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


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”.

© Adones Rukundo, Aras Atalar, and Philippas Tsigas; licensed under Creative Commons License CC-BY 4.0.

© 2019, Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany
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](https://via.placeholder.com/150.png)

**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
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 2D 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 dequesues.

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.
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_{\text{max}}) and minimum (Win_{\text{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_{\text{max}}), empty, if all sub-structures have minimum operations (sub-count = Win_{\text{min}}). The Window is defined by two parameters; width and depth. width = \#sub-structures, and depth = Win_{\text{max}} − Win_{\text{min}}.

To validate a sub-structure, its sub-count is compared with Win_{\text{min}}, or Win_{\text{max}}, either sub-count ≥ Win_{\text{min}} or sub-count < Win_{\text{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_{\text{min}} and Win_{\text{max}}, the process we refer to as, Window shifting. A Window can shift up or down, and is controlled by shift_{\text{up}} or shift_{\text{down}} values respectively, where, 0 < shift_{\text{up}} ≤ depth. Win_{\text{min}} and Win_{\text{max}} can only be incremented or decremented by a given shift value. shift_{\text{up}} and shift_{\text{down}} can be configured differently to optimize for different workloads.

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

**WinCoupled:** couples both \text{Pat} and \text{Get} to share the same Window and sub-count for each sub-structure. A successful \text{Pat} increments whereas, a successful \text{Get} decrements the given sub-count. On a full Window, \text{Pat} increments Win_{\text{max}} shifting the Window up (shift_{\text{up}}), whereas, on an empty Window, \text{Get} decrements Win_{\text{max}}, shifting the Window down (shift_{\text{down}}). WinCoupled resembles elimination [18], only that here, we cancel out operation counts for matching \text{Pat} and \text{Get} on the same sub-structure within the same Window. Just like elimination reduces joint access updates, WinCoupled reduces Window shift updates.

**WinDecoupled:** decouples \text{Pat} and \text{Get} and assigns them independent windows. Also, an independent sub-count is maintained for \text{Pat} or \text{Get}, on each sub-structure. Unlike WinCoupled, both operations always increment their respective sub-count on a successful operation and Win_{\text{max}} on a full Window. This implies that both sub-count and Window counters are always increasing.

---

**Algorithm 1** Window Coupled (2Dc).

```
1 Struct Descriptor Des
2 *item; count; version;
3 sub-count;
4 max;
5 Struct Window Win
6 max;
7 version;
8 Function Window(Op, index, cont)
9 IndexSearch = Random = notempty=0; LWin = Win;
10 if cont==True then
11 index=RandomIndex(); cont=False;
12 end
13 while True do
14 if IndexSearch == width then
15 return {Des,index};
16 end
17 DES = Array[index]; > Read descriptor
18 if Op == get ∧ Des.count < Win.max then
19 return {Des,index};
20 else if Op == get ∧ Des.count > (Win.max - depth) then
21 return {Des,index};
22 else if LWin == Win then
23 return {Des,index};
24 end
25 LWin = Win; IndexSearch = 0;
26 end
27 end
28 Macro SHIFTWINDOW()
29 if Op == get ∧ notempty==0 then
30 return {Des,index};
31 end
32 if Win == Win then
33 NWin = Win; Win.max = LWin.max + ShiftUp;
34 else if Op == get ∧ Win.max > depth then
35 NWin = Win; Win.max = ShiftDown;
36 end
37 NWin.version = LWin.version + 1;
38 CAS(LWin, NWin);
39 index += 1;
40 if notempty=1 then
41 index=RandomIndex(); Random+=1;
42 else
43 index=RandomIndex(); Random+=1;
44 end
45 if LWin == Win; IndexSearch = 0;
46 for i = 0 -> width then
47 if Des.item!=NULL then
48 notempty=1;
49 end
50 if Random==False then
51 return {Des,index};
52 end
53 IndexSearch += 1;
54 end
55 end
```
Semantic Relaxation in 2D

