evaluations show that our elastic designs match the performance of state-of-the-art statically relaxed structures when no elastic changes are utilized. We develop a lightweight, contention-aware controller for adjusting relaxation in real time, and demonstrate its benefits both in a dynamic producer-consumer micro-benchmark and in a parallel BFS traversal, where it improves throughput and work-efficiency compared to static designs.

Index Terms—Concurrent Programming, Data Structures, 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 [3], but theoretical results have demonstrated that many common data structures, such as queues, have inherent scalability limitations [4] as threads must contend for few access points. One of the most promising solutions to tackle this scalability issue is to relax the sequential specification of data structures [5], 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 by Henzinger et al. [6] is a popular relaxation type [7]–[10] 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 FIFO dequeue, is called the *rank error*.

While other relaxations, such as quiescent consistency [11] are incompatible with linearizability [12], k out-of-order relaxation can easily be combined with linearizability, as it modifies the semantics of the data structure rather than the consistency. Despite extensive work on out-of-order relaxation [6]–[8], [10], [13]–[17], almost all existing methods are static, requiring a fixed relaxation degree during the data structure's lifetime.

In applications with dynamic workloads, such as bursts of activity with latency constraints, it is essential to be able to temporarily sacrifice sequential semantics for improved performance. This paper addresses the problem of specifying and designing data structures whose degree of relaxation can be adjusted dynamically at run-time—a concept we term *elastic relaxation*. Elastically relaxed data structures enable the design of instance-optimizing systems, an area that is evolving rapidly across various communities [18]. The trade-off between rank error and throughput is well demonstrated by Williams et al. [14] and Postnikova et al. [15], whose shortest-path benchmarks show that increased relaxation improves throughput at the expense of work-efficiency. These relaxed designs have outperformed state-of-the-art static approaches in parallel SSSP on sparse high-diameter graphs [19], highlighting the potential of further exploring the field of relaxed data structures.

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 [7], [8], [14], [17], [20]. In this paper, we base our elastic designs on the relaxed 2D framework [7], 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 in any order. The Win^{enq} shifts upward by *depth* when full, and Win^{deq} similarly shifts upward when emptied, to allow further operations. The size of the window dictates the rank error, as a larger *Win* allows for more reorderings.

The algorithmic design concept we propose in this paper is the *Lateral* structure, which we use to extend the 2D framework to encompass elastic relaxation. This *Lateral* is a strict concurrent version of the relaxed data structure—so

This work was supported by the Swedish Research Council project with Registration No.: 2021-05443. A preliminary version of this work was published in Euro-Par 2024 [1]. Code for replicating all experiments in this paper is available in the Zenodo repository [2].

K. von Geijer is with the Graduate School of Computer Science and Engineering, Chalmers University of Technology and Gothenburg University, Gothenburg, Sweden.
P. Tsigas is with the Faculty of Computer Science and Engineering, Chalmers University of Technology and Gothenburg University, Gothenburg, Sweden.

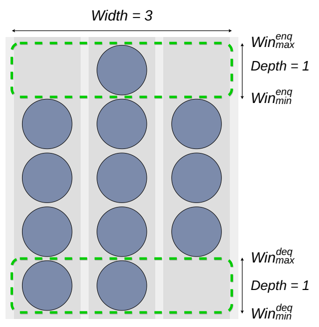


Figure 1: The 2D queue has two windows defining the operable area for the enqueue and dequeue operations.

for a relaxed queue, it is a strict queue—maintained alongside the substructures to track elastic changes. The *Lateral* is visualized for one of our elastic queues in Figure 2. We show two generic designs for incorporating the *Lateral* into the window mechanism of the 2D framework while achieving deterministic rank error bounds, one of which is simple, and one more complicated with better elastic capabilities. Although using the 2D framework as the base for our designs, the *Lateral* can also accommodate other designs, such as the distributed queues by Haas et al. [8], the k-queue [9], and the k-stack [6].

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.

- We introduce the concept of *elastic relaxation*, allowing rank errors to change over time. Furthermore, we introduce the *Lateral* component for efficiently enhancing relaxed data structures with elasticity.
- We elastically extend the 2D framework, deriving elastic queues, stacks, a counter, and a deque using the *Lateral*, and establish both correctness and rank error bounds. Our *Lateral*-based designs are grouped into two paradigms: Lateral-plus-Window (LpW), which can be applied to other relaxed structures with minimal modification, and Lateral-as-Window (LaW), which are simpler, but specially tailored for the 2D framework.
- We evaluate the scalability of our proposed data structures 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 best statically relaxed ones, while simultaneously supporting elastic relaxation.
- Finally, we demonstrate the elastic capabilities of our designs by implementing a lightweight, contention-aware controller for adjusting relaxation. We evaluate it in two dynamic settings: a producer-consumer benchmark, where it adapts relaxation over time, and a BFS traversal, where it improves both runtime and work-efficiency compared to static configurations.

Structure. Section 2 presents the preliminaries, focusing on the 2D framework. 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. Section 6 discusses related work. Section 7 concludes the paper.

2 Preliminaries

An *out-of-order* relaxed data structure relaxes the sequential specification of the underlying data structure to allow operations to linearize [12] out of order. The concept was formalized by Henzinger et al. [6] in their theoretical framework *quantitative relaxation*, which defines *relaxation errors* based on transition costs in the linearized history. In the case of a FIFO queue, if the third item is dequeued, the out-of-order error would be 2, as one would need to remove the enqueue operations of the two first items for the operation to be correct. Recent relaxed queue designs [14], [17] distinguish *rank errors* and *delay errors*, where rank errors are the errors described above, and the *delay* when dequeuing an item is the number of earlier dequeues that returned items of lower rank; items which were enqueued later in a FIFO queue.

The algorithm descriptions in this paper assume a sequentially consistent [21] programming model for simplicity. However, efficient implementations, such as ours for the evaluation [2], should use less restrictive memory orderings where possible. The algorithms also utilize the atomic *compare-and-swap* instruction, where `CAS(loc, exp, new)` atomically sets the memory at `loc` equal to `new` if the previous value at `loc` equals `exp`.

2.1 Static 2D Framework

The 2D framework for k out-of-order relaxed data structures [7] (hereby called the *static* 2D framework) unifies designs across FIFO queues, stacks, deques, and counters. Furthermore, its implementations outscale other relaxed implementations from the literature with deterministic error bounds [6], [8], [9] and its throughput scales monotonically with k. It achieves its good performance by distributing operations across disjoint sub-structures – reducing contention – as well as having threads return to the same sub-structure for several successive operations – increasing data locality.

The 2D framework can be seen as a two-dimensional grid, where the columns are concurrent sub-structures, and the rows are indexes within those sub-structure. In this view, a normal (strict) queue would be a single column, where one inserts items at rows $0, 1, 2\ldots$, and removes them in the same order. The core of the 2D framework is to superimpose a 2D *window* (*Win*) over the grid, and only allow operations at rows and columns within *Win*. An example 2D queue state is shown in Figure 1. If there is no valid operation in *Win*, a thread *shifts Win* up or down to allow further operations. For conciseness, we call the number of substructures or columns the *width* (Win_{width}), and the number of rows spanned by each window the *depth* (Win_{depth}).

Algorithm 1 shows the core code for the 2D queue, which is almost identical to the other data structures. For each operation, the thread iterates over the columns and tries to linearize on a row within *Win*, otherwise shifting *Win* before trying again. The dequeue deviates from this, in that it returns `EMPTY` if a linearizable double-collect scan [22] validates that all sub-queues are empty. The static *Win* mainly includes a *max* (Win_{max}) field ($Win_{min} := Win_{max} - depth$), and *Win* can thus be shifted with a single CAS that increases or decreases Win_{max}. An important detail is that the iterations at lines 1.5 and 1.13 should start at the column where the thread last linearized, for better cache performance, similarly to *stickiness* in later related work [14].

Algorithm 1: Core of the Static 2D Queue

```
1.1  global Window enqWin;
1.2  global Window deqWin;
1.3  function Enqueue(item)
1.4      win ← enqWin;
1.5      foreach sub ∈ sub_structures do        ▷ Start at last visited
1.6          while win.min() ≤ (row ← sub.enq_row) < win.max do
1.7              if win ≠ enqWin then goto line 1.4;
1.8              if sub.TryEnqueue(item, row) then return;
1.9      ShiftEnq(win);                   ▷ Increment enqWin.max with CAS
1.10     goto line 1.4;
1.11 function Dequeue()
1.12     win ← deqWin;
1.13     foreach sub ∈ sub_structures do        ▷ Start at last visited
1.14         while win.min() ≤ (row ← sub.deq_row) < win.max do
1.15             if win ≠ deqWin then goto line 1.12;
1.16             ok, item ← sub.TryDequeue(row);
1.17             if item = EMPTY then break else if ok then return item;
1.18     if DoubleCollectAllEmpty(sub_queues) then return EMPTY;
1.19     if sub.deq_row = win.max ∀ sub ∈ sub_structure then ShiftDeq(win);
1.20     goto line 1.12;
```

Looking closer at the 2D queue, both its windows (Win^{enq} and Win^{deq}) only contain a Win_{max} field, and both *ShiftEnq* and *ShiftDeq* simply increment Win_{max} by *depth* with a CAS. As in all 2D designs, if another thread shifts *Win* before you, your shift aborts after the failed CAS to not shift *Win* twice. Using the state in Figure 1 as an example, Win^{enq} spans row 4 where columns 0 and 2 are valid for enqueues, while column 1 is already enqueued into. If column 0 and 2 had been enqueued into, the enqueue operation would atomically shift Win^{enq}_{max} up by *depth*, before again trying to find a valid column to enqueue at. The sub-queues in the 2D queue are implemented as Michael-Scott (MS) queues [23] with counted head and tail pointers. This choice makes it easy to implement *TryEnqueue* (line 1.8) and *TryDequeue* (line 1.16), which only succeed if able to linearize at the correct row. Furthermore, the counted head and tail pointers are used to determine the row of the queue's head or tail (lines 1.6 and 1.14). The usage of such counted pointers has been used in similar relaxed queues [8], and they are simple to implement with the 16-byte CAS on x86 [24]. If not having access to a 16-byte CAS, one can instead include the row count within each node, or use an array-based sub-queue which maps row indexes to array slots [25]–[27]

The 2D stack is only slightly different from the queue in Algorithm 1. Firstly, only one *Win* is required, as both pushes and pops operate on the same side of the data structure. This leads to Win_{max} no longer increasing monotonically, but instead decreasing under pop-heavy workloads, and increasing under push-heavy ones. Furthermore, Win_{max} shifts by $Win_{depth}/2$ instead of Win_{depth}, which roughly leaves *Win* half-full after each shift. As Win_{max} does not strictly monotonically increase, *Win* also includes a version count to circumvent the ABA problem. The sub-stacks are implemented as Treiber stacks [28], again using counted pointers to the top item, to track its row and facilitate easy *TryPush* and *TryPop* methods.

The 2D deque can be derived as a combination of the queue and stack, where it has one *Win* at each end of the data structure, as the 2D queue, and these *Win* can shift both up and down (by $depth/2$), as the 2D stack. The simplest way to implement the sub-deques is using the Maged deque [29] with counted top and bottom pointers.

Finally, the 2D counter can be seen as a special case of the 2D stack which disregards the items, and only tracks the sub-structure sizes. Therefore, it only needs a single *Win*, and updates its sub-counters in the same way the 2D stack would increment or decrement the top row of each sub-stack. The size of the counter is approximated by the row of one sub-counter multiplied by Win_{width}.

In summary, the 2D designs can be modeled as two-dimensional grids, where items are inserted or deleted from valid rows within the current *Win*. The hard bound imposed by the *Win* makes it possible to give worst-case guarantees on the rank errors of the operations. For example, for the 2D queue, operations that linearize during different *Win* are totally ordered by proxy of the *Win* order, and each sub-queue is totally ordered as it is a strict MS queue. Therefore, it is simple to see that rank errors are bounded by $(Win_{width} - 1)Win_{depth}$ [7]. Similar bounds hold for the other designs [7], but are slightly harder to derive [30].

3 Design of Elastic Algorithms

This section introduces several algorithms with *elastic* out-of-order relaxation, by extending the static 2D framework [7]. The two dimensions of relaxation in the 2D framework are Win_{width} and Win_{depth}, whose optimal configurations depend on both the number of threads and the surrounding use-case. In general, Win_{width} is often set proportional to the number of threads [7], [14], while Win_{depth} is determined by algorithm-specific factors. As Win_{width} and Win_{depth} serve different uses, it is essential that both of them can be reconfigured during runtime for an elastic extension of the 2D framework to be practically useful.

Our new elastic designs therefore let Win_{width} and Win_{depth} change with every window shift. In the pseudocode, we use *width_desired()* and *depth_desired()* as hooks representing the currently desired values, which can be controlled either manually or by a dynamic controller. Changing Win_{depth} is straightforward in practice, although it affects the error bounds. Efficiently varying Win_{width} demands more attention, as it involves modifying the number of active sub-structures while maintaining error bounds. To keep this lightweight, our approach is to leave existing items in place and apply the new Win_{width} only to future insertions.

