Monotonically Relaxing Concurrent Data-Structure Semantics for Increasing Performance: An Efficient 2D Design Framework

Adones Rukundo

Chalmers University of Technology Department of Computer Science and Engineering, Sweden
adones@chalmers.se

Aras Atalar

Chalmers University of Technology Department of Computer Science and Engineering, Sweden
aaras@chalmers.se

Philippas Tsigas

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

Abstract

There has been a significant amount of work in the literature proposing semantic relaxation of concurrent data structures for improving scalability and performance. By relaxing the semantics of a data structure, a bigger design space, that allows weaker synchronization and more useful parallelism, is unveiled. Investigating new data structure designs, capable of trading semantics for achieving better performance in a monotonic way, is a major challenge in the area. We algorithmically address this challenge in this paper.

We present an efficient, lock-free, concurrent data structure design framework for *out-of-order* semantic relaxation. We introduce a new two dimensional algorithmic design, that uses multiple instances of a given data structure. The first dimension of our design is the number of data structure instances operations are spread to, in order to benefit from parallelism through disjoint memory access; the second dimension is the number of consecutive operations that try to use the same data structure instance in order to benefit from data locality. Our design can flexibly explore this two-dimensional space to achieve the property of monotonically relaxing concurrent data structure semantics for better performance within a tight deterministic *relaxation* bound, as we prove in the paper.

We show how our framework can instantiate lock-free *out-of-order* queues, stacks, counters and dequeues. We provide implementations of these *relaxed* data structures and evaluate their performance and behaviour on two parallel architectures. Experimental evaluation shows that our two-dimensional design significantly outperforms the respected previous proposed designs with respect to scalability and performance. Moreover, our design increases performance monotonically as relaxation increases.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis; Theory of computation → Design and analysis of algorithms; Theory of computation → Concurrency; Theory of computation → Concurrent algorithms; Computing methodologies → Concurrent algorithms

Keywords and phrases Lock Free, Concurrency, Semantics Relaxation, Data Structures

Digital Object Identifier 10.4230/LIPIcs.DISC.2019.31

Related Version An extended version of the paper is available at `https://arxiv.org/abs/1906.07105`.

Funding SIDA/Bright Project (317) under the Makerere-Sweden bilateral research programme 2015-2020, Mbarara University of Science and Technology, and, Swedish Research Council (Vetenskapsrådet) Under Contract No.: 2016-05360; "Models and Techniques for Energy-Efficient Concurrent Data Access Designs".

1 Introduction

Concurrent data structures allow operations to access the data structure concurrently, which require synchronised access to guarantee consistency with respect to their sequential semantics [9, 10]. The synchronisation of concurrent accesses is generally achieved by guaranteeing some notion of atomicity, where, an operation appears to occur at a single instant between its invocation and its response. A concurrent data structure is typically designed around one or more synchronisation *access* points, from where threads compute, consistently, the current state of the data structure. Synchronisation is vital to achieving consistency and cannot be eliminated [5]. Whereas this is true, synchronization might generate *contention* in memory resources hurting scalability and performance.

The necessity of reducing contention at the synchronisation *access* points, and consequently improving scalability, is and has been a major focus for concurrent data structure researchers. Techniques like; elimination [18, 31], combining [32], dynamic elimination-combining [6] and back-off strategies have been proposed as ways to improve scalability. To address, in a more significant way, the challenge of scalability bottlenecks of concurrent data structures, it has been proposed that the semantic legal behaviour of data structures should be extended [29]. This line of research has led to the introduction of an extended set of weak semantics including; weak internal ordering, weakening consistency and semantic *relaxation*.

One of the main definition of semantic *relaxation* proposed and used in the literature is *k-out-of-order* [1, 2, 15, 19, 24, 26, 33, 35]. *k-out-of-order* semantics allow operations to occur out of order within a given k bound, e.g. a pop operation of a *k-out-of-order* stack can remove any item among the k topmost stack items. By allowing a *Pop* operation to remove any item among the k topmost stack items, the semantics do not anymore impose a single *access* point. Thus, by *relaxing* the stack semantics, we allow for potentially more efficient stack designs with reduced synchronisation overhead, which is the motivation for concurrent data structure semantics *relaxation*.

Relaxation can be exploited to achieve improved parallelism by increasing the number of disjoint *access* points, or by increasing thread local data processing. Disjoint *access* is popularly achieved by distributing operations over multiple instances of a given data structure [2, 14, 15, 24]. On the other hand, the locality is generally achieved through binding single thread *access* to the same memory location for specific operations [13, 14, 35].

In this paper, we introduce an efficient two-dimensional algorithmic design framework, that uses multiple instances (*sub-structures*) of a given data structure as shown in Figure 1. The first dimension of the framework is the number of *sub-structures* operations are spread to, in order to benefit from parallelism through disjoint *access* points; the second dimension

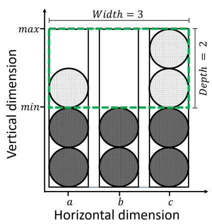


Figure 1 An illustration of our 2D design using a Stack as an example. There are three *sub-stacks* a, b and c. k is proportional to the area of the green dashed rectangle in which operations are bounded to occur. a can be used for both *Push* and *Pop*. b can be used for *Push* but not for *Pop*. c can be used for *Pop* but not for *Push*.

is the number of consecutive operations that can occur on the same *sub-structure* in order to benefit from data locality. We use two parameters to control the dimensions; *width* for the first dimension (horizontal) and *depth* for the second dimension (vertical).

A thread can operate on a given *sub-structure* for as long as a set of conditions hold (*validity*). Validity can be that valid *sub-structures* do not exceed (*max*) or go below (*min*), a given operation count threshold, as depicted by the dashed green rectangle in Figure 1. Validity conditions make *sub-structures* valid or invalid for a given operation. This implies that threads have to search for a valid *sub-structure*, increasing operation cost (latency). Our framework overcomes this challenge by limiting the number of *sub-structures* and allowing a thread to operate on the same *sub-structure* consecutively for as long as the validity conditions hold. *Max* and *min* can be updated if there are no valid *sub-structures*. We show algorithmically that the validity conditions provide for an efficient, tenable and tunable *relaxation* behaviour, described by tight deterministic *relaxation* bounds.

Our design framework can be used to extend existing lock-free data structure algorithms to derive *k-out-of-order* semantics for the given data structure. This can be achieved with minimal modifications to the data structure algorithm as we later show in this paper. Using our framework, we extend existing lock-free algorithms to derive lock-free *k-out-of-order* stacks, queues, dequeue and counters. Detailed implementation, correctness and performance analysis is also provided. Experimental evaluation shows that the derived data structures significantly outperform all previous data structure implementations of same category.