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 = \text{Win}_{\text{max}} - \text{Win}_{\text{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 \text{Win}_{\text{max}} (line 18,20). If no valid sub-structure is found, \text{Win}_{\text{max}} is updated atomically, shifting the Window up or down (line 39). Put increments \text{Win}_{\text{max}} to shift the Window up (line 34), whereas, Get decrements \text{Win}_{\text{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.

```plaintext
1 Function Push(NewItem)
2 while True do
3   \{Des,Index\} = Window(push,index,cont);
4  NewItem.next = Des.item;
5   NDes.item = NewItem;
6   NDes.count = Des.count + 1;
7   NDes.version = Des.version;
8   if CAS(Array[Index],Des,NDes) then
9     return 1;
10   else
11     cont=True;
12   end
13 end
14 Function Pop()
15 while True do
16   \{Des,Index\} = Window(pop,index,cont);
17   if Des.item != NULL then
18     NDes.item = Des.item.next;
19     NDes.count = Des.count - 1;
20     NDes.version = Des.version;
21     if CAS(Array[Index],Des,NDes) then
22       return Des.item;
23     else
24       cont=True;
25     end
26   else
27     return Null;
28   end
29 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 ($op_{\text{cont}}$) and contention-free ($op_{\text{free}}$) operation costs, given by $op = op_{\text{cont}}/\text{width} + op_{\text{free}}$. 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 $1/op_{\text{free}}$. 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 \(\lim_{\text{width} \to \infty} \text{op}_{\text{cont}} \to 0\) 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 \(p = P(\text{Put})\) and \(E(\text{Extra}) = E(\text{hop}) + E(\text{shift}).\)

\[ \textbf{Theorem 1.} \text{ For a 2Dc-structure that is initialized with parameters depth, width, shift = depth and } p = 1/2, E(\text{Extra}) = O(\sqrt{\ln \text{width}/\text{depth}}). \]

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 2D 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 \(\text{CAS}\) only if there is another successful operation. For \(\text{WinCoupled}\), \(\text{Window shifting}\) is lock-free iff \(\text{shift} < \text{depth}\), whereas it is always lock-free for \(\text{WinDecoupled}\). A Window shift can only fail if there is another successful shift operation preceded by a successful \(\text{Put}\) or \(\text{Get}\), ensuring system progress. Thus, all our derived algorithms are lock-free. Our design framework can also be used for lock based data structures.

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

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

Relaxation can be applied method-wise and it is applied only to \text{Pop} operations, that is, a \text{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 \(\text{shift} = \text{shift}_{\text{up}} = \text{shift}_{\text{down}}\). A Window \(i\) \((W_i)\) has an upper bound \(W_i^{\text{max}}\) and a lower bound \(W_i^{\text{min}}\), where \(W_i^{\text{max}} = i \times \text{shift}\) and \(W_i^{\text{min}} = i \times \text{shift} - \text{depth}\), respectively. For simplicity, let \(\text{Global}\) represent the current global upper bound \((\text{Win}_{\text{max}})\). A Window is active iff \(W_i^{\text{max}} = \text{Global}\). The number of items of the sub-stack \(j\) is denoted by \(N_j\), \(1 \leq j \leq \text{width}\). 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 \text{CAS} instruction.

\[ \textbf{Lemma 3.} \text{ Given that } \text{Global} = \text{shift} \times i, \text{ it is impossible to observe a state(S) such that } N_j > W_i^{\text{max}} \text{ or } N_j < W_i^{\text{min}}. \]

\[ \textbf{Proof.} \text{ We show that this is impossible by considering the interleaving of operations. Without loss of generality, assume thread } i (P_i) \text{ has set } \text{Global} = \text{shift} \times i \text{ at time } t_i. \text{ To do this, } P_i \text{ should have observed either } \text{Global} = \text{shift} \times (i - 1) \text{ and then } N_j = W_i^{\text{max}} \text{ or } \text{Global} = \text{shift} \times (i + 1) \text{ and then } N_j = W_i^{\text{min}}. \text{ Let this observation of Global happen at time } t_i. \text{ Consider the last successful push operation at sub-stack } j \text{ before the state } S \text{ is observed for the first time (we do not consider Pop operations as they can only decrease } N_j \text{ to a value} \]