We track the elastic changes to Win_{with} and Win_{depth} using a *Lateral* structure. This *Lateral* consists of a set of nodes, each corresponding to a range of rows in the substructures, and provides a width bound for those rows. The *Lateral width bound* of a row is the largest width bound of any *Lateral* node spanning that row. This approach of using the *Lateral* is visualized in Figure 2 for the queue. There, items were initially enqueued with $Win^{enq}_{width} = 3$, but then it changed to $Win^{enq}_{width} = 4$ at the second row, and then again to $Win^{enq}_{width} = 2$ at the fourth row. These two changes can be seen in the *Lateral*, which has a *Lateral* node for each, storing the new width and row where the change took effect.

Each *Lateral* node is enqueued at a row, and implicitly spans all rows until the next *Lateral* node. Notably, there are now empty slots in the sub-structures, such as the last column on the first row, where no item will be inserted. However, the rows are purely logical, in the implementations only corresponding to counters, so these empty rows do not consume memory. Importantly, Win^{enq} is allowed to adjust its dimensions freely, while Win^{deq} must adapt its *width* to these changes by inspecting in which columns items may have been enqueued, as indicated by the *Lateral*.

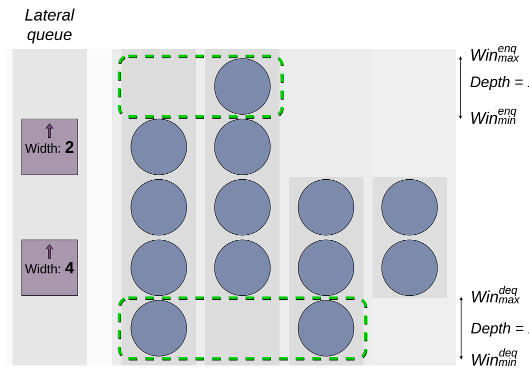


Figure 2: By adding a *Lateral* to the 2D queue, changes in width at Win^{enq} can be tracked and adjusted to by Win^{deq}. This depicts a possible state in the elastic LpW queue.

Our elastic queue designs, presented in Section 3.1 and 3.2, are relatively simple, as only a single Win^{enq} spans each row. In contrast, the remaining designs—such as the stacks in Section 3.3 and 3.4—are more complex. There, the width of a row can change multiple times during execution, requiring the *Lateral* to be updated continuously to stay correct. To denote this correctness, we call the *Lateral consistent* if the *width bound* for each row is greater than or equal to the maximum column index of all items currently in that row.

We identify two design paradigms for elastically extending the 2D data structures using the *Lateral*. The Lateral-plus-Window (LpW) designs add the *Lateral* alongside the existing static components and modify only the logic for shifting the windows. In contrast, the Lateral-as-Window (LaW) designs merge the *Lateral* and *Win* into a single linked-list structure of windows. LaW designs are simpler to implement and analyze than LpW designs, but they are more specific to the 2D framework and slightly less flexible in supporting certain elastic behaviors. The difference between the LpW and LaW designs can be seen when comparing the LpW queue in Figure 2 and LaW queue in Figure 3, where the LpW design only adds a *Lateral* node when Win^{enq}_{width} changes, and the LaW design lets each *Win* be a *Lateral* node – enqueuiing one each window shift.

In the following sections, we present algorithms for both LpW and LaW queues and stacks, as well as a LaW deque and an LpW counter. These algorithms focus only on the novel modifications to the *Win* structure and its shifting logic; all other behavior follows the static procedures of Algorithm 1, except that operations now traverse only the substructures within the current Win_{width}.

3.1 Elastic Lateral-plus-Window 2D Queue

The elastic Lateral-plus-Window (LpW) queue builds on the static 2D queue from Algorithm 1, with the key change that operations iterate only over columns within the current Win_{width}. Algorithm 2 presents the extensions to the static version, including the integration of the *Lateral* structure and. Elasticity is achieved by allowing Win^{enq}_{depth}, Win^{enq}_{width}, and Win^{deq}_{depth} to change mostly freely at each *Win* shift. When Win^{enq}_{width} changes, a new node is added to the *Lateral*, recording the new width for future dequeuers. This ensures that Win^{deq}_{width} always covers at least the rightmost populated column between Win^{deq}_{min} and Win^{deq}_{max}.

A possible state of the LpW queue is shown in Figure 2. As there are two *Lateral* nodes, there has been two changes in Win^{enq}_{width}, going from the current $Win^{deq}_{width} = 3$ at the bottom, to $Win_{width} = 4$ at the second row, and finally to $Win_{width} = 2$ at the fourth row. Changes in Win_{depth} are not tracked in the *Lateral*. The core idea is that Win^{enq}_{width} changes freely, but must *first* enqueue a *Lateral* node with the width change, and that the Win^{deq}_{width} adapts based on the *Lateral*.

The LpW queue expands the fields within the windows compared to the static algorithm, which only tracks Win_{max}. Firstly, Win^{enq}_{width} and Win^{deq}_{width} are added so that operations know which sub-queues to iterate over. Additionally, $Win^{enq}_{next\_width}$ is added, which holds the upcoming Win^{enq}_{width}. The delay of one Win^{enq} shift before applying this new width is included to keep the *Lateral* consistent. Both the sub-queues and *Lateral* are implemented as MS queues [23], and provide *TryEnqueue* and *TryDequeue* methods, which succeed only if that end of the queue matches the argument.

Shifting Win^{enq} is done by creating a new Win^{enq} with a desired Win^{enq}_{depth}, Win^{enq}_{width} equal to the last $Win^{enq}_{next\_width}$, and $Win^{enq}_{next\_width}$ as desired, finally updating the global Win^{enq} with a CAS (line 2.25). However, before this is done, if $Win^{enq}_{width} \neq Win^{enq}_{next\_width}$ (line 2.19), the algorithm ensures a *Lateral* node with $width = Win^{enq}_{next\_width}$ is enqueued into the *Lateral* first. Enqueuing this node into the *Lateral* before setting the new Win^{enq}_{width} into effect ensures that the *Lateral* is always consistent, so that Win^{deq}_{width} will always cover all populated columns between Win^{deq}_{max} and Win^{deq}_{min}.

Conversely, when shifting Win^{deq} (line 2.37), the algorithm first dequeues all *Lateral* nodes below Win^{deq}_{max} (line 2.38), as they represent changes in *width* that have already been logically used. If the bottom of the new Win^{deq} is the same row as the bottommost *Lateral* node, the new Win^{deq}_{width} is set to that node's *width* (line 2.43); otherwise, it remains the same as the previous Win^{deq}_{width} (line 2.41). The new Win^{deq}_{max} is then set by incrementing the old Win^{deq}_{max} by the desired *depth*, while ensuring it does not surpass Win^{enq}_{max} or the row of the *Lateral* tail (line 2.45). This restriction exists to ensure that Win^{deq} overlaps with at most one *Lateral* node, which occurs when that node has $row = Win^{deq}_{min} + 1$. This maintains the invariant that all rows in a Win^{deq} have equal widths.

Having covered how Win^{enq}, Win^{deq}, and the *Lateral* synchronize, the detail of which row to enqueue items on remains. In the static 2D queue, items are always enqueued immediately above the last item. But as seen at the bottom right in Figure 2, our elastic queues can have gaps in the

Algorithm 2: Elastic *Win* and *Lateral* in the LpW queue

```
2.1  struct EnqueueWindow
2.2      uint max;
2.3      uint width;
2.4      uint next_width;
2.5  struct DequeueWindow
2.6      uint max;
2.7      uint width;
2.8  struct LateralNode
2.9      LateralNode* next;
2.10     uint row;
2.11     uint width;
2.12 struct Lateral
2.13     LateralNode* head;
2.14     LateralNode* tail;
2.15 global EnqueueWindow* enqWin;
2.16 global DequeueWindow* deqWin;
2.17 global Lateral lat;
2.18 function ShiftEnq(old_win)
2.19     if $old\_win.width \neq old\_win.next\_width$ then
             lat.SyncTail(old_win) ;
2.20     new_win $\leftarrow$ EnqueueWindow {
2.21         width: old_win.next_width,
2.22         next_width: width_desired(),
2.23         max: old_win.max + depth_desired()
2.24     };
2.25     CAS(&enqWin, old_win, new_win);
2.26 method Lateral.SyncTail(window)
2.27     tail $\leftarrow$ lat.tail;
2.28     if $tail.row \leq window.max$ then
2.29         new_tail $\leftarrow$ LateralNode {
2.30             row: window.max + 1,
2.31             width: window.next_width
2.32         };
2.33         lat.TryEnqueue(new_tail, tail);
2.34 method Lateral.TryEnqueue(new, old);
2.35 method Lateral.TryDequeue(old);
2.36 method Lateral.Peek();
2.37 function ShiftDeq(old_win)
2.38     while $(head \leftarrow lat.Peek()).row \leq old\_win.max$ do
2.39         lat.TryDequeue(head);                 ▷Cleanup
2.40     if $head.row > old\_win.max + 1$ then
2.41         width $\leftarrow$ old_win.width;
2.42     else
2.43         width $\leftarrow$ head.width;
2.44         head $\leftarrow$ head.next;
2.45     max $\leftarrow$ Min(enqWin.max, head.row - 1,
             old_win.max + depth_desired()) ;
2.46     new_win $\leftarrow$ DequeueWindow {
2.47         width: width,
2.48         max: max
2.49     };
2.50     CAS(&deqWin, old_win, new_win);
```

sub-queues following increases in Win^{enq}_{width}. Therefore, items are instead enqueued at the larger of the row above the last item and $Win^{enq}_{min} + 1$. This can be implemented by storing row information within each node in the sub-queues, but this pointer-indirection causes a slight overhead, and one can instead either include Win^{enq}_{depth} in Win^{enq}, or use the row of the *Lateral*'s tail to calculate the row.

3.2 Elastic Lateral-as-Window 2D Queue

This elastic LaW queue merges the *Win* and *Lateral* structures into a Michael-Scott (MS) queue [23] of windows. Algorithm 3 presents the extensions relative to the static queue in Algorithm 1. It again uses conditional *TryEnqueue* and *TryDequeue* methods, which only linearize if the tail or head matches the expected value. Each *Win* in the *Lateral* contains *max*, *depth*, and *width*. The global Win^{enq} and Win^{deq} become the head and tail nodes in the *Lateral*. Every shift of Win^{enq} enqueues a new *Win* into the *Lateral* (line 3.12), and each shift of Win^{deq} dequeues a *Win* (line 3.20).

Algorithm 3: The *Lateral* in the elastic LaW queue

```
3.1  struct Window
3.2      Window* next;
3.3      uint max;
3.4      uint depth;
3.5      uint width;
3.6  struct Lateral
3.7      Window* head;
3.8      Window* tail;
3.9  global Lateral lat;
3.10 method Lateral.TryEnqueue(old, new);
3.11 method Lateral.TryDequeue(old);
3.12 function ShiftEnq(old_win)
3.13     depth $\leftarrow$ depth_desired();
3.14     new_win $\leftarrow$ Window {
3.15         width: width_desired(),
3.16         depth: depth,
3.17         max: oldWin.max + depth
3.18     };
3.19     lat.TryEnqueue(old_win,
             new_win);
3.20 function ShiftDeq(old_win)
3.21     lat.TryDequeue(old_win);
```

An example state of the LaW queue is shown in Figure 3, and can be compared to the LpW queue in Figure 2, which may result from the same sequence of operations. In the LpW queue, only Win^{enq} and Win^{deq} are stored, and the *Lateral* contains only *changes* in Win^{enq}_{width}. In contrast, the LaW queue stores information about *every Win* between Win^{enq} and Win^{deq}. This results in increased memory usage, but the overhead is typically modest, as each *Win* is just a node in a linked list that often spans many items. The benefit of keeping all *Win* instances in the *Lateral* is that it unifies

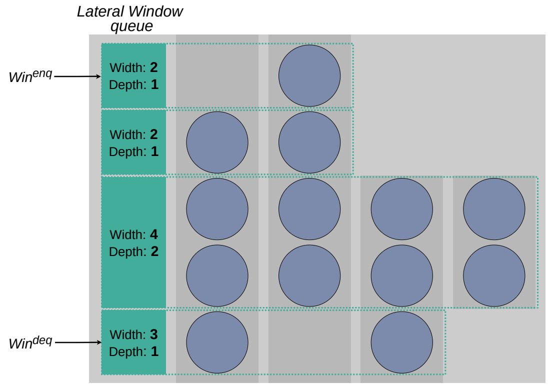


Figure 3: A possible state in the Elastic LaW queue, where windows are stored in the *Lateral*, such that Win^{enq} is the tail and Win^{deq} is the head. Looking at the *Lateral*, the queue must have initially been configured as $Win^{enq}_{width} = 3, Win^{enq}_{depth} = 1$, but then changed to $Win^{enq}_{width} = 4, Win^{enq}_{depth} = 2$, and finally $Win^{enq}_{width} = 2, Win^{enq}_{depth} = 1$.

the *Lateral* and *Win* designs, making it easier to change relaxation and track its changes over time.

As shown in *ShiftEnq* (line 3.12), Win_{depth} and Win_{width} can be updated from arbitrary variables at each shift, enabling elastic behavior. The main drawback of this design is that the relaxation can only change at Win^{enq}, and must propagate through the queue to eventually reach Win^{deq}.

Finally, as in the LpW queue, the LaW queue only enqueues items within the current Win^{enq}. After selecting a sub-queue, each item is enqueued at row $max(Win^{enq}_{max} - Win^{enq}_{depth}, sub\_queue.tail\_row) + 1$, which may create gaps in the sub-queues, as shown in Figure 3.