The rest of the paper is structured as follows. In Section 2 we discuss literature related to this work. We present the $2D$ framework and derived 2D algorithms in Section 3. We prove correctness and linearization bounds in Section 4. An experimental evaluation is discussed in Section 5 and the paper concludes in Section 6.

2 Related Work

Recently, data structure semantic *relaxation* has attracted the attention of researchers, as a promising direction towards improving concurrent data structures' scalability [19, 29, 33]. It has also been shown that small changes on the semantics of a data structure can have a significant effect on the computation power of the data structure [30]. The interest in semantic *relaxation* is largely founded on the ease of use and understanding. One of the main definition of semantic *relaxation* proposed and used is *k-out-of-order*.

Using the *k-out-of-order* definition, a segmentation technique has been proposed in [1], later revisited in [19] realizing a *relaxed* Stack (*k-Stack*) and FIFO Queue (*Q-segment*) with *k-out-of-order* semantics. The technique involves a linked-list of memory segments with k number of indexes on which an item can be added or removed. The stack items are accessed through the topmost segment, whereas the queue has a tail and head segment from which *Enqueue* and *Dequeue* can occur respectively. Segments can be added and removed. *Relaxation* is only controlled through varying the number of indexes per segment. As discussed in Section 1, increasing the number of indexes increases operation latency and later becomes a performance bottleneck. This limits the performance benefits of the technique to a small range of *relaxation* values.

Also, load balancing together with multiple queue instances (*sub-queues*) has been used to design a *relaxed* FIFO queue (*lru*) with *k-out-of-order* semantics [15]. Each *sub-queue* maintains two counters, one for *Enqueue* another for *Dequeue*, while two global counters, one for *Enqueue* another for *Dequeue* maintain the total #operations for all *sub-queues*. The global counters are used to calculate the expected #operations on the last-recently-used *sub-queue*. Threads can only operate on the least-recently-used *sub-queue*. This implies that for every operation threads must synchronise on the global counter, making it a sequential bottleneck. Moreover, threads have to search for the last-recently-used *sub-queue* leading to latency increase. Randomisation has also been used to balance load between multiple *sub-structures*, leading to *relaxed* designs such as MultiQueues and MultiCounters [2, 24].

The proposed *relaxation* techniques, mentioned above, apply *relaxation* in one dimension, i.e, increase disjoint *access* points to improve parallelism and reduce contention. However, this also increases operation latency due to increased *access* points to select from. Without a remedy to this downside, the proposed techniques cannot provide monotonic *relaxation* for better performance. Other *relaxed* data structures studied in the literature include priority queues [4, 24, 35]. Apart from semantic *relaxation*, other design strategies for improving scalability have been proposed including; elimination [6, 18, 23, 31], combining [32], internal weak ordering [11], and local linearizability [14]. However, these strategies have not been designed to provide bounded out of order semantic *relaxation*.

Elimination implements a collision path on which different concurrent operations try to collide and cancel out, otherwise, they proceed to access the central structure [18]. Combining, on the other hand, allows operations from multiple threads to be combined and executed by a single thread without the other threads contending on the central structure [12, 17]. However, their performance depends on the specific workload characteristics. Elimination mostly benefits symmetric workloads, whereas combining mostly benefits asymmetric workloads. Furthermore, the central structure sequential bottleneck problem still persists.

Weak internal ordering has been proposed and used to implement a timestamped stack (*TS-Stack*) [11], where *Push* timestamps each pushed item to mark the item's precedence order. Each thread has its local buffer on which it performs *Push* operations. However, *Pop* operations pay the cost of searching for the latest item. In the worst case, *Pop* operations might contend on the same latest item if there are no concurrent *Push* operations. This leads to search retries, especially for workloads with higher *Pop* rates than *Push* ones.

Local linearizability has also been proposed for concurrent data structures such as; FIFO queues and Stacks [14]. The technique relies on multiple instances of a given data structure. Each thread is assigned an instance on which it locally linearizes all its operations. Operations: *Enqueue* (FIFO queue) or *Push* (Stack) occur on the assigned instance for a given thread, whereas, *Dequeue* or *Pop* can occur on any of the available instances. With *Dequeue* or *Pop* occurring more frequently, contention quickly builds as threads try to access remote buffers. The threads also lose the locality advantage while accessing remote buffers, cancelling out the caching advantage especially for single access data structures such as the Stack [7, 16, 28].

3 2D Framework

In this section, we describe our 2*D* design framework and show how it can be used to extend existing data structure designs to derive *k-out-of-order relaxed* semantics. Such data structures include; stacks, FIFO queues, counters and dequeues.

The 2D framework uses multiple copies (*sub-structures*) of the given data structure as depicted in Figure 1. Threads can operate on any of the *sub-structures* following the fixed maximum *max* and minimum *min* operation count threshold. Herein, *operation* refers to the process that updates the data structure state by adding (*Put*) or removing (*Get*) an item (*Push* and *Pop* respectively for the stack example). Each *sub-structure* holds a counter (*sub-count*) that counts the number of local successful operations.

Algorithm 1 Window Coupled (2Dc).

```
Struct Descriptor Des
    *item;
    count;                                        ▷ sub-count
    version;
Struct Window Win
    max;
    version;
Function Window(Op,index,cont)
    IndexSearch = Random = notempty=0; LWin = Win;
    if cont==True then
        index=RandomIndex(); cont=False;
    end
    while True do
        if IndexSearch == width then
            SHIFTWINDOW();
        end
        Des = Array[index];                       ▷ Read descriptor
        if Op == put ∧ Des.count < Win.max then
            return {Des,index};
        else if Op == get ∧ Des.count > (Win.max - depth) then
            return {Des,index};
        else if LWin == Win then
            HOP();
        else
            LWin = Win; IndexSearch = 0;
        end
    end
Macro SHIFTWINDOW()
    if Op == get ∧notempty==0 then
        return {Des,index};
    end
    if LWin == Win then
        if Op == put then
            NWin.max = LWin.max + ShiftUp;
        else if Op == get ∧ Win.max > depth then
            NWin.max = LWin.max - ShiftDown;
        end
        NWin.version = LWin.version + 1;
        CAS(Win,LWin,NWin);
    end
    LWin = Win; IndexSearch = 0;
Macro HOP()
    if Random<2 then
        index=RandomIndex(); Random+=1;
    else
        if Op == get ∧ Des.item!=NULL then
            notempty=1;
        end
        if index == width - 1 then
            index=0;
        else
            index += 1;
        end
        IndexSearch += 1;
    end
```

A combination of *max*, *min* and *#sub-structures*, form a logical count period, we refer to it as *Window*, depicted by the dashed green rectangle in Figure 1. *Window* defines the maximum (Win_{max}) and minimum (Win_{min}) operation count threshold for all *sub-structures*, for a given period. This implies that, for a given period, a *sub-structure* can be valid or invalid as exemplified in Figure 1, and a *Window* can be full or empty. The *Window* is *full* if all *sub-structures* have maximum operations (*sub-count* $= Win_{max}$), *empty*, if all *sub-structures* have minimum operations (*sub-count* $= Win_{min}$). The *Window* is defined by two parameters; *width* and *depth*. *width* = *#sub-structures*, and *depth* $= Win_{max} - Win_{min}$.

