Technical Report no. 2018:06

2D-Stack:
A scalable lock-free stack design that continuously relaxes semantics for better performance

Adones Rukundo, Aras Atalar, Philippas Tsigas

Email: {adones, aaras, tsigas}@chalmers.se

CHALMERS | GÖTEBORG UNIVERSITY

Distributed Computing and Systems
Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden
Göteborg, 2018

.

Technical Report in Computer Science and Engineering at
Chalmers University of Technology and Göteborg University

Technical Report no. 2018:06
ISSN: 1652-926X

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden

Göteborg, Sweden, 2018

Abstract

In this report, we propose an efficient lock-free concurrent stack design with tunable and tenable relaxed semantics to allow for better performance. The design is materialized by a shared memory distributed stack design, that allow for a continuous monotonic trade of weaker semantics for better throughput performance. Concurrent stacks have an inherent scalability bottleneck due to their single access point for both push and pop operations.

Elimination and semantics relaxation have been proposed in the literature to address this problem. Semantic relaxation has the potential and flexibility to reach monotonically very high throughput. Previous solutions could not fully leverage this potential. We propose a new two-dimensional design that can achieve this by exploiting disjoint access parallelism in one dimension and locality in the other. This is achieved through distributing the stack in form of sub-stacks that are accessed independently in parallel. Load balancing is used to keep a balanced number of operations on individual sub-stacks.

We also provide tight relaxation bounds for the behaviour of our algorithm. We compare experimentally to previous work, with respect to throughput and relaxed behaviour observed, on different relaxation and concurrency settings. The results show that our algorithm significantly outperform all other algorithms in terms of performance, while maintaining better quality in contrast to other designs with relaxed semantics.

Keywords— Stack, Lock-free, Relaxed Semantics, Concurrency, Data Structures, Weak Consistence, Distributed Algorithms.

1 Introduction

Stacks entered the computer science literature in 1946, when Alan M. Turing used the terms "bury" and "unbury" as a means of calling and returning from subroutines. Stacks are linear data structures or more abstractly sequential collections of ordered items with two principle operations: Push, which adds an item and pop, which removes an item. The *Push* and *Pop* operations occur at a single point of the structure, referred to as the top or head of the stack. Stack operations follow the Last In First Out (*LIFO*) semantics. Concurrent stacks, like any other concurrent data structure, require synchronization to guarantee the behavior that is legal with respect to their exact sequential specifications. However, synchronization may incur high performance overhead with increase in the number of *threads*. In the case of a stack, the overhead can be attributed to the joint access point at the stack top leading to contention and a scalability bottleneck. Synchronization is vital to achieving correctness and cannot be eliminated [5], whereas this is true, synchronization and scalability conflict in form of contention. To reduce contention and improve on scalability, synchronization points need to be distributed by creating disjoint access points. Disjoint access techniques for stacks like; elimination [1, 12, 17], combining [18] and dynamic elimination-combining [7] have been proposed in the literature. Such techniques allow *threads* to complete their operations without necessarily accessing the top of the stack. These techniques however depend on the existence of other concurrent operations being present, introducing a waiting time between dependant operations. For example *Push* waiting for *Pop* elimination and *Push* waiting for *Push* combining.

To further improve performance scalability of concurrent data structures, recent research has focused on expanding the set of legal behaviours, including; weakening consistency and semantic relaxation for providing trade-offs between scalability and linearizability guarantees. Computability of relaxed data-structures [16] together with their relaxed semantics definitions including; *k-Out-of-Order*, k-Lateness and k-Stuttering have been proposed in the literature as interesting relaxation models to consider [13, 20].

Distributing parts and hence access of the data-structure [11, 15, 21], has come out as a frequent technique used to implement relaxation. A given data-structure is split into multiple sub-structures with independent access points to improve on disjoint access parallelism. Operations are distributed over the sub-structures using different scheduling techniques; *thread* binding [21], random access [15], load-balancing [11], round robin and a combination of others. Various relaxed data-structures have been proposed in the literature, most use one dimension relaxation exploiting disjoint access parallelism or locality. Disjoint access is achieved through creating extra access points, whereas locality is achieved by controlling the number of *threads* that share a given access point.