3.3 Elastic Lateral-as-Window 2D Stack

The elastic LaW stack retains as much simplicity from the LaW queue as possible, while handling the bidirectional window shifts of the 2D stack. As the LaW queue in Figure 3, the LaW stack merges the *Win* with the *Lateral*, keeping a Treiber stack [28] of windows. Pushes and pops work similarly to the static 2D stack, where the thread finds a sub-stack to operate on within the current *Win* (the top *Win* in the *Lateral*), and then linearizes with a CAS. Additionally, sub-stack nodes store the row of the node below, and items are always inserted above Win_{min} as in the elastic queues.

The main complication of elastically extending the static 2D stack is that its *Win* can shift downward at any point after all sub-queues inside Win_{width} have been seen at Win_{min}. This does not guarantee that all sub-stacks remain at Win_{min} at the linearization of the shift. Therefore, a naive elastic design could shift downward from Win^A to Win^B and leave items above Win^B_{max}, and outside Win^B_{width}, which would lead to correctness issues.

The LaW stack is designed around the simplifying constraint that items are always pushed within, and remain within, a *Win* on the *Lateral*, as formalized by Property 3.1. This property simplifies elastic extension and also leads to a tighter and simpler rank error bound than the Static 2D stack (as presented and discussed in [30]).

Property 3.1 (LaW Stack window cover)**.** *If item x is pushed to sub-stack col_x at row row_x, and has not been popped, then $\exists Win^x \in$ Lateral such that $col_x < Win^x_{width}$ and $Win^x_{min} < row_x \leq Win^x_{max}$.*

Algorithm 4: Elastic 2D LaW Stack

```
4.1  function ShiftUp(old_win)
4.2      if old_win.shrinking then
4.3          new_win ← old_win.Clone();
4.4          new_win.shrinking ← false;
4.5      else
4.6          depth ← depth_desired();
4.7          new_win ← {
4.8              max: old_win.max + depth/2,
4.9              depth: depth,
4.10             width: width_desired(),
4.11             shrinking: false,
4.12             next: old_win
4.13         };
4.14     CAS(&win, old_win, new_win);

4.15 struct Window
4.16     Window* next;
4.17     uint max;
4.18     uint depth;
4.19     uint width;
4.20     bool shrinking;
     // Top of the Lateral stack
4.21 global Window win;

4.22 function ShiftDown(old_win)
4.23     if !old_win.shrinking then
4.24         new_win ← old_win.Clone();
4.25         new_win.shrinking ← true;
4.26     else
4.27         assert(∀sub ∈ sub_stacks[old_win.next.width... old_win.width]:
                 sub.row < old_win.max − old_win.depth and sub.frozen);
4.28         assert(∀sub ∈ sub_stacks[0... old_win.width]:
                 sub.row ≤ old_win.next.max);
4.29         new_win ← old_win.next.Clone();
4.30     CAS(&win, old_win, new_win);
```

Algorithm 4 shows how to shift the *Win* for the LaW stack in a way to uphold Property 3.1. A downward *Win* shift is now split in two steps: shrinking *Win* (line 4.25), and in a later shift invocation, shifting *Win* down (line 4.29). When $Win_{shrinking}$ is set, push operations are only allowed to push within the intersection of *Win* and Win^{next}. Similarly, upward shifts either unshrink the *Win*, or if if it is not shrinking, push a new *Win* to the *Lateral*.

Furthermore, a *frozen* bit is incorporated into the counted top pointer of each sub-stack. If this bit is set, a push is not allowed to linearize on the sub-stack. Pop operations must have seen this bit set on all sub-stacks outside Win^{next}_{width}, together with them being at or below Win_{min}, before shifting down (assert at line 4.27). As a sub-stack push can only linearize by first reading the descriptor, then validating it is valid under the global *Win*, and finally linearizing with a CAS, setting these *frozen* bits ensure that no item can be pushed outside the current *Win*, ensuring Property 3.1.

Shifting the *Win* upward is a similar process, but without the possibility of leaving items outside the *Lateral* (as it only expands it). Firstly, if $Win_{shrinking}$ is set, we clear the bit and undo the shrinking with a CAS (line 4.4). Otherwise, a new *Win* is simply created with the desired Win_{width} and Win_{depth}, and pushed to the *Lateral* (line 4.7). Finally, when push operations encounter a frozen sub-stack valid for a push, it first clears the *frozen* bit with a CAS, and then re-reads the sub-stack and *Win* before proceeding with the push.

3.4 Elastic Lateral-plus-Window 2D Stack

The elastic LpW stack is similar in design and motivation to the LpW queue. It can change relaxation behavior more flexibly than the LaW stack, and slightly improves performance by not having to shrink the *Win* at shifts. Instead of storing all windows in the *Lateral*, as the LaW stack, it only stores *Lateral* nodes at rows where the width changes, similarly to the LpW queue in Figure 2. As this algorithm can shift the *Win* more freely than the LaW stack, it requires more care to keep the *Lateral* consistent, which is its main drawback.

Algorithm 5 shows how to shift the *Win* and synchronize the *Lateral* in the LpW stack. It again implements the *Lateral* as a Treiber stack [28], but now only pushes nodes to it when the *width* changes. However, as the *width* of a row can change many times in a 2D stack, due to its non-monotonic Win_{max}, the *Lateral* nodes must be re-adjusted before every *Win* shift (line 5.14). There, it possibly replaces several *Lateral* nodes with a single CAS (line 5.73. Like the LaW stack, it always pushed items above Win_{min}, and stores the row count of items in their sub-stack nodes.

To increase the elasticity from the LaW stack, the LpW stack separates Win_{width} into Win_{push_width} and Win_{pop_width} for the push and pop operations. Then, Win_{push_width} can be updated at will (line 5.16) and Win_{pop_width} is set as the upper bound on the *width* of rows in the *Win*, which can be calculated using the *Lateral* (line 5.26).

Shifting the *Win* (line 5.13) is done by first ensuring the *Lateral* is stabilized (line 5.14), then creating a new *Win*, including Win_{push_width} and Win_{pop_width} as described above, finally linearizing with a CAS (line 5.27). Furthermore, the *Win* stores the direction of the shift (`UP` or `DOWN`), and the last Win_{push_width}, which are used to keep the *Lateral* consisent at the next shift.

The function *StabilizeLateral* is the core of the LpW stack (line 5.14), keeping the *Lateral* consistent at every *Win* shift. The function maintains the stack invariant that for any *Lateral* node l, all sub-stack items between $l.row$ and $l.next.row$ will be at columns less than $l.width$. We divide the synchronization into two phases, which together create a local top candidate for the *Lateral* stack, and linearize with a CAS at line 5.73.

The first phase clones and tries to *lower Lateral* nodes above Win_{min} (line 5.54). By lowering a *Lateral* node l to row r, we set $l.row \leftarrow min(l.row, r)$, and then if $l.row \leq l.next.row$, l is removed from the stack (by re-linking its

Algorithm 5: *Lateral* and *Win* logic for the elastic LpW stack

```
5.1  struct Window
5.2      uint max;
5.3      uint depth;
5.4      uint push_width;
5.5      uint pop_width;
5.6      uint last_push_width;
5.7      enum last_shift;                          ▷ UP or DOWN
5.8      uint version;

5.9  method Window.Min(self)
5.10     return self.max - self.depth;

5.11 global Window* win;
5.12 global Lateral* lat;

5.13 function Shift(old_win, dir)
5.14     StabilizeLateral(lat, old_win);
5.15     new_win ← Window {
5.16         push_width: width_desired(),
5.17         last_push_width: old_win.push_width,
5.18         depth: depth_desired(),
5.19         last_shift: dir,
5.20         version: old_win.version + 1
5.21     };
5.22     if $dir = UP$ then
5.23         new_win.max ← old_win.max +
               new_win.depth/2;
5.24     else
5.25         new_win.max ← old_win.Min() +
               new_win.depth/2;

5.26     new_win.pop_width ← max(new_win.push_width,
           lat.Width(new_win.Min()));
5.27     CAS(&win, old_win, new_win);

5.28 struct Lateral
5.29     LateralNode* top;
5.30     uint64 version;

5.31 struct LateralNode
5.32     LateralNode* next;
5.33     uint row;
5.34     uint width;

5.35 method Lateral.Width(self, row)
5.36     return max(node.width for node in self.nodes where
           node.row > row);

5.37 method LateralNode.Update(self, old_win)
5.38     if $self.row \leq old\_win.Min()$ then
5.39         return self;

5.40     new_node ← node.Clone();
5.41     if $node.width \leq old\_win.push\_width$ then
5.42         new_node.row ← old_win.Min();
5.43     else if $node.width > old\_win.last\_push\_width$ and
           $old\_win.last\_shift = DOWN$ then
5.44         last_min ← old_win.max − old_win.depth/2;
5.45         new_node.row ← min(node.row, last_min);

5.46     next ← new_node.next.Update(old_win);
5.47     if $new\_node.row \leq node.next.row$ then
5.48         return next;                          ▷ Remove node
5.49     else
5.50         new_node.next ← next;
5.51         return new_node;

5.52 function StabilizeLateral(old_lat, old_win)
5.53     if $win \neq old\_win$ or $old\_lat.version = old\_win.version$ then
5.54         return ;                              ▷ Already completed

         //Phase 1: Update local Lateral
5.55     new_lat.top ← old_lat.top.Update(old_win);
         //Phase 2: Push new top if changed width
5.56     upper_bound ← max(old_win.max, stack.row for stack
           in subStacks[old_win.push_width...
           old_win.last_push_width]);
5.57     if $old\_win.push\_width > old\_win.last\_push\_width$ and
           $new\_lat.top.row < old\_win.Min()$ then
5.58         new_node ← LateralNode {              ▷ New top node
5.59             row: old_win.Min(),
5.60             width: old_win.last_push_width,
5.61             next: new_lat.top
5.62         };
5.63         new_lat.top ← new_node;               ▷ Change width
5.64     else if $old\_win.push\_width < old\_win.last\_push\_width$
           and $new\_lat.top.row < upper\_bound$ then
5.65         new_node ← LateralNode {              ▷ New top node
5.66             row: upper_bound,
5.67             width: old_win.last_push_width,
5.68             next: new_lat.top
5.69         };
5.70         new_lat.top ← new_node;               ▷ Change width

5.71     if $new\_lat \neq old\_lat$ then
5.72         new_lat.version ← old_win.version;
5.73         CAS(&lat, old_lat, new_lat);
```

parent $l', l'.next \leftarrow l.next$). If a *Lateral* node $l, l.row > Win_{min}$ has $l.width \leq Win_{push\_width}$, then it is lowered to Win_{min} (line 5.42), as new nodes can have been pushed outside $l.width$ within the *Win*, invalidating its bound. Otherwise, if $l.width > Win_{last\_push\_width}$ and the last shift was downwards, all sub-stacks must have been seen at the previous Win_{min} before it shifted down, so l is lowered to a conservative estimate of the previous Win_{min} (line 5.45). This estimate only deviates from the actual previous Win_{min} when the last Win^{min} was smaller than $Win_{depth}/2$, in which case it overestimates, thus keeping a correct bound.

In the second phase (line 5.55), a new *Lateral* node with $width = Win_{last\_push\_width}$ is pushed if the *width* has changed.

- If $Win_{push\_width} > Win_{last\_push\_width}$, the width has increased, and a *Lateral* node is pushed at Win_{min} to signify that the *width* from there on is smaller. This is not needed for correctness, but is helpful in limiting the Win_{pop_width} if the *width* shrinks in the future.
- If $Win_{push\_width} < Win_{last\_push\_width}$, the *width* has decreased and a new *Lateral* node needs to be pushed at the highest row containing nodes between Win_{push_width} and $Win_{last\_push\_width}$. This can not reliably be calculated from the present *Win* variables, so we simply iterate over the sub-stacks (line 5.56).

In summary, the elastic LpW stack uses a similar idea as the elastic LpW queue, but needs to do extra work to maintain the *Lateral* invariant. However, unless the workload is very pop-heavy and there have been very many elastic *width* changes, the *Lateral* nodes should quickly stabilize and let the *push_width* and *pop_width* be equal.

3.5 Elastic Lateral-as-Window 2D Deque

The elastic LaW deque combines the ideas from the LaW queue and LaW stack, but is mainly similar to the LaW stack due to both having bidirectional *Win* shifts. However, as the deque can operate on both ends of the data structure, it requires one *Win* at the top and one at the bottom.

Algorithm 6 shows how the LaW deque shifts *Win*, but to avoid repetition, it only shows how to shift Win^{top}, as the methods for Win^{bot} are the mirrored equivalents. Enqueues and dequeues are done as in the LaW stack, finding a sub-deque within the current *Win* and updating the sub-deque with a linearizable enqueue or dequeue within the *Win*.