To validate a *sub-structure*, its *sub-count* is compared with Win_{min} or Win_{max}; either *sub-count* $\geq Win_{min}$ or *sub-count* $< Win_{max}$. If the given *sub-structure* is invalid, the thread has to *hop* to another *sub-structure* until a valid *sub-structure* is found (validity is operation specific as we discuss later). If a thread cannot find a valid *sub-structure*, then, the *Window* is either full or empty. The thread will then, either increment or decrement both Win_{min} and Win_{max}, the process we refer to as, *Window* $shifting$. A *Window* can $shift$ up or down, and is controlled by $shift^{up}$ or $shift_{down}$ values respectively, where, $0 < shift^{up}, shift_{down} \leq$ *depth*. Win_{min} and Win_{max} can only be incremented or decremented by a given $shift$ value. $shift^{up}$ and $shift_{down}$ can be configured differently to optimize for different workloads.

We define two types of *windows*: *WinCoupled* (*2Dc*) and *WinDecoupled* (*2Dd*).

- ***WinCoupled*:** couples both *Put* and *Get* to share the same *Window* and *sub-count* for each *sub-structure*. A successful *Put* increments whereas, a successful *Get* decrements the given *sub-count*. On a full *Window*, *Put* increments Win_{max} *shifting* the *Window* up ($shift^{up}$), whereas, on an empty *Window*, *Get* decrements Win_{max}, *shifting* the *Window* down ($shift_{down}$). *WinCoupled* resembles elimination [18], only that here, we cancel out operation counts for matching *Put* and *Get* on the same *sub-structure* within the same *Window*. Just like elimination reduces joint *access* updates, *WinCoupled* reduces *Window* $shift$ updates.
- ***WinDecoupled*:** decouples *Put* and *Get* and assigns them independent *windows*. Also, an independent *sub-count* is maintained for *Put* or *Get*, on each *sub-structure*. Unlike *WinCoupled*, both operations always increment their respective *sub-count* on a successful operation and Win_{max} on a full *Window*. This implies that both *sub-count* and *Window* counters are always increasing.

Data structures such as FIFO queues with disjoint *access* for *Put* and *Get*, can benefit more from the *WinDecoupled* disjoint *Window* design. Whereas, data structures such as stacks with joint *access*, can benefit more from the *WinCoupled* operation count cancelling design. Here, we present *WinCoupled* due to its interesting operation count cancelling and refer the reader to our extended version [27] for the *WinDecoupled* presentation.

In Algorithm 1, we present the algorithmic steps for *WinCoupled*. Recall, $width = \#$*sub-structures* and $depth = Win_{max} - Win_{min}$. Each *sub-structure* is uniquely identified by an index, which holds information including a pointer to the *sub-structure*, *sub-count* counter, and a version number (line 1-4). The version number is to avoid ABA related issues. Using a wide `CAS`, we update the index information in a single atomic step.

To perform an operation, the thread has to search and select a valid *sub-structure* within a *Window* period. Starting from the search start index, the thread stores a copy of the *Window* locally (line 9) which is used to detect *Window* $shifts$ while searching (line 22,32). During the search, the thread validates each *sub-structure* count against Win_{max} (line 18,20). If no valid *sub-structure* is found, Win_{max} is updated atomically, $shifting$ the *Window* up or down (line 39). *Put* increments Win_{max} to $shift$ the *Window* up (line 34), whereas, *Get* decrements Win_{max} to $shift$ the *Window* down (line 36). Before *Window* $shifting$ or index *hopping*, the thread must confirm that the *Window* has not yet shifted (line 32 and 22 respectively). For every *Window* shift during the search, the thread restarts the search with the new *Window* values (line 25,41).

If a valid index is selected, the respective descriptor state and index are returned (line 19,21). The thread can then proceed to try and operate on the given *sub-structure* pointed to by the index descriptor. As an *emptiness check*, the *Window* search can only return an empty *sub-structure* (line 29), if during the search, all *sub-structures* where empty ($NULL$ pointer). Using the *Window* parameters, *width*, and *depth*, we can tightly bound the *relaxation* behaviour of derived 2D data-structures as discussed later in Section 4.

3.1 Deriving 2D Data structures

Our framework can be used to extend existing algorithms to derive *k-out-of-order* data structures. Using *WinCoupled* we derive a *2Dc-Stack* and a *2Dc-Counter*, whereas by using *WinDecoupled*, we derive a *2Dd-Stack*, a *2Dd-Queue*, a *2Dd-Deque* and a *2Dd-Counter*. The base algorithms include but not limited to; Treiber's stack [34], MS-queue [21] and Deque [20] for Stack, FIFO Queue and Deque respectively. As an example, we shall discuss the *2Dc-Stack* due to its simplicity and refer the reader to our extended version [27] for the other algorithmic implementations.

Algorithm 2 *2Dc-Stack.*

```
Function Push(NewItem)
    while True do
        {Des,Index} = Window(push,index,cont);
        NewItem.next = Des.item;
        NDes.item = NewItem;
        NDes.count = Des.count + 1;
        NDes.version = Des.version;
        if CAS(Array[Index],Des,NDes) then
            return 1;
        else
            cont=True;
        end
    end
```

```
Function Pop()
    while True do
        {Des,Index}=Window(pop,index,cont);
        if Des.item != NULL then
            NDes.item = Des.item.next;
            NDes.count = Des.count - 1;
            NDes.version = Des.version;
            if CAS(Array[Index],Des,NDes) then
                return Des.item;
            else
                cont=True;
            end
        else
            return Null;
        end
    end
```

As depicted in Algorithm 2, a stack has two operations: *Push* that adds an item and *Pop* that removes an item from the stack. *2Dc-Stack* is composed of multiple lock-free *sub-stacks*. Each *sub-stack* is implemented according to the Treiber's stack design, modified only to fit the *Window* design. The stack head is modified to a descriptor containing the top item pointer, operation count, and descriptor version. Note that, the descriptor is updated in a single atomic step using a wide `CAS` (line 8,21), the same way as in the Treiber's stack.