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 \( \text{Global} > W_{i+1}^{max} \). Let \( j \) be selected at time \( t_0 \). And the linearization of the operation happens at \( t_0' > t_0 \).

- If \( t_0' < t_1 \), the concerned state \( (S) \) can not be observed since \( \text{Global} \) cannot be changed (to \( \text{shift} \times i \)) after \( N_j > W_{i+1}^{max} \) is observed.
- Else if \( t_1 < t_0 \), the concerned state \( (S) \) cannot be observed since the push operation cannot proceed after observing \( \text{Global} \) 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 \( \text{Global} \) 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 \( \text{Global} \) has been modified, since it was read by \( P_1 \), thus updating \( \text{Global} \) would fail, at least based on the version number.

\[ \textbf{Lemma 4.} \text{ At all times, there exist an } i \text{ such that } \forall j, 1 \leq j \leq \text{width}: W_i^{min} \leq N_j \leq W_i^{max} \]

**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 \( (y \text{ and } z) \) at some point in time such that \( \exists i, N_y < W_i^{min} \) and \( N_z > W_i^{max} \). Consider the last \( \text{Push} \) at sub-stack \( z \) and last \( \text{Pop} \) 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 \( N_y \geq W_i^{max} \) and \( \text{Global} > W_i^{max} \), let sub-stack \( z \) be selected at \( t_0 \). And, the linearization of the \( \text{Push} \) operation occurs at \( t_0' > t_0 \). Similarly, for \( P_1 \) Pop operation, let sub-stack \( y \) be selected at \( t_1 \), \( P_1 \) should have observed \( \text{Global} \leq W_i^{min} \). And, let the \( \text{Pop} \) operation linearize at time \( t_1 > t_1 \). Now, we consider the possible interleavings.

- If \( t_0' < t_1 \) (or the symmetric \( t_1 < t_0 \) 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 \( \text{Global} \leq W_i^{min} \).
- 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 \( \text{Global} \) 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.

\[ \textbf{Theorem 5.} \text{ 2Dc-Stack is linearizable with respect to } k\text{-out-of-order stack semantics, where } k = (2\text{shift} + \text{depth})(\text{width} - 1) \]

**Proof.** Consider the \( \text{Push} \) \( (t_e^{push}) \) and \( \text{Pop} \) \( (t_e^{pop}) \) 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: \( \forall i: N_j \geq W_{i}^{min} - \text{shift} \). Similarly, we can deduce that at time \( t_e^{pop} \), the following holds: \( \forall i: N_j \leq W_{i}^{max} + \text{shift} \). 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 \( W_{i}^{max} + \text{shift} - (W_{i}^{min} - \text{shift}) = \text{depth} + 2\text{shift} \). We know that this number is zero for sub-stack \( j \) (the sub-stack that \( e \) is inserted) and we have \( \text{width} - 1 \) other sub-stacks. So, there can be at most \( 2\text{shift} + \text{depth})(\text{width} - 1) \) 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 
\[ k = (2 \times \text{#threads} - 1)(\text{#sub-stacks} - 1). \]

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 pinned 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 × #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 bru, 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.
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 \(3 \times \#\text{threads}\), 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 2D 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 2D algorithms increases on the Multi-S (Figure 3b) machine as compared to that on the Single-S (Figure 3b) machine. This demonstrates how much 2D 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].
Throughput and observed accuracy as the number of threads increases, with $k = 10^4$.

*Figure 3*

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 $2D$ 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, deques 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


31:14  Semantic Relaxation in 2D