Algorithm 6: Elastic 2D LaW deque

```
6.1  function ShiftTopUp(old_win)
6.2      assert(old_win.shrinking ≠ UP);
6.3      if $old\_win.shrinking \neq NO$ then
6.4          new_win ← old_win.Clone();
6.5          new_win.shrinking ← NO;
6.6          lat.TryReplaceTop(old_win,
               new_win);
6.7      else
6.8          new_depth ← depth_desired();
6.9          new_win ← Window {
6.10             below: old_win,
6.11             max: old_win.max +
                         new_depth/2,
6.12             depth: new_depth,
6.13             width: width_desired(),
6.14             shrinking: NO
6.15         };
6.16         lat.TryPushTop(old_win,
              new_win);

6.17 struct Window
6.18     Window* above;
6.19     Window* below;
6.20     int max;
6.21     uint depth;
6.22     uint width;
         // NO, UP, or DOWN
6.23     enum shrinking;

6.24 global Deque<Window> lat;
     // Conditional updates, only
        linearizing if the old top
        is unchanged.
6.25 method Deque.TryPushTop(old, new);
6.26 method Deque.TryPopTop(old);
6.27 method Deque.TryReplaceTop(old, new);

6.28 function ShiftTopDown(old_win)
6.29     assert(old_win.shrinking ≠ UP);
6.30     if $old\_win.shrinking = NO$ then
6.31         new_win ← old_win.Clone();
6.32         new_win.shrinking ← DOWN;
6.33         lat.TryReplaceTop(old_win, new_win);
6.34     else if $old\_win.below \neq NULL$ then
6.35         assert(∀sub ∈ sub_deques[old_win.below.width... old_win.width]:
               sub.top_row < old_win.max − old_win.depth and
               sub.top_frozen);
6.36         assert(∀sub ∈ sub_deques[0...old_win.width]:
               sub.top_row ≤ old_win.below.max);
6.37         lat.TryPopTop(old_win);
```

As in the LaW stack, the LaW deque relies on all items in the deque always being within one *Win* in the *Lateral*, as described by Property 3.2. To achieve this, it employs the same method of shrinking the *Win* before dequeueing it (line 6.32), and freezing sub-deques outside Win_{width} during a shrinking *Win*, ensuring that no item is pushed outside the current *Win*. However, as a deque can push and pop on both ends, a *Win* can now shrink in either direction. However, due to the *Win* only starting to shrink when it is at an end of the *Lateral*, with another *Win* below or above, and stops shrinking when enqueueing an adjacent *Win* (line 6.6), it can only ever start shrinking from a non-shrinking state.

Property 3.2 (LaW Deque window cover)**.** *If item x is enqueued in sub-deque col_x at row row_x, and has not been dequeued, then $\exists Win^x \in Lateral$ such that $col_x < Win^x_{width}\textbf{and}Win^x_{min} < row_x \leq Win^x_{max}$.*

Finally, to ensure correctness of the deque, some requirements are placed upon the *Lateral* deque, which can be achieved by modifying the Maged deque [29]. Firstly, it must support conditional *TryPush*, *TryPop*, and *TryReplace* methods for both deque ends, which take in the old expected end *Win*, only linearizing if the end is unchanged. The *TryReplace* method is non-standard, and repalces the end *Win*, and is used for shrinking or un-shrinking the end *Win*. Secondly, the deque must be ABA aware when reading an end of the deque, so that it notices if another *Win* has been pushed and then popped between two reads. This also applies to the update methods *TryPush*, *TryPop*, *TryReplace*, which take in the current expected *Win*.

3.6 Elastic Lateral-plus-Window 2D Counter

The 2D counter stands out as the only non-queue 2D design, not allocating any nodes and instead only using the counters which in the queues associate items with rows. Beside incrementing and decrementing counts, much as the 2D stack, the counter also estimates the total counter size by multiplying a sampled sub-count by Win_{width}. Elastic extensions must ensure that they can estimate the total count efficiently, while allowing elastic changes in Win_{width}.

The elastic LpW counter adds a *Lateral* counter that tracks the offset of the true count to the sum of all counters within *Win*, as described in Property 3.3. When reducing Win_{width}, all counts outside Win_{width} are set to 0 and added to the *Lateral* count. Similarly, then increasing Win_{width}, all counts are set to the middle row in *Win*, and those counts are subtracted from the *Lateral*.

Property 3.3 (LpW Counter True Count)**.** *When $Lateral.version = Win_{version}$, all sub-counters outside Win_{width} have count 0, and the true count of the counter (number of linearized increments − decrements) is: $Lateral.offset + \sum_i sub\_counter[i]$.*

Algorithm 7 presents how to shift the window and synchronize the *Lateral* for the elastic LpW counter. In line with earlier LpW designs, shifting *Win* is done by creating a new *Win* with the desired dimensions, including saving the previous Win_{width} and incrementing a version count, finally changing the global *Win* with a CAS (line 7.13).

The *Lateral* is said to be synchronized with *Win* when $Lateral.version = Win_{version}$, and this is required when shifting *Win* or reading the counter. They become unsynchronized every *Win* shift (line 7.13), after which one must call *SyncLateral* to synchronize them. If $Win_{width} = Win_{last\_width}$, synchronizing the lateral only involves incrementing *Lateral.version*.

Algorithm 7: Elastic 2D LpW counter

```
7.1  function Shift(old_win, dir)
7.2      new_win ← Window {
7.3          last_width: old_win.width,
7.4          width: width_desired(),
7.5          depth: depth_desired(),
7.6          version: old_win.version + 1,
7.7      };
7.8      new_max ← if dir = UP then
7.9          old_win.max +
               new_win.depth/2;
7.10     else
7.11         old_win.max − old_win.depth
               + new_win.depth/2;
7.12     new_win.max ←
           max(new_win.depth, new_max);
7.13     CAS(&win, old_win, new_win);
7.14     SyncLateral();

7.15 struct Window
7.16     uint max;
7.17     uint version;
7.18     uint depth;
7.19     uint width;
7.20     uint last_width;

7.21 struct Lateral
7.22     uint offset;
7.23     uint version;
7.24     bool syncing;

7.25 global Window* win;
7.26 global Lateral* lat;

7.27 function SyncLateral()
7.28     repeat
7.29         old_win ← win;
7.30         old_lat ← lat;
7.31     until old_win = win;
7.32     win_mid ← old_win.max − old_win.depth/2;
7.33     if old_lat.version < old_win.version  &  !old_lat.syncing then
             // Update Lateral offset count
7.34         offset ← if old_win.width < old_win.last_width then
7.35             sum of counters[old_win.widthwin_mid. old_win.last_width];
7.36         else
7.37             − win_mid ∗(old_win.width − old_win.last_width);
7.38         new_lat ← Lateral, offset: old_lat.offset − offset, version:
               old_lat.version, syncing: true};
7.39         CAS(&lat, old_lat, new_lat);
7.40     old_lat ← lat;
7.41     if old_lat.version < old_win.version then
7.42         for i ∈ {old_win.width... old_win.last_width} do
7.43             old_counter ← counters[i];
7.44             if old_win ≠ win return;
7.45             new_counter ← {count: 0, version: old_counter.version + 1};
7.46             CAS(&counters[i], old_counter, new_counter);
7.47         for i ∈ {old_win.last_width... old_win.width} do
7.48             old_counter ← counters[i];
7.49             if old_lat ≠ lat return;
7.50             new_counter ← {count: win_mid, version:
                   old_counter.version + 1};
7.51             CAS(&counters[i], old_counter, new_counter);
7.52         new_lat ← Lateral {offset: old_lat.offset, version: old_win.version,
               syncing: false};
7.53         CAS(&lat, old_lat, new_lat);
```

Otherwise, synchronizing the *Lateral* is done in two parts that ensure the correctness of Property 3.3:

1) First, the *Lateral* offset change is calculated. If the Win_{width} has decreased, the offset increases by the sum of all counts outside Win_{width} (line 7.35), as these counts will be reset to 0. To avoid updates to the counters after these reads, the readers also increment the version count on the sub-counters. If Win_{width} has increased, the offset decreases by $Win_{mid} := Win_{max} - Win_{depth}/2)$ multiplied by the change in Win_{width} (line 7.37), as those counters will be incremented to Win_{mid} before being operated on in the new window. This step linearizes by updating the *Lateral* on line 7.39, also setting *Lateral.syncing*, to flag that the *Lateral* is between the two synchronization steps.
2) Then, the potential sub-counters outside Win_{width} are set to 0 (line 7.46), or the new ones inside Win_{wdith} are set to Win_{mid} (line 7.51), counteracting the change in the *Lateral* above. This linearizes by updating the *Lateral* to a synchronized state at line 7.53.

Incrementing and decrementing a counter is done as in the static 2D counter, by reading a sub-counter, validating that it is within *Win*, additionally verifying that the *Lateral* is synchronized with *Win*, and then updating the sub-counter with a CAS. Reading the counter is done equivalently and returns *sub-count* $\cdot$ Win_{width} + *Lateral.offset*.

4 Analysis

In this section, we prove the correctness and relaxation bounds for our elastic designs. Our elastic bounds give the worst-case rank errors, but if no elastic changes are made, our elastic designs have equal or lower error bounds compared to their static 2D counterparts.

Static k out-of-order relaxation is formalized by defining and bounding a *transition cost* (rank error) of the "get" methods within the linearized concurrent history [6]. 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 similarly to static out-of-order, with the difference that the rank error bound is a function of the relaxation configuration history during the lifetime of the accessed item. In the simplest case, such as the elastic LaW queue from Section 3.2, the rank error bound for every dequeued item is a function of Win^{deq}_{width} and Win^{deq}_{depth} when the item is dequeued.

Lock-freedom. To avoid repetitive arguments, we here collectively state that our designs are lock-free as (i) each sub-structure, the *Win*, and the *Lateral* are updated in a lock-free manner (by linearizing with a CAS), (ii) the *Lateral* is updated at most twice every window, and (iii) that there cannot be an infinite number of window shifts without progress on any of the sub-structures [7].

Memory usage. The required memory of our designs is presented in Property 4.1. It is simple to generalize across designs, so we motivate it in the following paragraph. Importantly, as there is only an additive factor with the *width*, the memory overhead compared to a strict data structure is not very large.

Property 4.1 (Memory usage of Elastic designs)**.** *Our elastic queues, stacks, and deque, use $\mathcal{O}(n + w)$ memory, where n is the number of items in the data structure, and w is the largest width of all rows. The counter only uses $\mathcal{O}(w)$ memory.*

The memory usage of our designs is split into three parts: (1) sub-structures, (2) the *Lateral* (3) inserted items. Each sub-structure uses a small, constant amount of memory, when ignoring stored items. Thus, if keeping a dynamic vector of the number of sub-structures needed, they use $\mathcal{O}(w)$ memory. For the counter, as the sub-counters and *Lateral* are of $\mathcal{O}(1)$ size, its total size is $\mathcal{O}(w)$. Otherwise, the *Lateral* —in the worst case—has one *Lateral* node per occupied row in the data structure, which is $\mathcal{O}(n)$, where n is the current number of items. However, in most cases, as in the queues, the worst case is one *Lateral* node per $Win_{width} \cdot Win_{depth}$ items, which is often quite large. Finally, each item is enqueued into a single sub-structure, using the same memory as a strict linked list of $\mathcal{O}(n)$, where n is the list size.

Notation. When referencing item x, we denote $Win^{enq\,x}$ as the *Win* which was read at the enqueue of x, analogously defining $Win^{deq\,x}$. Furthermore, $Win^{max\,x}_{field}$ (or $Win^{min\,x}_{field}$) reference the maximum (or minimum) value of Win_{field} during the lifetime of x. Finally, row_x refers to the row where x was inserted and col_x to the sub-structure x was inserted.

4.1 Elastic LpW Queue

The monotonically increasing Win_{max} of the FIFO queues makes their analysis relatively straightforward. The main difficulty with the LpW queue is that Win^{enq}_{depth} does not necessarily equal Win^{deq}_{depth}, which means that $Win^{enq\,x}$ does not have to span the same rows as $Win^{deq\,x}$.

The proof idea of the rank error bound is that we in Lemma 4.2 prove that any changes in Win^{enq}_{width} are *first* inserted into the *Lateral*, and *then* appear in a new Win^{enq}_{width}. This essentially proves the *Lateral* is consistent. Using this, we essentially show that the worst rank error of dequeueing item x is when $Win^{enq\,x}_{max} = row_x = Win^{deq\,x}_{min} + 1$, and as $Win^{enq\,x}_{width} = Win^{deq\,x}_{width}$, this leads to a worst-case rank error of $(Win^{enq\,x}_{width} - 1)(Win^{enq\,x}_{depth} + Win^{deq\,x}_{depth} - 1)$.

To simplify notation, we introduce an ordering of the windows such that $Win^i < Win^j$ if $Win^i_{max} < Win^i_{max}$.

Lemma 4.2. *If a Lateral node l is enqueued at time t, then $Win^{deq}_{max} < row_l$ at t.*

Proof. We first note that $row_l \leftarrow Win^{enq}_{max} + 1$ when it is enqueued (line 2.30). Furthermore, the enqueue of l completes before the window shifts, and it cannot be enqueued again after the window has shifted as the comparison against the strictly increasing row count of the tail will fail (line 2.28). As $Win^{deq}_{max} \leq Win^{enq}_{max}$ (line 2.45), and $Win^{enq}_{max} < row_l$ at t, it holds that $Win^{deq}_{max} < row_l$ at t. □

Theorem 4.3. *The elastic LpW queue is linearizeable with respect to a k out-of-order elastically relaxed FIFO queue, where for every dequeue of x, $k = (Win^{enq\,x}_{width} - 1)(Win^{enq\,x}_{depth} + Win^{deq\,x}_{depth} - 1)$.*

Proof. Enqueues and non-empty dequeues linearize with a successful update on a MS sub-queue. An empty dequeue linearizes when $Win^{deq}_{max} = Win^{enq}_{max}$ after a double-collect where it sees all sub-queues within Win^{deq}_{width} empty. Lemma 4.2 together with $Win^{deq}_{max} = Win^{enq}_{max}$ gives that all nodes must be within Win^{deq}_{width}, meaning that empty returns can linearize at a point where the queue was completely empty.