To perform an operation, a given thread obtains a *sub-stack* by performing a *Window* search (line 3,16). The thread then prepares a new descriptor based on the existing descriptor at the given index (line 4-7,18-20). Using a `CAS`, the thread tries to atomically swap the existing descriptor with the new one (line 8,21). If the `CAS` fails, the thread sets the contention indicator to true (line 11,24) and restart the *Window* search.

A successful *Push* increments whereas a *Push* decrements the operation count by one (line 6,19). Also, the topmost item pointer is updated. At this point, a *Push* adds an item whereas a *Pop* returns an item for a non-empty or $NULL$ for empty stack (line 27). Recall that the framework performs a special emptiness check before returning an empty *sub-stack*.

3.2 Optimizations

Our design framework can be tuned to optimize for; locality, contention and *hops* overhead, using the *width* and *depth* parameters.

3.2.1 Locality and Contention

To exploit locality, the thread starts its *Window* search from the previously known index on which it succeeded. This allows the thread a chance to operate on the same *sub-structure* multiple times locally, given that the *sub-structure* is valid. Working locally improves the caching behaviour, which in return improves performance especially under a NUMA execution environment with high communication cost across NUMA nodes [7, 16, 28].

A failed operation on a valid *sub-structure* signals the possibility of contention. The thread that fails on a `CAS` (Algorithm 2: line 11,24), starts the *Window* search on a randomly selected index (Algorithm 1: line 11). This reduces possible contention that might arise if the failed threads were to retry on the same *sub-structure*. Furthermore, random selection avoids contention on individual *sub-structures* by uniformly distributing the failed threads to all available *sub-structures*.

For every *Window* search, if the search start index is invalid, the thread tries a given number of random jumps (Algorithm 1: line 44), then switches to round robin (Algorithm 1: line 52) until a valid *sub-structure* is found. In our case, we use two random jumps as the optimal number for a random search basing on the power of random two choices [22]. However, this is a configurable parameter that can take any value.

We further note that contention is inversely proportional to the *width*. As a simple model, we split the latency of an operation into contention (op_{cont}) and contention-free (op_{free}) operation costs, given by $op = op_{cont}/width + op_{free}$. This means that we can increase the *width* to further reduce contention.

3.2.2 *hops* Overhead

The number of *hops* increases with an increase in *width*. This counteracts the performance benefits from contention reduction through increasing *width*, necessitating a balance between contention and *hops* reduction. Based on our simple contention model above, the performance would increase as the contention factor vanishes with the increase of *width*, but with an asymptote at $1/op_{free}$. This implies that beyond some point, one cannot really gain throughput by increasing the *width*, however, throughput would get hurt due to the increased number of *hops*. At some point as *width* increases, gains from the contention factor ($\lim_{width \to \infty} op_{cont} \to 0$) are surpassed by the increasing cost of *hops*. To avoid this, we switch to increasing *depth* instead of *width*, at the point of *width* saturation. Increasing *depth* reduces the number of *hops*. This is supported by our step complexity analysis [27] given by Theorem 1. Where $p = P(Put)$ and $\mathbb{E}(Extra) = \mathbb{E}(hop) + \mathbb{E}(shift)$.

▶ **Theorem 1.** *For a 2Dc-structure that is initialized with parameters depth, width,* $shift =$ *depth and* $p = 1/2$, $\mathbb{E}(Extra) = O(\frac{\ln width}{depth})$.

4 Correctness

In this section, we prove the correctness of our 2D derived data structures, including their *relaxation* bounds and lock freedom. Due to space constraints, we present the *2Dc-Stack* correctness proofs and only give Theorem 2 for our other derived 2*D* data structures. We refer the reader to our extended version [27] for the rest of the proofs.

Each *sub-structure* is lock-free: An operation can fail on `CAS` only if there is another successful operation. For *WinCoupled*, *Window shifting* is lock-free iff $shift < depth$, whereas it is always lock-free for *WinDecoupled*. A *Window shift* can only fail if there is another successful *shift* operation preceded by a successful *Put* or *Get*, ensuring system progress. Thus, all our derived algorithms are lock-free. Our design framework can also be used for lock based data structures.

▶ **Theorem 2.** *All our derived 2D data structures are linearizable with respect to k-out-of-order semantics for the respective data structure Semitics. Where, for 2Dd-Stack* $k = (3depth)(width{-}1)$*, for 2Dd-Queue* $k = (depth)(width{-}1)$*, for 2Dd-Deque* $k = (8depth)(width{-}1)$*, for 2Dd-Counter and 2Dc-Counter* $k = (2depth)(width - 1)$.

2Dc-Stack is linearizable with respect to the sequential semantics of *k-out-of-order* stack [19]. *2Dc-Stack Push* and *Pop* linearization points are similar to those of the original Treiber's Stack. As shown in Algorithm 2, *Pop* linearizes either by returning *NULL* (line 27) or with a successful `CAS` (line 22). *Push* linearizes with a successful `CAS` (line 9).

Relaxation can be applied method-wise and it is applied only to *Pop* operations, that is, a *Pop* pops one of the topmost *k* items. Firstly, we require some notation. The *Window* defines the number of operations allowed to proceed on any given *sub-stack*. The *Window* is shifted by the parameter *shift*, $1 \leq shift < depth$ and $width = \#sub\text{-}stacks$. For simplicity, let $shift = shift^{up} = shift_{down}$. A *Window* i (W_i) has an upper bound (W_i^{max}) and a lower bound (W_i^{min}), where $W_i^{max} = i \times shift$ and $W_i^{min} = (i \times shift) - depth$, respectively. For simplicity, let *Global* represent the current global upper bound (Win_{max}). A *Window* is active iff $W_i^{max} = Global$. The number of items of the *sub-stack* j is denoted by N_j, $1 \leq j \leq width$. To recall, the top pointer, the version number and N_j are embedded into the *descriptor* of *sub-stack* j and all can be modified atomically with a wide `CAS` instruction.