The core observation for proving the out-of-order bound is to observe that the row counts for the sub-queues and the max of the windows strictly increase. For a node y to be enqueued before x, and not dequeued before x, it must hold that $Win^{enq\,y} \leq Win^{enq\,x}$ and $Win^{deq\,y} \geq Win^{deq\,x}$. As $Win^{enq\,x}_{min} \leq Win^{deq\,x}_{max}$, we can bound the possible row of y by $Win^{deq\,x}_{min} \leq row_y \leq Win^{enq\,x}_{max}$. Using Lemma 4.2 together with the fact that Win^{deq} only spans rows with one *width* at a time (line 2.45), we get that these valid rows for y must have width $Win^{enq\,x}_{width} = Win^{deq\,x}_{width}$. As both $Win^{enq\,x}$ and $Win^{deq\,x}$ must share at least row_x, it holds that $Win^{enq\,x}_{min} \leq Win^{deq\,x}_{max}$. This is then used to limit the valid number of row_y by $Win^{enq\,x}_{max} - Win^{deq\,x}_{min} = Win^{enq\,x}_{min} + Win^{enq\,x}_{depth} - Win^{deq\,x}_{max} + Win^{deq\,x}_{depth} \leq Win^{enq\,x}_{depth} + Win^{deq\,x}_{depth} - 1$. As all operations within each sub-queue are ordered, we reach the final bound of $(Win^{enq\,x}_{width} - 1)(Win^{enq\,x}_{depth} + Win^{deq\,x}_{depth} - 1)$. □

Furthermore, one can follow the same arguments to prove that the *delay* (as introduced in Section 2) for the elastic LpW queue is bounded by the k in Theorem 4.3.

4.2 Elastic LaW Queue

Analyzing correctness for the elastic LaW queue becomes simple due to the strict order the *Lateral* windows define over all operations. Namely, for item x, as $Win^{enq\,x} = Win^{deq\,x}$, the enqueue and dequeue operations are totally ordered against all operations linearizing in other windows, and can only be out-of-order with respect to other operations in the same window. This leads to the simple bound in Theorem 4.4. As Win^{deq}_{depth} cannot change arbitrarily as in the LpW queue, the LaW queue gets a tighter worst-case bound than the LpW queue in Theorem 4.3.

Theorem 4.4. *The elastic LaW queue is linearizeable with respect to a k out-of-order elastically relaxed FIFO queue, where $k = (Win^{deq}_{width} - 1)Win^{deq}_{depth}$.*

Proof. The key observation is that every Win^{enq} must fill all Win^{deq}_{depth} rows of Win^{deq}_{width} items each before shifting to the next Win. These items can be enqueued in any order, except that each sub-queue is totally ordered. Enqueues to different Win are totally ordered due to the sequential semantics of the *Lateral*. The Win^{deq} uses the same Win structs as the Win^{enq}, by traversing the *Lateral* of past Win^{enq}, not shifting past such a window until it has also dequeued all its $Win_{width} \times Win_{depth}$ items.

Thus, as the oldest items in the queue always are in Win^{deq}, and dequeues only ever operate within Win^{deq}, together with the fact that the sub-queues are totally ordered, means that a dequeue can at most skip the first $(Win^{deq}_{width} - 1)Win^{deq}_{depth}$ items. $\square$

As in the LpW queue, it is simple to see that the k in Theorem 4.4 also bounds the *delay* of the LaW queue.

4.3 Elastic LaW Stack

Bounding the rank error for the LaW stack becomes relatively simple thanks to Property 3.1, which guarantees that items are always within one of the Win in the *Lateral*. Furthermore, bounding the rank error of an individual item x is helped by the key insight that any item pushed after x, but not popped before x, must have been pushed into a Win which contains x. Thus the error bound stretches Win_{depth} up and down for every sub-stack, potentially reordering x against the $Win_{width} - 1$ other sub-stacks.

Theorem 4.5. *The elastic LaW stack is linearizable with respect to a k out-of-order elastically relaxed stack, where k is bounded for every item x as $k = (Win^{max\,x}_{width} - 1)(2Win^{max\,x}_{depth} - 1)$.*

Proof. As the algorithm always tries to pop from all sub-stacks within Win_{width} before returning empty, that it linearizes with a CAS like the Treiber stack [28], and that Property 3.1 gives that there are no items not contained in a Win inside *Lateral*, it is obvious that the design is correct with respect to pool specifications.

To bound the number of items pushed after x, but not popped before x, we will bound on what rows they can be pushed. If an item y is pushed during the lifetime of x, then Property 3.1 gives that it must be pushed in a window $Win^{push\,y}$ where $Win^{push\,y}_{max} \geq row_x$. Furthermore, when popping x, Property 3.1 gives that no items are above $Win^{pop\,x}_{max}$. Thus, the possible range of allowed rows for items pushed after x, but not popped before x, spans $Win^{pop\,x}_{depth} + Win^{push\,y}_{depth} - 1$ rows where the -1 comes from the two windows overlapping with at least row_x. As there are no out-of-order items in col_x, the maximum error is bounded by $(Win^{max\,x}_{width} - 1)(2Win^{max\,x}_{depth} - 1)$. $\square$

Another way to view the relaxation errors in the LaW stack it to logically relax both pushes and pops, instead of just pops, which is the norm [6]. Using similar arguments as above, one would then see that the worst-case rank errors would be $(Win^{push\,x}_{width} - 1)Win^{push\,x}_{depth}$ for the push and $(Win^{pop\,x}_{width} - 1)Win^{pop\,x}_{depth}$ for the pop. This alternate formulation might be more useful for describing the actual relaxation in the stack.

4.4 Elastic LpW 2D Stack

Analyzing the LpW stack is more involved than analyzing the simpler LaW stack. The idea is to first prove that the *Lateral* correctly bounds row widths in Lemma 4.6, then bound the size of sub-stacks in Lemma 4.7 and 4.8, and finally derive the rank error bound in Theorem 4.9.

For brevity, we denote $Win_{shift} := \lfloor Win_{depth}/2 \rfloor$ and the top row (size) of sub-stack j as N_j. Furthermore, we introduce a *width bound* ($width_r$) for each row r as $l.width$ if there exists a *Lateral* node $l, l.next.row < r \leq l.row$, or Win_{push_width} if $r > l.row\ \forall\ l$ (if this properly bounds the row widths, the *Lateral* is *consistent*). Due to *Lateral* nodes being removed from the stack if their row overlaps the next node's row, this width bound is uniquely defined.

Lemma 4.6. *At the moment preceding the linearization of each window shift, it holds for each row r and every item x wherer$ow_x = r$, that $width_r \geq index_x$.*

Proof. Informally, this lemma states that the *Lateral* stack properly bounds all rows with widths greater than Win_{push_width} after the call to *Stabilize* at line 5.52. We prove this with induction over the sequence of window shifts, and it is easy to see that it holds at the first window shift, as all nodes will be pushed within Win_{put_width}. Now, if the lemma held at the previous window shift, we check if it continues to hold for the next shift where we shift from Win^i.

First, we inspect rows at and below Win^i_{min} and note that during the lifetime of Win^i, all nodes are pushed above Win^i_{min}, and $width_r$ will not be changed at rows at or below Win^i_{min} (lines 5.38, 5.42, 5.45) from lowering a *Lateral* node. Thus, if $Win^i_{push\_width} = Win^i_{last\_push\_width}$, the lemma continues to hold for all rows lower or equal to Win^i_{min}. If the $Win^i_{push\_width} \neq Win^i_{last\_push\_width}$ a new *Lateral* node can be inserted with width $Win^i_{last\_push\_width}$. This node changes row bounds for rows at or below Win^i_{min} if there was no other *Lateral* node above Win^i_{min} at the shift to Win^i, and in that case those rows would have before Win^i been bounded by $Win^i_{last\_push\_width}$, which is the same as the width bound this node enforces. Therefore, the induction invariant continues to hold for rows at or below Win^i_{min}.

Now we inspect rows above Win^i_{min} to see if the invariant also holds there. Firstly, no row will have a width bound smaller than Win_{push_width}, as smaller widths will be lowered or inserted to or below Win^i_{min} (lines 5.42, 5.58). Therefore, only items above Win^i_{min} outside $Win^i_{push\_width}$ can break the invariant. In the lowering phase, *Lateral* nodes $l, l.width >$

$Win^i_{last\_push\_width} \wedge l.width > Win^{push\_width}$ are lowered iff the shift to Win^i was downwards, as then all sub-stacks outside $Win^i_{last\_push\_width}$ were seen at the bottom of the last window. Thus, if $Win^i_{push\_width} \geq Win^i_{last\_push\_width}$ no nodes could have been pushed outside $Win^i_{push\_width}$ since the shift to Win^i, and as all widths bound held then, they will hold at the shift from Win^i. Otherwise, if $Win^i_{push\_width} < Win^i_{last\_push\_width}$, every row $r, r \leq l.row$ for the topmost *Lateral* node l must have a valid bound, and the rows above $l.row$ were before Win^i bounded by $Win^i_{last\_push\_width}$ which is smaller than the new bound $Win^i_{push\_width}$, so the width bounds must hold for all rows in this case as well. □

Lemma 4.7. *If x lives on the stack during $'x$, then for any item y pushed during $'x$, $row_y \geq Win^{push\,x}_{min} - Win^{max\,x}_{shift}$.*

Proof. This is proved by contradiction. Assume x was pushed at time t_x and there exists an item $y : row_y \leq Win^x_{min} - Win^{max\,x}_{shift}$, pushed at time $t_y > t_x$. Call the point in time where a thread shifted the window to $Win^{push\,y}$ as t_s.

- If $t_s < t_x$, then as $t_y > t_x$, x must be pushed during $Win^{push\,y}$. However, t_x can't be during $Win^{push\,y}$, as an item is pushed at or below Win_{max}.
- If instead $t_s > t_x$, we call the window before $Win^{push\,y}$ as Win^z. For a thread to shift from Win^z, it must have seen $\forall j, N_j = Win^z_{min}$ (as Lemma 4.6 shows iterating over Win_{pop_width} is the same as iterating over all j) at some point t_z (set t_z as the time it started this iteration) during Win^z. As $Win^z_{min} < row_x$, $t_x > t_z$, which means that t_x must have been during Win^z, as we above showed $t_x < t_s$. This is impossible as during Win^z, items are pushed at or below Win^z_{max}.

□

Lemma 4.8. *If x lives on the stack during $'x$, then for any item y pushed, but not popped, during $'x$, $row_y \leq Win^{pop\,x}_{max} + Win^{max\,x}_{shift}$.*

Proof. Similarly to Lemma 4.7 this is proved by contradiction. Assume that x was pushed at t_x, popped at time t'_x and that there exists an item $y, row_y > Win^{pop\,x}_{max} + Win^{max\,x}_{shift}$ pushed at t_y and not popped at t'_x, where $t_x < t_y < t'_x$. Call the point in time where a thread shifted to $Win^{pop\,x}$ as t_s.

- If $t_s < t_y$, then y must have been pushed in the same window as x was popped. But items are only pushed at or below Win_{max}, which contradicts the assumption, as y is pushed too low.
- If $t_y < t_s$, we call the window proceeding $Win^{pop\,x}$ as Win^z. For a thread to shift from Win^z, it must have seen $\forall j, N_j = Win^z_{min}$ at some point t_z (set t_z as the time it started the iteration, seeing the first $N_j = Win^z_{min}$) during Win^z. Therefore, $t_z < t'_y$, which is impossible as $Win^z_{max} < row_y$ and $t_y < t_s$.

□

Theorem 4.9. *The elastic LpW stack is linearizeable with respect to a k out-of-order elastically relaxed stack, where k is bounded for every item x as $k = (Win^{max\,x}_{width} - 1)(3Win^{max\,x}_{depth} - 1)$.*

Proof. Assume that y is pushed after x and not popped before x. Then Lemma 4.7 and 4.8 gives $Win^{push\,x}_{min} - Win^{max\,x}_{shift} \leq row_y \leq Win^{pop\,x}_{max} + Win^{max\,x}_{shift}$. As each sub-stack is internally ordered and the maximum number of items pushed after x and not popped before x becomes $(Win^{max\,x}_{push\_width} - 1)(Win^{pop\,x}_{max} - Win^{push\,x}_{min} + 2Win^{max\,x}_{shift}) \leq (Win^{max\,x}_{push\_width} - 1)(3Win^{max\,x}_{depth} - 1)$. □

As for the LaW stack, one can reformulate this rank error bound such that both push and pop operations are relaxed, with worst-case bounds of $(Win^{push\,x}_{last\,width} - 1)Win^{push\,x}_{last\,depth} + (Win^{push\,x}_{width} - 1)Win^{push\,x}_{depth}/2$ for the push and $(Win^{pop\,x}_{last\,width} - 1)Win^{pop\,x}_{last\,depth} + (Win^{pop\,x}_{width} - 1)Win^{pop\,x}_{depth}/2$ for the pop, where $Win_{last\,width}$ and $Win_{last\,depth}$ is the value of the respective dimension in the previous Win. The reason the bounds for the LpW stack are looser than the LaW stack is that it does not shrink Win before shifting, potentially linearizing in the previous Win at times.

4.5 LaW Elastic 2D Deque

The rank error of when dequeueing x from one end of a deque becomes the number of other items pushed to the same end after x, but not popped before x. Even though we can now push items from both ends of the deque, using Property 3.2 instead of Property 3.1, proving the following correctness theorem of the LaW deque follows the same path as the LaW stack. We therefore omit the proof, and refer the reader to the proof of Theorem 4.5.

Theorem 4.10. *The elastic LaW deque is linearizable with respect to a k out-of-order elastically relaxed deque, where k is bounded for every item x as $k = (Win^{max\,x}_{width} - 1)(2Win^{max\,x}_{depth} - 1)$.*

4.6 LpW Elastic 2D Counter

The relaxation bound for the LpW counter is slightly different from the queues, as we the out-of-order error for the counter is defined as the absolute difference between the returned count and the true count, contrary to the ordering of items. The idea is to use Property 3.3 to restrict the area of concern to columns within Win_{width}, and then see that sub-counts must be within the current, or previous Win, bounding the maximum difference between any two sub-counters.

Theorem 4.11. *The elastic LpW counter is linearizable with respect to a k out-of-order elastically relaxed counter, where $k = (Win_{width} - 1)(Win_{depth}/2 + Win^{last}_{depth} - 1)$, where Win^{last} is the previous Win.*

Proof. Using Property 3.3, and assuming we sample sub-counter i for the read, the relaxation error of the read operation becomes $|\sum_j^{Win_{width}}(count_i - count_j)| \leq \sum_j^{Win_{width}}|count_i - count_j| \leq (Win_{width} - 1)max_j|count_i - count_j|$.

As we only consider sub-counts within Win_{width}, the maximum difference between any two sub-counters can be proven in the same was as in the static 2D case, but also considering elastic Win_{depth}. Lemma 4 in the static 2D paper [7] proves that all counts are always in two adjacent windows, and with elastic Win_{depth}, this can easily be adapted to state that all sub-counts are within Win^{last} or Win. As the shift from Win^{last} to Win is $Win_{depth}/2$, the maximum difference between any two sub-counts becomes $Win_{depth}/2 + Win^{last}_{depth} - 1$. □

5 Experimental Evaluation

In this section, we first evaluate the overhead of our elastic queues, stacks, and counters, when not using their elastic capabilities, after which we examine how these elastic capabilities can be utilized in dynamic executions. All experiments run 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 with the 6.13.1 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 [7]. We use SSMEM [31] for efficient epoch-based memory management [32] of our dynamic memory. Furthermore, to facilitate fair comparison against the static 2D designs, we don't change the used sub-structures, and thus use the 16-byte CAS for the counted head and tail pointers in the queues and stacks. Our implementations, including scripts to re-run all experiments, are available in the Zenodo repository [2].

5.1 Static Relaxation

To understand the potential overhead of our elastic extensions, we compare their scalability, during constant relaxation, against that of state-of-the-art k out-of-order and strict concurrent designs. For the queues, we select the static 2D queue [7] and the k-segment queue [9] as k out-of-order designs. Furthermore, we select the LCRQ [26] and the YMC queue [27] as the state-of-the-art strict FIFO queues. For the stacks, we select the 2D stack [7] and the k-segment stack [6] as k out-of-order designs, the lock-free elimination-backoff stack [33] as an efficient strict design, and the Treiber stack [28] as a baseline. For the counters, we select the 2D counter [7] as the relaxed design, and two simple counters implemented on top of fetch-and-add (FAA) and CAS for our strict designs. The counters are strictly non-negative, but the simple FAA counter violates this due to the nature of FAA, giving it an advantage when the other counters have to do emptiness checks. All data structures were implemented in our framework using SSMEM [31] for memory management, with external non-trivial designs based on their respective paper's implementation.

We use a benchmark where threads repeatedly perform insert or remove operations at random, each with equal probability, for a duration of one second. Test results are averaged over 10 runs, with standard deviation included in the plots. The benchmark requires specifying the maximum rank error of each relaxed data structure, and as the optimal choice of Win_{width} and Win_{depth} is application-specific [7], we set $Win_{width} = 2 \times \texttt{nbr\_threads}$, and use the maximum Win_{depth} which does not violate the bound. This choice of *Win* dimensions gives acceptable scalability, and we saw the same trends when using other combinations.

Accurately measuring rank errors without altering their distribution is an open problem, and we adapt a common method in the literature [7], [20], [34] of encapsulating the linearization points of all methods in a global lock, giving a total order of operations. This order can be used to re-run the execution with a strict data structure, where the rank error of each operation easily can be calculated. For queues and stacks, the rank error of dequeuing x is the distance between x and the tail or top item, and the rank error for counters becomes the absolute difference between the true count at each operation and the returned count.

Figure 4 presents how the elastic designs scale with threads, including both settings with pre-filled and empty data structures. Pre-filling drastically changes the dynamics of dequeues, as they don't have to do potentially costly emptiness checks. However, the throughput trends are mostly identical irrespective of prefill. The largest difference when not using pre-fill is that the rank errors become smaller, as they naturally cannot exceed the queue size at any time. All experiments show that the elastic designs have close to zero overhead when compared to the static 2D designs. Similarly, the elastic designs don't show any increase in rank errors when compared to the static designs. Overall, the elastic designs scale as well as the static 2D framework, and out-scale the other data structures, highlighting the light weight of the *Lateral*.

Figure 5 further shows how the designs scale with increasing rank error bounds. Here we pre-fill the data structures with 10^6 items, as the throughput is not that affected as seen in Figure 4, and the designs otherwise just become pools. The elastic LaW stack and elastic LpW counter have slight overheads in throughput at lower relaxation levels. The reason for this is that the *Lateral* incurs some overhead when shifting *Win* in these designs, and such shifts happen more often at lower relaxation levels. However, overall, the elastic 2D designs scale monotonically with relaxation, mostly performs identically to the static 2D designs, and outperforms other designs as relaxation increases.

5.2 Elastic Relaxation - Dynamic Controller

In this seciton, we evaluate the usefulness of our elastic extensions in dynamic workloads with varying degrees of parallelism. We focus on the LaW queue, as relaxed FIFO queues are more popular in the literature than stacks or counters, and as we think the simplicity of the LaW design makes it preferable over the LpW queue. As the optimal Win_{width} is tightly coupled to the degree of parallelism, we design a lightweight controller to dynamically adjust Win_{width} based on measured contention. We first examine how the controller reacts over time in a dynamic producer-consumer micro-benchmark, and then how it can provide end-to-end benefits in a parallel BFS algorithm with varying number of threads.

The dynamic controller should balance relaxation errors with latency, and a lightweight control signal for this is the failure rate of sub-structure CAS linearizations [1], [34], [35]. So the controller should strive to keep the failure rate constant, while being stable and responding quickly, while imposing low latency overhead. The conference version of this paper [1] used simple overflow-based thread-local controllers, similar to CA-PQ [35], but such simple and local controllers have a hard time reacting fast without significantly overshooting and being unstable. Instead, we here use a shared wait-free controller inspired by leaky proportional-integral controllers from control theory [36]. The controller keeps an exponential moving average (EMA)

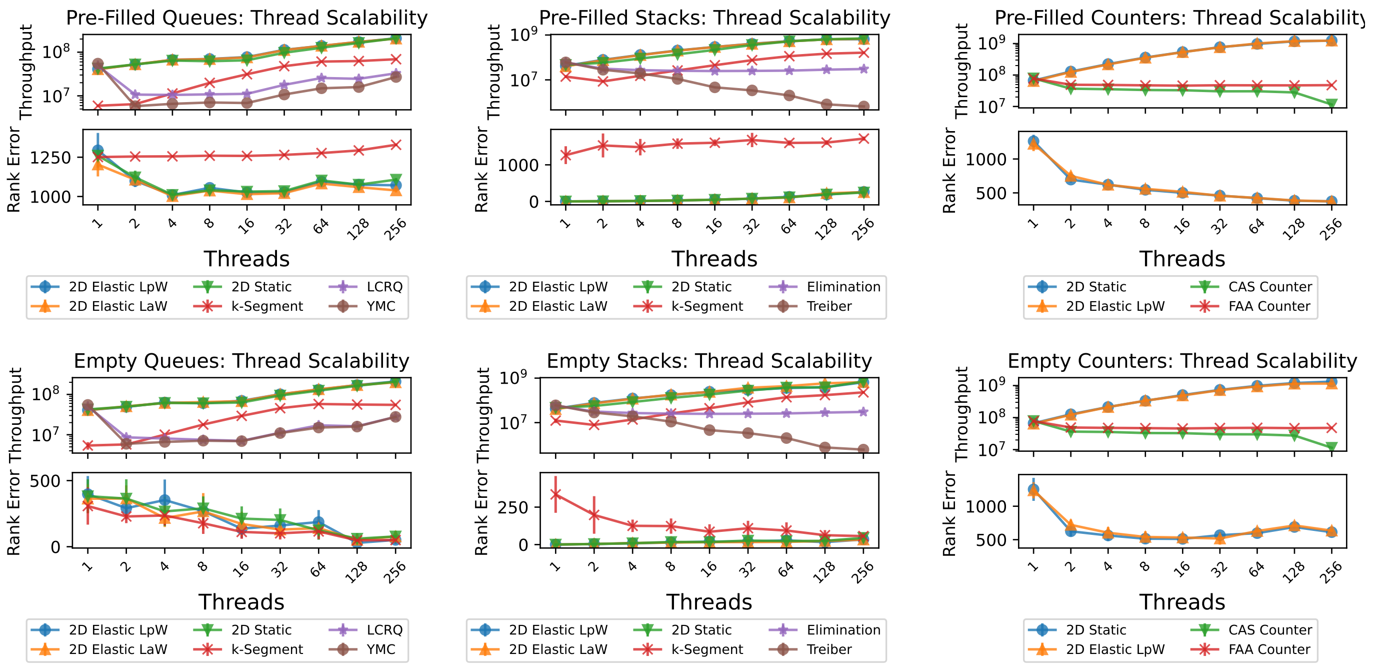


Figure 4: Scalability of throughput and mean rank error with increasing number of threads, using a rank error bound of $k = 5 \times 10^3$. Plots in the top row have a pre-fill of 10^6, and the plots in the bottom row do not use any pre-fill.

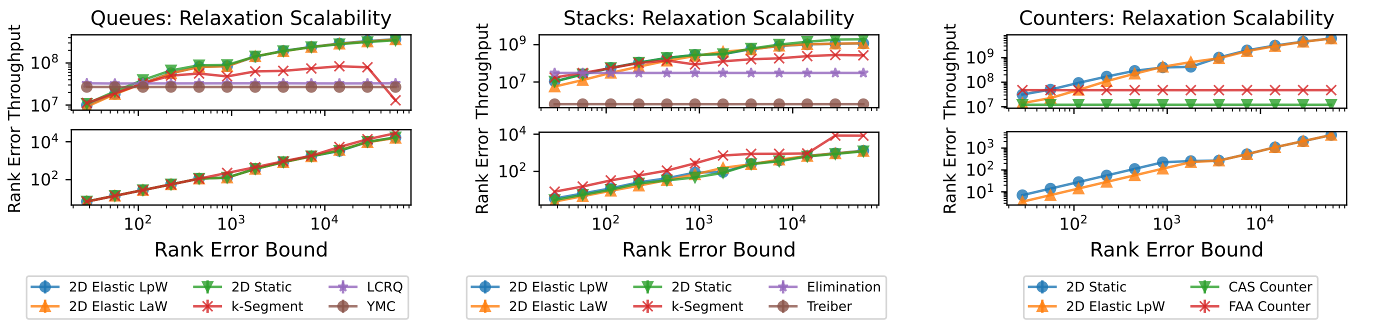


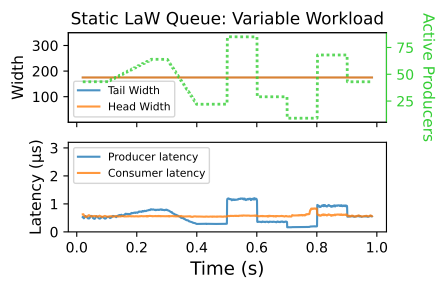
Figure 5: Scalability of throughput and mean rank error as rank error bound increases, using 256 threads and 10^6 pre-fill.

of the total sub-queue CAS failure rate, which provides an accurate an reasonably up-to-date estimate of the true failure rate. To reduce overhead, the threads only update this EMA every 200 operations, and if an update fail due to contention, it is aborted. The controller can be seen as a leaky integrator with a small proportional factor.