▶ **Lemma 3.** *Given that* $Global = shift \times i$*, it is impossible to observe a state(S) such that* $N_j > W_{i+1}^{max}$ *(or* $N_j < W_{i-1}^{min}$*).*

Proof. We show that this is impossible by considering the interleaving of operations. Without loss of generality, assume thread 1 (P_1) has set $Global = shift \times i$ at time $t_1\prime$. To do this, P_1 should have observed either $Global = shift \times (i-1)$ and then $N_j = W_{i-1}^{max}$ or $Global = shift \times (i+1)$ and then $N_j = W_{i+1}^{min}$. Let this observation of *Global* happen at time t_1. Consider the last successful push operation at *sub-stack* j before the state S is observed for the first time (we do not consider *Pop* operations as they can only decrease N_j to a value that is less than W_{i+1}^{max}, this case will be covered by the first item below). Assume thread 0 (P_0) sets N_j to $N_j > W_{i+1}^{max}$ in this push operation. P_0 should observe $N_j \geq W_{i+1}^{max}$ and *Global* $> W_{i+1}^{max}$. Let j be selected at time t_0. And the linearization of the operation happens at $t_0\prime > t_0$.

- If $t_0\prime < t_1$, the concerned state(S) can not be observed since *Global* cannot be changed (to $shift \times i$) after $N_j > W_{i+1}^{max}$ is observed.
- Else if $t_1\prime < t_0$, the concerned state(S) cannot be observed since the push operation cannot proceed after observing *Global* with such N_j.
- Else if $t_1 > t_0$, then P_0 cannot linearize because, this implies N_j has been modified (the difference between the value of *Global* that is observed by P_0 and then by P_1 implies this) since P_0 had read the descriptor, the version numbers would have changed since then.
- Else if $t_1 < t_0$, then this implies *Global* has been modified, since it was read by P_1, thus updating *Global* would fail, at least based on the version number. ◀

▶ **Lemma 4.** *At all times, there exist an i such that $\forall j, 1 \leq j \leq width$: $W_i^{min} \leq N_j \leq W_{i+1}^{max}$.*

Proof. Informally, the lemma states that the size (number of operations) of a *sub-stack* spans to at most two consecutive accessible *windows*. Assume that the statement is not true, then there should exist a pair of *sub-stacks* (y and z) at some point in time such that $\exists i, N_y < W_i^{min}$ and $N_z > W_{i+1}^{max}$. Consider the last *Push* at *sub-stack* z and last *Pop* at *sub-stack* y that linearize before or at the time t.

Assume thread P_0 (*Push*) sets N_z and thread P_1 (*Pop*) sets N_y. To do this, P_0 should observe $N_z \geq W_{i+1}^{max}$ and *Global* $> W_{i+1}^{max}$, let *sub-stack* z be selected at t_0. And, the linearization of the *Push* operation occurs at $t_0\prime > t_0$. Similarly, for P_1 *Pop* operation, let *sub-stack* y be selected at t_1, P_1 should have observed *Global* $\leq W_i^{min}$. And, let the *Pop* operation linearize at time $t_1\prime > t_1$. Now, we consider the possible interleavings.

- If $t_0\prime < t_1$ (or the symmetric $t_1\prime < t_0$ for which we do not repeat the arguments), then for P_1 to proceed and pop an item from *sub-stack* y, it is required that *Global* $\leq W_i^{min}$. Based on Lemma 3, this is impossible when $N_z > W_i^{max}$.
- Else if $t_1 > t_0$, then P_0 cannot linearize, because this implies that N_z has been modified (the difference between the value of *Global* that is observed by P_0 and then by P_1 implies this) since P_0 has read the *descriptor*. The version number would have changed since then.
- Else if $t_0 > t_1$, the argument above holds for P_1 too, so P_1 should fail to linearize.

Such N_z and N_y pair can not co-exist at any time. ◀

▶ **Theorem 5.** *2Dc-Stack is linearizable with respect to k-out-of-order stack semantics, where $k = (2shift + depth)(width - 1)$.*

Proof. Consider the *Push* (t_e^{push}) and *Pop* (t_e^{pop}) linearization points, that insert and remove an item e for a given *sub-stack* j respectively, where, $t_e^{pop} > t_e^{push}$. Now, we bound the maximum number of items, that are pushed after t_e^{push} and are not popped before t_e^{pop}, to obtain k. Let item e be the ${N_j}^{th}$ item from the bottom of the *sub-stack*. Consider a *Window* i such that: $W_i^{min} \leq N_j \leq W_i^{max}$.

Lemma 4 states that the sizes of the *sub-stacks* should reside in a bounded region. Relying on Lemma 4, we can deduce that at time t_e^{push}, the following holds: $\forall i : N_j \geq W_i^{min} - shift$. Similarly, we can deduce that at time t_e^{pop}, the following holds: $\forall i : N_j \leq W_i^{max} + shift$. Therefore, the maximum number of items, that are pushed to *sub-stack* j after t_e^{push} and are not popped before t_e^{pop} is at most $W_i^{max} + shift - (W_i^{min} - shift) = depth + 2shift$. We know that this number is zero for *sub-stack* j (the *sub-stack* that e is inserted) and we have $width - 1$ other *sub-stacks*. So, there can be at most $(2shift + depth)(width - 1)$ items that are pushed after t_e^{push} and are not popped before t_e^{pop}. ◀

5 Experimental Evaluation

We experimentally evaluate the performance of our derived $2D$ algorithms, in comparison to *k-out-of-order relaxed* algorithms available in the literature, and other state of the art data structure algorithms. *k-out-of-order relaxed* algorithms include; Last recently used queue (*lru*) [15], Segmented queue (*Q-segment*) and *k-Stack* [1, 19], other algorithms include; MS-queue (*MS-queue*) [21], Wait free queue (*wfqueue*) [36], Time stamped stack (*TS-Stack*) [11] and Elimination back-off stack (*Elimination*) [18]. To facilitate a detailed study,we implement three extra *relaxation* techniques following the same multiple *sub-structures* design; *Random*, *Random-C2* and *Round-Robin*. These techniques present a combination of characteristics that add value to our evaluation. In order to compare to data structures that have used such techniques in the literature, we implement general data structures for each technique as we describe in the next paragraph. We shall use *width* generally to refer to #*sub-structures* for all algorithms using the multiple *sub-structures* design.

For *Random*, a *Put* or *Get* operation selects a *sub-structure* randomly and proceeds to operate on it, whereas for *Random-C2*, a *Get* operation randomly selects two *sub-structures*, compares their items returning the most correct depending on the data structure semantics [2, 3, 24, 25]. *Put* operations time stamp items marking their time of entry. It is these timestamps that are compared to determine the precedence order among the two items. Due to the randomized distribution of operations, we expect low or no contention, no locality, no *hops* and no deterministic *k-out-of-order* relaxation bound. We derive *S-random* and *S-random-c2* stacks, *Q-random* and *Q-random-c2* queues, *C-random* and *C-random-c2* counters for both *Random* and *Random-C2* respectively.

Under *Round-Robin*, a thread selects and operates once on a *sub-structure* in a strict round-robin order following its local counter. The thread must succeed on the selected *sub-structure* before proceeding to the next. Due to retries on the same *sub-structure*, we expect contention and no *hops*. The thread operates once on each *sub-structure*, hence no locality. We derive *S-robin* stack, *Q-robin* queue, and *C-robin* counter. *Round-Robin* provides *relaxation* bounds [27], we demonstrate this using *S-robin* bound given by Theorem 6.

▶ **Theorem 6.** *S-robin is linearizable with respect to k-out-of-order stack semantics, where* $k = (2 \times \#threads - 1)(\#sub\text{-}stacks - 1)$.

To facilitate a uniform comparison, we implemented all the evaluated algorithms using the same development tools. The source code will be made publicly available.

5.1 System Description

Experiments are run on two x86-64 machines: (i) Intel Xeon E5-2687W v2 machine with 2 sockets, 8-core Intel Xeon processors each running at 3.4GHz, L2 cache = 256KB, L3 cache = 25.6MB (*Multi-S*) and (ii) Intel Xeon Phi 7290 with one 72-core processor running at 1.5GHz, L2 cache = 1024KB (*Single-S*). *Multi-S* and *Single-S* run on Ubuntu 16.04.2 LTS and CentOS Linux 7 Core Operating systems receptively. The *Multi-S* machine is used to evaluate inter-socket execution behaviour, whereas *Single-S* is used to evaluate intra-socket. Threads are pined one per core, for both machines excluding hyper-threading. Inter socket execution is evaluated through pinning the threads one per socket in round robin fashion. Threads randomly select between *Put* or *Get* with a given probability (operation rate). Memory is managed using the ASCYLIB framework SSMEM [8].

Our main goal is to achieve scalability under high contention. To evaluate this, we simulate high contention by excluding work between operations. To reduce the effect of *Get* $NULL$ returns in our results, any given algorithm is initialized with 2^{17} items. Each experiment is then run for five seconds obtaining an average of five repeats. Throughput is measured in terms of operations per second, whereas the *relaxation* behaviour (*accuracy*) is measured in terms of the error distance from the exact data structure sequential semantics [19]. The higher the distance, the lower the *accuracy*.

Our design framework is tunable, giving designers the ability to manage performance optimizations for different execution environments and workloads, within a given tight *relaxation* bound (k). This, however, calls for a multi-objective optimization model, which is beyond the scope of this paper. We instead run tuning experiments to obtain a tuned configuration for our evaluation. From our tuning experiments [27], we observe that $width = 3 \times \#threads$ provides a fair balance between *accuracy* and throughput for all $2D$ algorithms. We use this *width* configuration in the rest of the experiments.

5.2 Measuring *accuracy*

We adopt a similar methodology used in the literature [4, 24]. A sequential linked-list is run alongside the data structure being measured. For each operation *Put* or *Get*, a simultaneous insert or delete is performed on the linked-list respectively, following the exact semantics of the given data structure. A global lock is carefully placed at the data structure linearization points, locking both the linked-list and the data structure simultaneously. The lock allows only one thread to update both the data structure and the linked-list in isolation.

A given thread has to acquire the lock before it tries to linearize on any given *sub-structure*. Note that, *Window* search is independent of the lock. Items on the data structure are duplicated on the linked-list and can be identified by their unique labels. Insert operations happen at the head or tail of the list for LIFO or FIFO measurements respectively. A delete operation searches for the given item deletes it and returns its distance from the head (*error-distance*). For counter measurements, we replace the linked-list with a fetch and add (`FAA`) counter. Both counters are updated in isolation using a lock like explained above. The error distance is calculated from the difference between the two counter values.

Experiment results are then plotted using logarithmic scales, throughput (solid lines) and error distance (dotted lines) sharing the x-axis.

5.3 Monotonicity With High Degree of Relaxation

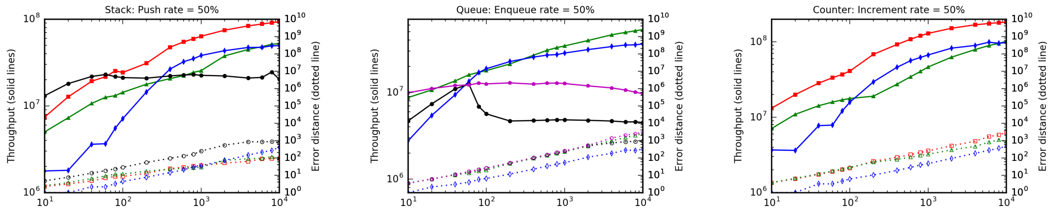
In order to evaluate monotonicity with increasing *relaxation* (k), we fix the number of threads to 16 as presented in Figure 2. This is to match the number of cores available on *Multi-S*.

First, we observe the difference between *WinCoupled* and *WinDecoupled* by comparing the *2Dc-Stack* and the *2Dd-Stack* respectively. *2Dc-Stack* consistently outperforms *2Dd-Stack* due to the reduced *Window* $shifting$ updates. With *2Dc-Stack*, a given thread can locally operate on the same *sub-stack* longer since operation counts cancel out each other, leaving the *sub-stack* in a valid state. This increases the probability of exploiting locality.

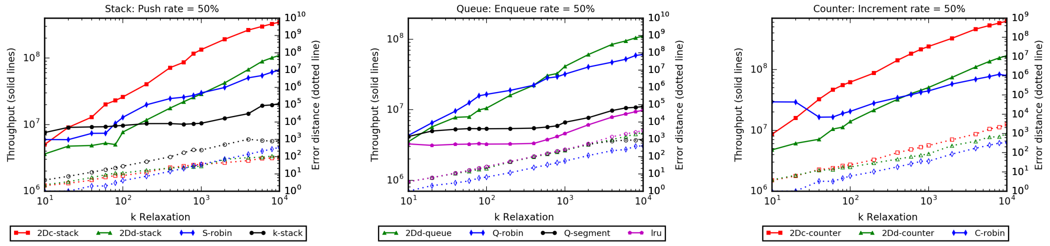
All algorithms increase their *width* as k increases to reduce contention and allow for increased disjoint *access*. However, for *k-Stack*, *Q-segment*, and *lru*, *hops* increase as *width* increases, this explains their observed low throughput. *S-robin*, *Q-robin* and *C-robin* are not affected by *hops*. However, for smaller k values, they suffer from high contention arising from contending threads retrying on the same *sub-structure* until they succeed. As contention vanishes with high k values, throughput gain saturates due to lack of locality.