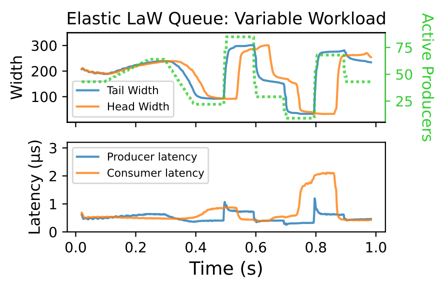
The controller has a target T, which is the desired fraction of failing sub-queue CAS. When a thread synchronizes its state with the controller, it computes its local failure rate F and logarithmic error $E_t = \ln(F/T)$ since the last synchronization, updating the global error E_g as $E_g = \alpha E_t + (1 - \alpha)E_g$, which is the EMA of failures. When shifting Win^{enq}, its width w_i is updated from the previous width w_{i-1} based on the global error: $\ln(w_i) = \ln(w_{i-1}) + K_p E_g \Leftrightarrow w_i = w_{i-1} \exp(K_p E_g)$, where K_p is a gain constant. The EMA lives in the log domain as its updates then become symmetric w.r.t. percentage deviations from T. Finally, to reduce windup, E_g is reduced by 25% after each change to Win^{enq}_{width}. In our experiments, we use $\alpha = 0.01, K_p = 0.01, T = 0.012$. Configuring T can easily be done by finding the failure rate of good configurations in static executions. Varying α between 0.1 and 0.005 did not make a large difference. Finally, K_p is the most important to tune, as a too large one can cause instability, and we therefore used this relatively low value.

5.2.1 Producer-Consumer Micro-Benchmark

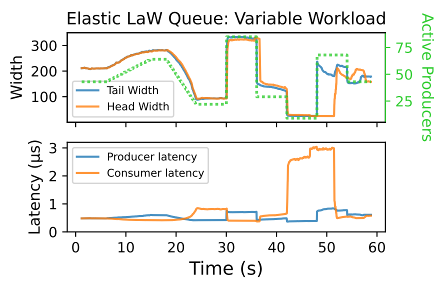
To evaluate the controller's ability to adapt to workload changes, we design a micro-benchmark that emulates a dynamic producer-consumer scenario. It consists of a shared relaxed FIFO queue accessed by 128 threads. One third of the threads (42) act as consumers, continuously attempting to dequeue tasks. The remaining 86 threads act as producers, repeatedly enqueuing tasks. Furthermore, the number of active producers varies over time, as shown in Figure 6. There queues start empty. This setup models a high-contention server-side task queue, where a fixed pool of worker threads (consumers) handles a fluctuating stream of external requests (producers). The benchmark tests whether the controller can dynamically adjust Win^{enq}_{width} to maintain



(a) Static Relaxation, 1 second



(b) Dynamic Relaxation, 1 second



(c) Dynamic Relaxation, 1 minute

Figure 6: Latency, width, and active producers over time, in a producer-consumer benchmark where the number of producers vary over time. All plots use the LaW queue, but the leftmost uses static relaxation, while the other uses the dynamic controller to keep the producers' contention stable.

a suitable balance between relaxation and latency under shifting load.

Figure 6 shows the average operational latency for consumers and producers, as well as the Win^{enq}_{width} and Win^{deq}_{width}, averaged over 10 runs, both for static Win_{width} and when the producers use the dynamic controller. It uses $Win_{depth} = 8$ and the static Win_{width} was set to 180, the stable width of the controller when half the producers are active. When using static width, the producers' latency clearly scales with the number of active producers, while the consumer latency is mostly constant. When using the controller, the Win^{enq}_{width} quickly adapts to the number of active producers, which in turn makes its producers' latencies more stable than the static queue. Notably, as we only directly change Win^{enq}_{width}, the change must propagate through the queue for it to affect Win^{deq}_{width}, delaying the Win^{deq}_{width} change. Furthermore, the latencies of the producers and consumers are not completely independent, as for example seen around 53 seconds in Figure 6c where the increase in Win^{deq}_{width} casuse the consumers' latencies to decrease, which in turn increases the producers' latencies and contention, in turn also increasing Win^{enq}_{width}. This dependence of different aspects of the system highlights the difficulty of choosing a good control signal for the controller.

The results in Figure 6 shows that the controller is quick, stable, has reasonably low overhead, and mostly adjusts the Win^{enq}_{width} to be roughly proportional to the number of active producers. One disadvantage is how the size of Win^{deq}_{width} seems to affect the contention at the enqueues, for example leading to Win^{enq}_{width} not dropping as fast or much as we would expect at 0.9 seconds in Figure 6b, or suddenly increasing without the nubmer of threads changing after 52 seconds in Figure 6c. However, this is hard to avoid when basing the controller on CAS contention, and we anticipate it will not pose large issues in practice.

The most important consideration when using such a controller is defining the desired behavior. In this synthetic scenario, we aimed at stabilizing producer latency effectively minimizing the relaxation of incoming user requests, while still keeping user latency acceptable under high load. Since the number of consumers remained constant and they performed no elastic changes, their latency was not prioritized and occasionally rose to as much as 3 μs. If consumer latency is also a concern, they could be included in the controller's feedback loop, or a lower bound could be placed on Win^{enq}_{width} to prevent excessive spikes in their latency.

Table 1: Thread workloads for the BFS benchmark. They describe the fraction of active threads over time (t), where t is normalized to $[0, 1)$, after which the pattern repeats.

Name	Definition	Description
Const.	1	Constantly max threads
MJ	See Figure 7	Google queries, 1 day
Wikipedia	See Figure 7	Wikipedia queries, 1 week
Lin. Inc.	t	Linear increase
Lin. Dec.	$1 - t$	Linear decrease
Exp. Inc.	$\frac{1-e^{5t}}{1-e^5}$	Exponential increase
Exp. Dec.	$1 - \frac{1-e^{5t}}{1-e^5}$	Exponential decrease

Table 2: Graphs used in the BFS benchmarks.

Graph	Nodes	Edges	Graph Type
USA (road_usa [37])	$24M$	$58M$	road
EU (europe_osm [37])	$51M$	$108M$	road
GL5 (geolife5_sym [38], [39])	$25M$	$309M$	kNN

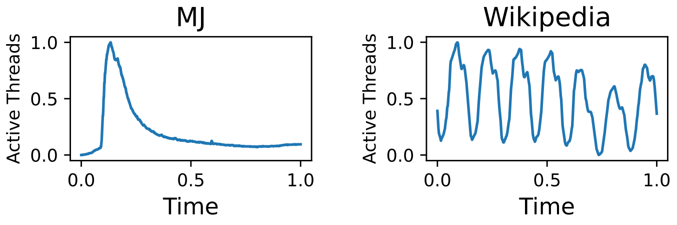


Figure 7: Realistic workloads used in BFS benchmarks. The MJ workload (left) is the number of google searches of Michael Jackson the day of his death, from 13pm PDT [40]. The Wikipedia workload (right) is the number of search queries on the English Wikipedia over one week in 2009 [41].

5.2.2 *Breadth-First Search Macro-Benchmark*

To evaluate the effectiveness of our elastic queue designs and dynamic controller in a realistic algorithmic setting, we implement a concurrent breadth-first search (BFS) benchmark in a setting with dynamically accesible parallelism.

		Const.		**MJ**		**Wikipedia**		**Lin. Inc.**		**Lin. Dec.**		**Exp. Inc.**		**Exp. Dec.**		**GMean**	
		Time	**Work**	**Time**	**Work**	**Time**	**Work**	**Time**	**Work**	**Time**	**Work**	**Time**	**Work**	**Time**	**Work**	**Time**	**Work**
USA	Static	664	2.48	1501	2.96	815	2.60	979	3.01	735	2.44	1864	2.94	667	2.46	956	2.69
	Dynamic	**549**	**2.11**	**1012**	**1.63**	**621**	**1.84**	**673**	**1.77**	**659**	**1.96**	**1300**	**1.54**	**582**	**2.06**	**735**	**1.83**
EU	Static	1171	**1.86**	**2006**	1.92	1220	**1.89**	1333	1.97	**1207**	**1.88**	2484	1.94	1104	**1.86**	1437	1.90
	Dynamic	**919**	2.12	2076	**1.68**	**1172**	1.97	**1073**	**1.73**	1226	2.04	**1964**	**1.44**	**967**	2.03	**1279**	**1.85**
GL5	Static	**463**	3.07	1197	4.38	**635**	3.64	789	4.25	**550**	3.27	1528	4.32	479	3.10	731	3.68
	Dynamic	499	**2.74**	**1027**	**2.82**	655	**3.08**	**581**	**2.46**	661	**2.96**	**1147**	**2.30**	**472**	**2.46**	**684**	**2.67**
AVG	Static	711	2.42	1533	2.92	858	2.62	1010	2.93	**787**	2.47	1920	2.91	707	2.42	1001	2.66
	Dynamic	**631**	**2.31**	**1292**	**1.98**	**781**	**2.24**	**749**	**1.96**	811	**2.28**	**1431**	**1.72**	**643**	**2.17**	**863**	**2.08**

Table 3: Runtime (in ms) and work, lower is better, for a concurrent BFS algorithm using the 2D LaW queue with either statically $Win_{width} = 1024$, or using the dynamic controller. Thread workloads (columns) defined in Table 1 and the graphs (rows) in Table 2. The last column is the geometric mean of all worklaods, and similarly the last row is the geometric mean over all graphs.

BFS is a representative example of a parallel algorithm that uses relaxed queues as a shared work-list, where balancing throughput and ordering accuracy is critical. In such algorithms, increasing relaxation may improve scalability but can reduce work efficiency due to redundant or out-of-order processing [14], [17]. Our goal is to assess whether dynamic control over relaxation can yield end-to-end benefits in this more realistic scenario.

The benchmark uses a relaxed FIFO queue as a work-list [17], and is analogous to how Dijkstra's algorithm is parallelized with relaxed priority queues [14], [20]. We compare the performance of the LaW queue with the dynamic controller to the static 2D queue across a range of workloads. These workloads, shown in Table 1, include constant, increasing, and decreasing thread counts, as well as real-world load traces, shown in Figure 7. Each workload runs for half a second before repeating. We measure both total runtime and total *work*, defined as the number of successful enqueues divided by the number of uniquely enqueued items (vertices).

Relaxed priority queues have been shown to perform very well for parallel SSSP over high-diameter sparse graphs [14]–[16], [19], so we focus on such graphs. The graphs are summarized in table 2. Firstly, we use the EU and USA road graphs [37], which are commonly used road graphs [14], [16], [17]. Second, we use the GL5 kNN graph [38], [39], whose irregular nature made it a difficult graph for many participants in the 2025 FastCode Pogramming Challenge on parallel SSSP [19].

Table 3 shows the runtime and work across the selected graphs and workloads, where all results are averaged over 100 runs. We searched power of 2 combinations for the optimal *width* and *depth* configuartion for the static queue, and found that $Win_{width} = 1024$ and $Win_{depth} = 8$ had the best geometric mean (gmean) runtime across all graphs and workloads. The dynamic queue therefore also used $Win_{depth} = 8$, and its contention target T was chosen so that the average *width* during the constant workload had a gmean of 1024 across the three graphs, which was 0.012.

The results show that the dynamic controller in the majority of these benchmarks reduce runtime, with the gmean across all graphs and tests being 13% faster and 28% more work-efficient. Furthermore, the dynamic LaW is faster in 16, and more work-efficient in 17, out of the 21 benchmarks. The dynamic relaxation is the most beneficial in the *Lin. Inc.* and *Exp. Inc.* workloads, with a gmean speedup of 35% and 34% respectively. On the contrary, its worst workload is the *Lin. Dec.*, where it has a 3% gmean slowdown, the only workload with a gmean slowdown.

The impact of relaxation on work efficiency in concurrent BFS remains poorly understood. Our results indicate that the early phase of execution plays a critical role where large relaxation errors at the start can compound over time, leading to significant work inefficiencies. Unlike relaxed priority queues, relaxed FIFO queues lack self-stabilizing ordering mechanisms [14]–[16], making them especially sensitive to early misordering. This may explain why the dynamic queue outperforms the static design in the *Const.* workload, with a lower gmean of work, and why increasing workloads yield better results than decreasing ones.

In summary, using our elastic LaW queue with the dynamic controller outperforms a statically configured 2D queue in the BFS benchmark with varying levels of parallelism. A key advantage of this approach is that it eliminates the need to know the number of accessing threads in advance, instead only a contention target must be specified. This target is a single tunable parameter that can also be adjusted at runtime. As a result, we believe our elastic designs, together with a dynamic controller, are better suited for integration into long-running systems—where workload characteristics may change over time—than static designs.

6 Related Work

One of the earliest uses of relaxed data structures is from 1993 by Karp and Zhang [42]. However, it was not until more recently that relaxed data structures garnered standalone interest as a promising technique to boost parallelism [5]. Relaxed designs have shown exceptional throughput on highly parallel benchmarks [7], [8], [14], [43], have proven suitable in heuristics for graph algorithms [14], [15], [44], and been theoretically analyzed in e.g. [6], [45]–[47].

Henzinger et al. formalized *quantitative relaxation* [6] to define relaxed data structures with a rank error bound. They also introduce the relaxed k-segment stack, which builds upon the earlier relaxed k-FIFO queue [9]. 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 the *Lateral*, as their k-designs can be modeled within the 2D framework.

Rather than deterministically bounding rank error, some designs achieve better performance by only giving probabilistic error guarantees [8], [14], [17], [45]. The MultiQueue is a relaxed priority queue that has proven effective in for example shortest-path graph algorithms [14]–[16], [19]. It enqueues items into random sub-queues and dequeues the highest-priority item from a random choice of two sub-queues. Similar strategies have been applied to FIFO queues [8], [17] and stacks. The probabilistic rank error guarantees of the MultiQueue have been analyzed both with a strong potential argument [45], [47] and with Markov chains [48], but its errors are still not completely understood.