$2D$ algorithms maintain throughput gain through limiting *width* to a size beneficial to reducing contention and switch to adjusting the *depth* to reduce *hops*. The *depth* parameter allows $2D$ algorithms to maintain throughput gain through exploiting locality while reducing latency. This is observed for both *Single-S* and *Multi-S* machines.

In terms of *accuracy*, we observe an almost linear decrease in *accuracy* as k increases for all algorithms.



(a) *Single-S*



(b) *Multi-S*

Figure 2 Throughput and observed *accuracy* as k bound relaxation increases, with 16 threads.

5.4 Scaling With Threads

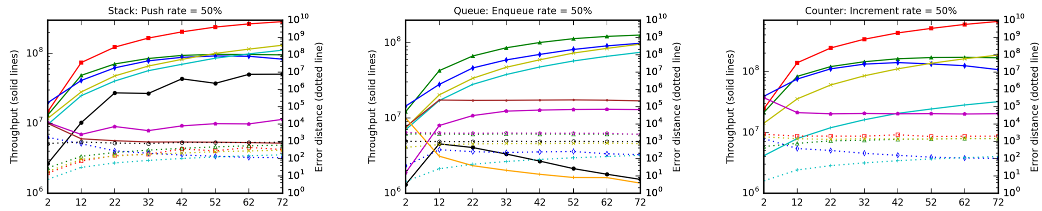
To evaluate scalability, we fix the *relaxation* bound to ($k = 10^4$) and vary the number of threads as shown in Figure 3. The reason for $k = 10^4$ is to reduce the effect of contention due to small *width* at lower k values. This helps us focus on scalability effects. *Random* and *Random-C2* algorithms' *width* is set to $3 \times$ *#threads*, as the optimal balance between throughput and *accuracy* since both of them do not provide *k relaxation* bounds as we mentioned before. *Random* and *Random-C2* *width* matches the 2*D* algorithms' *width* configuration, providing for a fair comparison.

For *Round-Robin* algorithms, the *width* is inversely proportional to #threads (see Theorem 6). As #threads increases, *width* reduces leading to increased contention. This explains the observed drop in throughput for a high number of threads, especially for the *S-robin* and the *C-robin* algorithms due to their *sub-structure* single *access*. The effect of lack of locality can be reduced by hardware pre-fetching, a feature available on both machines. This can also explain the *Round-Robin* better performance compared to the performance of the other algorithms that lack locality.

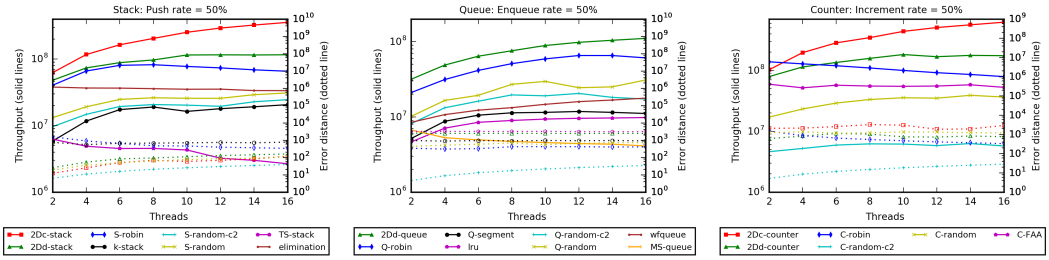
k-Stack and *Q-segment* maintain a constant segment size as the #threads increases. This increases the rate at which segments get filled up, leading to a high frequency of *hops* and segment maintenance cost. As observed, throughput gain saturates at high #threads leading to limited scalability.

The scalability of *lru* is limited by the global counter used to calculate the last recently used *sub-queue*. For every operation, the thread has to increment the global counter using a `FAA` instruction, turning the counter into a scalability bottleneck. This can be observed when *lru* performance is compared against that of a single `FAA` counter (*C-FAA*). *wfqueue* suffers from the same `FAA` counter sequential bottleneck.

Random and *Random-C2* algorithms are affected by the lack of locality, which is evident by the difference between *Single-S* and *Multi-S* results. We observe that the performance difference between *Random* and 2*D* algorithms increases on the *Multi-S* (Figure 3b) machine as compared to that on the *Single-S* (Figure 3b) machine. This demonstrates how much 2*D* algorithms gain from exploiting locality when executing on a *Multi-S* machine. Locality helps to avoid paying the high inter-socket communication cost [7, 16, 28].



(a) *Single-S*



(b) *Multi-S*

Figure 3 Throughput and observed *accuracy* as the number of threads increases, with $k = 10^4$.

TS-Stack's throughput is limited by the *Pop* search retries, searching for the newest item. Moreover, *Pop* operations might contend on the same newest items if there are not enough concurrent *Push* operations. Also, *Pop* lacks locality, which explains the drop in throughput on the *Multi-S* machine, due to the high inter-socket communication costs.

We observe an increase in the *accuracy* loss as the number of threads increase for 2*D* algorithms, *Random* and *Random-C2*. This is due to the increase in *width* as threads increase in number. *Round-Robin* algorithms show an increase in *accuracy* due to their decrease in *width*, whereas *lru*, *Q-segment* and *k-Stack* show little change in *accuracy* since the *width* and segment size does not change for the different number of threads respectively.

6 Conclusion

In this work, we have shown that semantics *relaxation* has the potential to monotonically trade relaxed semantics of concurrent data structures for achieving throughput performance within tight *relaxation* bounds. This has been achieved through an efficient two-dimensional framework that is simple and easy to implement for different data structures. We demonstrated that, by deriving two-dimensional lock-free designs for stacks, FIFO queues, dequeues and shared counters.

Our experimental results have shown that *relaxing* in one dimension, restricts the capability to control *relaxation* behaviour in-terms of throughput and *accuracy*. Compared to previous solutions, our framework can be used to extend existing data structures with minimal modifications while achieving better performance in terms of throughput and *accuracy*.

References

1 Yehuda Afek, Guy Korland, and Eitan Yanovsky. Quasi-linearizability: Relaxed consistency for improved concurrency. In *International Conference on Principles of Distributed Systems*, pages 395–410. Springer, 2010.

2 Dan Alistarh, Trevor Brown, Justin Kopinsky, Jerry Zheng Li, and Giorgi Nadiradze. Distributionally Linearizable Data Structures. *CoRR*, abs/1804.01018, 2018. `arXiv:1804.01018`.