The SprayList [34] is another probabilistically relaxed priority queue. Experimentally, the SprayList is outperformed by the MultiQueue [14] but it is, to the best of our knowledge, the only design in the literature that can reconfigure relaxation well during run-time, which it does by adjusting thread-local parameters based on contention.

The CA-PQ [35] is another relaxed priority queue, again outperformed by the MultiQueue [14], which can be seen as elastic, as threads toggle between synchronizing with the global state and working locally, depending on contention. They use a simple thread-local controller, similar to the one in the conference version of this paper [1], which works well because (1) synchronization is toggled per thread, and (2) only on/off decisions are made. In contrast, our shared controller from Section 5.2 better supports global configurations such as *Win* dimensions, and can prevent divergence in per-thread settings when the configuration space is larger.

Relaxed priority queues have been widely used to parallelize single-source shortest path (SSSP) graph traversals [14]–[16], [19], [35]. In the recent FastCode Programming Challenge at PPoPP 2025 [49], the winning contribution in the parallel SSSP track used the MultiQueue [14] to achieve strong results on sparse input graphs [19]. Strict and relaxed FIFO queues have also been proposed for parallel BFS [17], [50], though their performance has yet to surpass level-synchronous algorithms [51]. The work-efficiency of relaxed queues in such graph traversals remains largely unstudied and is an interesting open question.

7 Conclusion

We have presented the concept of elastic relaxation for concurrent data structures, and extended the 2D framework to incorporate elasticity. Using the *Lateral* component, we proposed the simple LaW designs and more involved LpW designs. If incorporating the *Lateral* into designs outside the 2D framework, we recommend the LaW paradigm for its simplicity and efficiency, but recognize that the LpW design might be more suitable for some data structures, such as the 2D counter. Our implementations offer worst-case bounds and match state-of-the-art performance during periods of constant relaxation, while also supporting dynamic reconfiguration of relaxation at runtime. We introduced a lightweight dynamic controller for relaxation, based on the exponential moving average of per-thread CAS contention, which efficiently trades relaxation for latency in real time. Its performance was evaluated both over time in a producer-consumer benchmark and in a BFS macrobenchmark, where it demonstrated improved throughput and work efficiency in several dynamic workloads compared to a statically relaxed queue. We believe elasticity to be essential for relaxed data structures to become practically viable, and see this paper as a first step in that direction.

As future work, we find it of interest to incorporate elastic relaxation into other data structures, such as relaxed priority queues. As relaxed priority queues have already proven state-of-the-art when it comes to parallel SSSP on some graphs, we find it of interest to see if elastic relaxation could help further increase their performance in similar graph traversal benchmarks.

References

[1] K. von Geijer and P. Tsigas, "How to relax instantly: Elastic relaxation of concurrent data structures," in *Euro-Par 2024: Parallel Processing*. Cham: Springer Nature Switzerland, 2024, pp. 119–133.

[2] ——, "Artifact of the paper: Elastic Relaxation of Concurrent Data Structures." [Online]. Available: https://doi.org/10.5281/zenodo.11547062

[3] M. Herlihy, N. Shavit, V. Luchangco, and M. Spear, *The Art of Multiprocessor Programming*. Elsevier Science, 2020.

[4] F. Ellen, D. Hendler, and N. Shavit, "On the inherent sequentiality of concurrent objects," *SIAM Journal on Computing*, vol. 41, no. 3, pp. 519–536, 2012.

[5] N. Shavit, "Data structures in the multicore age," *Comm. ACM*, vol. 54, no. 3, p. 76–84, 2011.

[6] T. A. Henzinger, C. M. Kirsch, H. Payer, A. Sezgin, and A. Sokolova, "Quantitative relaxation of concurrent data structures," *SIGPLAN Not.*, vol. 48, no. 1, p. 317–328, 2013.

[7] A. Rukundo, A. Atalar, and P. Tsigas, "Monotonically Relaxing Concurrent Data-Structure Semantics for Increasing Performance: An Efficient 2D Design Framework," in *33rd International Symposium on Distributed Computing (DISC 2019)*, vol. 146. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, pp. 31:1–31:15.

[8] A. Haas, M. Lippautz, T. A. Henzinger, H. Payer, A. Sokolova, C. M. Kirsch, and A. Sezgin, "Distributed queues in shared memory: Multicore performance and scalability through quantitative relaxation," in *Proceedings of the ACM International Conference on Computing Frontiers*. ACM, 2013.

[9] C. M. Kirsch, H. Payer, H. Röck, and A. Sokolova, "Performance, Scalability, and Semantics of Concurrent FIFO Queues," in *Algorithms and Architectures for Parallel Processing*. Springer Berlin Heidelberg, 2012, vol. 7439, pp. 273–287.

[10] M. Wimmer, J. Gruber, J. L. Träff, and P. Tsigas, "The lock-free k-lsm relaxed priority queue," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2015, p. 277–278.

[11] J. Derrick, B. Dongol, G. Schellhorn, B. Tofan, O. Travkin, and H. Wehrheim, "Quiescent consistency: Defining and verifying relaxed linearizability," in *FM 2014: Formal Methods*. Springer, 2014, pp. 200–214.

[12] M. P. Herlihy and J. M. Wing, "Linearizability: a correctness condition for concurrent objects," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 3, pp. 463–492, 1990.

[13] G. Kappes and S. V. Anastasiadis, "A family of relaxed concurrent queues for low-latency operations and item transfers," *ACM Trans. Parallel Comput.*, vol. 9, no. 4, 2022.

[14] M. Williams, P. Sanders, and R. Dementiev, "Engineering MultiQueues: Fast Relaxed Concurrent Priority Queues," in *29th Annual European Symposium on Algorithms*, vol. 204. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, pp. 81:1–81:17.

[15] A. Postnikova, N. Koval, G. Nadiradze, and D. Alistarh, "Multi-queues can be state-of-the-art priority schedulers," in *Proceedings of the 27th ACM Symposium on Principles and Practice of Parallel Programming*. ACM, 2022, p. 353–367.

[16] G. Zhang, G. Posluns, and M. C. Jeffrey, "Multi bucket queues: Efficient concurrent priority scheduling," in *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 113–124.

[17] K. von Geijer, P. Tsigas, E. Johansson, and S. Hermansson, "Balanced allocations over efficient queues: A fast relaxed fifo queue," in *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '25. New York, NY, USA: ACM, 2025, p. 382–395.

[18] T. Kraska, "Towards instance-optimized data systems, keynote," *2021 International Conference on Very Large Data Bases*, 2021.

[19] M. D'Antonio, K. von Geijer, T. S. Mai, P. Tsigas, and H. Vandierendonck, "Relax and don't stop: Graph-aware asynchronous sssp," in *Proceedings of the 1st FastCode Programming Challenge*, ser. FCPC '25. New York, NY, USA: ACM, 2025, p. 43–47.

[20] H. Rihani, P. Sanders, and R. Dementiev, "Multiqueues: Simple relaxed concurrent priority queues," in *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, 2015, p. 80–82.

[21] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Trans. Comput.*, vol. 28, no. 9, p. 690–691, Sep. 1979.

[22] G. L. Peterson and J. E. Burns, "Concurrent reading while writing II: The multi-writer case," in *28th Annual Symposium on Foundatiouns of Computer Science*. IEEE, 1987, pp. 383–392.

[23] M. M. Michael and M. L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," in *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, 1996, pp. 267–275.

[24] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer's Manual*, Intel Corporation, 2022, combined Volumes 1-5. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

[25] A. Rudén and L. Andersson, "Relaxed priority queue & evaluation of locks," Master's thesis, Chalmers University of Technology and University of Gothenburg, Gothenburg, Sweden, Jun. 2023. [Online]. Available: https://gupea.ub.gu.se/handle/2077/78919

[26] A. Morrison and Y. Afek, "Fast concurrent queues for x86 processors," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2013, p. 103–112.

[27] C. Yang and J. Mellor-Crummey, "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*. ACM, 2016.

[28] R. Treiber, "Systems programming: Coping with parallelism," International Business Machines Incorporated, Thomas J. Watson Research Center, Tech. Rep., 1986.

[29] M. M. Michael, "Cas-based lock-free algorithm for shared deques," in *Euro-Par 2003 Parallel Processing*. Springer Berlin Heidelberg, 2003, pp. 651–660.

[30] K. von Geijer and P. Tsigas, "How to relax instantly: Elastic relaxation of concurrent data structures," 2024, arXiv:2403.13644.

[31] T. David, R. Guerraoui, and V. Trigonakis, "Asynchronized concurrency: The secret to scaling concurrent search data structures," *SIGARCH Comput. Archit. News*, vol. 43, no. 1, p. 631–644, 2015.

[32] K. Fraser, "Practical lock-freedom," Ph.D. dissertation, University of Cambridge, 2003.

[33] D. Hendler, N. Shavit, and L. Yerushalmi, "A scalable lock-free stack algorithm," *Journal of Parallel and Distributed Computing*, vol. 70, no. 1, pp. 1–12, 2010.

[34] D. Alistarh, J. Kopinsky, J. Li, and N. Shavit, "The spraylist: A scalable relaxed priority queue," *SIGPLAN Not.*, vol. 50, no. 8, p. 11–20, 2015.

[35] K. Sagonas and K. Winblad, "The Contention Avoiding Concurrent Priority Queue," in *Languages and Compilers for Parallel Computing*, ser. Lecture Notes in Computer Science, C. Ding, J. Criswell, and P. Wu, Eds. Cham: Springer International Publishing, 2017, pp. 314–330.

[36] K. Åström and T. Hägglund, *PID Controllers: Theory, Design, and Tuning*. ISA - The Instrumentation, Systems and Automation Society, 1995.

[37] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, Dec. 2011.

[38] Y. Wang, S. Yu, L. Dhulipala, Y. Gu, and J. Shun, "Geograph: A framework for graph processing on geometric data," *SIGOPS Oper. Syst. Rev.*, vol. 55, no. 1, p. 38–46, Jun. 2021.

[39] Y. Zheng, L. Liu, L. Wang, and X. Xie, "Learning transportation mode from raw gps data for geographic applications on the web," in *Proceedings of the 17th International Conference on World Wide Web*, ser. WWW '08. New York, NY, USA: ACM, 2008, p. 247–256.

[40] J. Pittman, R. "Outpouring of searches for the late michael jackson," *Official Google Blog*, Jun. 2009, blog post. [Online]. Available: https://googleblog.blogspot.com/2009/06/outpouring-of-searches-for-late-michael.html

[41] G. Urdaneta, G. Pierre, and M. van Steen, "Wikipedia workload analysis for decentralized hosting," *Comput. Netw.*, vol. 53, no. 11, p. 1830–1845, Jul. 2009.

[42] R. M. Karp and Y. Zhang, "Randomized parallel algorithms for backtrack search and branch-and-bound computation," *J. ACM*, vol. 40, no. 3, p. 765–789, 1993.

[43] A. Haas, T. Hütter, C. M. Kirsch, M. Lippautz, M. Preishuber, and A. Sokolova, "Scal: A benchmarking suite for concurrent data structures," in *Networked Systems*. Springer, 2015, pp. 1–14.

[44] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, p. 456–471.

[45] D. Alistarh, T. Brown, J. Kopinsky, J. Z. Li, and G. Nadiradze, "Distributionally Linearizable Data Structures," *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, pp. 133–142, 2018.

[46] D. Hendler, A. Khattabi, A. Milani, and C. Travers, "Upper and Lower Bounds for Deterministic Approximate Objects," *2021 IEEE 41st International Conference on Distributed Computing Systems*, vol. 00, pp. 438–448, 2021.

[47] D. Alistarh, J. Kopinsky, J. Li, and G. Nadiradze, "The power of choice in priority scheduling," in *Proceedings of the ACM Symposium on Principles of Distributed Computing*. ACM, 2017, p. 283–292.

[48] S. Walzer and M. Williams, "A simple yet exact analysis of the multiqueue," 2024.

[49] *FCPC '25: Proceedings of the 1st FastCode Programming Challenge*. New York, NY, USA: ACM, 2025.

[50] M. A. Hassaan, M. Burtscher, and K. Pingali, "Ordered vs. unordered: a comparison of parallelism and work-efficiency in irregular algorithms," in *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '11. New York, NY, USA: ACM, 2011, p. 3–12.

[51] S. Beamer, K. Asanović, and D. Patterson, "Direction-optimizing breadth-first search," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Washington, DC, USA: IEEE Computer Society Press, 2012.

Kåre von Geijer received his M.Sc. degree in Engineering Mathematics from Lund University, Sweden, in 2022. He is currently a Ph.D. Candidate within the Computer Networks and Systems Division at Chalmers University of Technology, Sweden. His research centers around relaxed semantics for concurrent data structures, but he is also interested in parallel graph algorithms, task scheduling for parallel algorithms, lock-freedom, and algorithm engineering.

Philippas Tsigas received the B.Sc. degree in mathematics and the Ph.D. degree in computer engineering and informatics from the University of Patras, Greece. He was at the National Research Institute for Mathematics and Computer Science, Amsterdam, The Netherlands (CWI), and at the Max-Planck Institute for Computer Science, Saarbrücken, Germany, before. At present, he is a professor in the Department of Computing Science at Chalmers University of Technology, Sweden. His research interests include concurrent data structures and algorithmic libraries for multiprocessor and many-core systems, communication, and synchronization in parallel systems, power aware computing, fault-tolerant computing, autonomic computing, and scalable data streaming.