3 Dan Alistarh, Justin Kopinsky, Jerry Li, and Giorgi Nadiradze. The Power of Choice in Priority Scheduling. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, PODC '17, pages 283–292, 2017.

4 Dan Alistarh, Justin Kopinsky, Jerry Li, and Nir Shavit. The SprayList: A scalable relaxed priority queue. *ACM SIGPLAN Notices*, 50(8):11–20, 2015.

5 Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov, Maged M. Michael, and Martin Vechev. Laws of Order: Expensive Synchronization in Concurrent Algorithms Cannot Be Eliminated. *SIGPLAN Not.*, 46(1):487–498, January 2011.

6 Gal Bar-Nissan, Danny Hendler, and Adi Suissa. A Dynamic Elimination-Combining Stack Algorithm. *CoRR*, abs/1106.6304, 2011. `arXiv:1106.6304`.

7 Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 33–48, 2013.

8 Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures. *SIGARCH Comput. Archit. News*, 43(1):631–644, March 2015.

9 Edsger W. Dijkstra. The Structure of "THE"-multiprogramming System. *Commun. ACM*, 11(5):341–346, May 1968.

10 EW Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.

11 Mike Dodds, Andreas Haas, and Christoph M. Kirsch. A Scalable, Correct Time-Stamped Stack. *SIGPLAN Not.*, 50(1):233–246, January 2015.

12 Panagiota Fatourou and Nikolaos D. Kallimanis. Revisiting the Combining Synchronization Technique. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 257–266, 2012.

13 Elad Gidron, Idit Keidar, Dmitri Perelman, and Yonathan Perez. SALSA: scalable and low synchronization NUMA-aware algorithm for producer-consumer pools. In *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*, pages 151–160. ACM, 2012.

14 Andreas Haas, Thomas A. Henzinger, Andreas Holzer, Christoph M. Kirsch, Michael Lippautz, Hannes Payer, Ali Sezgin, Ana Sokolova, and Helmut Veith. Local Linearizability for Concurrent Container-Type Data Structures. In *27th International Conference on Concurrency Theory, CONCUR 2016, August 23-26, 2016, Québec City, Canada*, pages 6:1–6:15, 2016. `doi:10.4230/LIPIcs.CONCUR.2016.6`.

15 Andreas Haas, Michael Lippautz, Thomas A. Henzinger, Hannes Payer, Ana Sokolova, Christoph M. Kirsch, and Ali Sezgin. Distributed Queues in Shared Memory: Multicore Performance and Scalability Through Quantitative Relaxation. In *Proceedings of the ACM International Conference on Computing Frontiers*, CF '13, pages 17:1–17:9, 2013.

16 Daniel Hackenberg, Daniel Molka, and Wolfgang E. Nagel. Comparing Cache Architectures and Coherency Protocols on x86-64 Multicore SMP Systems. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 413–422, 2009.

17 Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat Combining and the Synchronization-parallelism Tradeoff. In *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, pages 355–364, 2010.

18 Danny Hendler, Nir Shavit, and Lena Yerushalmi. A Scalable Lock-free Stack Algorithm. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '04, pages 206–215, 2004.

19 Thomas A. Henzinger, Christoph M. Kirsch, Hannes Payer, Ali Sezgin, and Ana Sokolova. Quantitative Relaxation of Concurrent Data Structures. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 317–328, 2013.

20 Maged M. Michael. CAS-Based Lock-Free Algorithm for Shared Deques. In *Euro-Par 2003. Parallel Processing, 9th International Euro-Par Conference, Klagenfurt, Austria, August 26-29, 2003. Proceedings*, pages 651–660, 2003. `doi:10.1007/978-3-540-45209-6_92`.

21 Maged M. Michael and Michael 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*, PODC '96, pages 267–275, 1996.

22 Michael Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, 2001.

23 Mark Moir, Daniel Nussbaum, Ori Shalev, and Nir Shavit. Using Elimination to Implement Scalable and Lock-free FIFO Queues. In *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '05, pages 253–262, 2005.

24 Hamza Rihani, Peter Sanders, and Roman Dementiev. Brief announcement: Multiqueues: Simple relaxed concurrent priority queues. In *Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures*, pages 80–82. ACM, 2015.

25 Adones Rukundo, Aras Atalar, and Philippas Tsigas. 2D-Stack: A scalable lock-free stack design that continuously relaxes semantics for better performance. Technical report, Chalmers University of Technology, 2018. URL: `https://research.chalmers.se/publication/506089/file/506089_Fulltext.pdf`.

26 Adones Rukundo, Aras Atalar, and Philippas Tsigas. Brief Announcement: 2D-Stack - A Scalable Lock-Free Stack Design that Continuously Relaxes Semantics for Better Performance. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing, PODC 2018, Egham, United Kingdom, July 23-27, 2018*, pages 407–409, 2018. URL: `https://dl.acm.org/citation.cfm?id=3212794`.

27 Adones Rukundo, Aras Atalar, and Philippas Tsigas. Monotonically relaxing concurrent data-structure semantics for performance: An efficient 2D design framework. *CoRR*, abs/1906.07105, 2019. `arXiv:1906.07105`.

28 Hermann Schweizer, Maciej Besta, and Torsten Hoefler. Evaluating the Cost of Atomic Operations on Modern Architectures. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT)*, PACT '15, pages 445–456, 2015.

29 Nir Shavit. Data Structures in the Multicore Age. *Commun. ACM*, 54(3):76–84, March 2011.

30 Nir Shavit and Gadi Taubenfeld. The Computability of Relaxed Data Structures: Queues and Stacks As Examples. *Distrib. Comput.*, 29(5):395–407, October 2016.

31 Nir Shavit and Dan Touitou. Elimination Trees and the Construction of Pools and Stacks: Preliminary Version. In *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '95, pages 54–63, 1995.

32 Nir Shavit and Asaph Zemach. Combining funnels: a dynamic approach to software combining. *Journal of Parallel and Distributed Computing*, 60(11):1355–1387, 2000.

33 Edward Talmage and Jennifer L. Welch. *Relaxed Data Types as Consistency Conditions*, pages 142–156. Springer International Publishing, Cham, 2017.

34 R.K. Treiber. *Systems Programming: Coping with Parallelism.* Research Report RJ. International Business Machines Incorporated, Thomas J. Watson Research Center, 1986.

35 Martin Wimmer, Jakob Gruber, Jesper Larsson Träff, and Philippas Tsigas. The Lock-free k-LSM Relaxed Priority Queue. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 277–278, 2015.

36 Chaoran Yang and John 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*, PPoPP '16, pages 16:1–16:13, 2016.