In this report, we aim to leverage the semantics relaxation through exploiting disjoint access parallelism and locality. Locality can be obtained by letting a *thread* work on the same access point for some time in isolation. Previous works have used *thread* to sub-structure bidding to exploit locality. However, to maintain LIFO accuracy bounds and differ from pool semantics [19, 10, 1], a mechanism that synchronizes the *thread* local works or limits the amount of the local work must be introduced. This might turn out to be the performance bottleneck as one increases the degree of concurrency [10]. We introduce an algorithm (*2D-stack*) that enables disjoint access parallelism and exploits the locality within strict deterministic accuracy bounds in an efficient way, avoiding expensive work sharing mechanisms. This would not only increase the performance for a given configuration but also give one the capability to monotonically trade the accuracy for better performance. This is achieved through a distributed stack, composed of multiple *sub-stacks*. Each *sub-stack* independently accessed through a pointer pointing to the topmost item of the given *sub-stack*. We also implement three other distributed stack designs based on known basic scheduling techniques, including *Random Single Choice* (*Random*), *Random Choice of Two* (*Random-c2*), *Round-Robin* (*k-robin*). This is to help us have a detailed comparison of stack semantic relaxation, since such designs have not been proposed with relaxed semantics before. Our *2D-stack* design exhibits a two dimension scheduling technique (load balancing), exploiting disjoint access parallelism in one dimension and locality in the other. *2D-stack* significantly outperforms previous stack implementations including the extra

three implemented stack designs as observed in the experimental evaluation Section 7.

The report is structured as follows. In Section 2 we discuss literature related to this work. We present the implementations in Section 3 and prove correctness in Section 4. A step complexity analysis is discussed in Section 5. We select optimization parameters in Section 6 and use them for experimental evaluation presented in Section 7. We then conclude in Section 8

2 Related Work

Concurrent stacks are inherently sequential due to their single point access bottleneck. In the quest to improve performance scalability, disjoint access strategies have been proposed for designing concurrent stacks including; elimination trees [1, 17], combining funnels [18] and elimination back-off [7, 12]. Elimination back-off implements a collision array in which pop operations try to collide and cancel with concurrent push operations to reduce joint access on top of the stack. Such operation pairs create disjoint collisions that are executed in parallel with operations accessing the main stack implementation. As an extended back-off strategy, it reduces joint access to the main stack by canceling out paired operations and completing their execution on the collision array. However, the performance benefits flatten out fast as the number of *threads* increases to a certain threshold. Elimination back-off mostly benefits symmetric workloads in which the numbers of push and pop operations are roughly equal, its performance deteriorates when workloads are asymmetric.

Recently, semantic relaxation has been proposed for data-structures that provide trade-offs between scalability and linearizability guarantees. Relaxation techniques introduce an acceptable error within the legal strict semantics of a given data-structure, i.e. the pop operation of a relaxed stack is allowed to return any of the k topmost items of the stack. To quantify this error, relaxed semantic definitions including; *k-Out-of-Order*, k-Stuttering and k-Lateness have been introduced [13, 20]. Based on these definitions, new designs have been proposed for some fundamental data structures to introduce relaxation. A *k-Out-of-Order* stack has been proposed in [13, 2], referred to as *Segmented* (*k-segment*) henceforth. It is composed of a linked list of memory segments whose size is defined by k number of indexes. The stack items can only be accessed through the topmost segment, where an operation pushes or pops an item from any k indexes. A $Push$ operation adds a new segment if top segment is full whereas a Pop removes a segment if it is empty and not the last segment. A $Push$ operation tries to push an item onto an empty slot in the top segment, adding a new segment if the segment is full. A Pop operation tries to remove an item from the top segment, removing segment if empty and is not the only segment on the stack. Operations perform a linear search for an empty ($Push$) or filled (Pop) index, starting with randomly index in the topmost segment. Relaxation is controlled through the *width* dimension with the segment size increasing with increase in of k. Increasing k improves on disjoint access and reduces contention up-to a given thresh hold. This is because as k increases to infinity, at some point the gains from the reduced contention diminishes whereas the cost for traversing memory in search of an index increases. This is coupled with accuracy loss proportional to increase in k. Also for small k numbers, there is high cost of operation synchronization when trying to remove or add a segment. These performance characteristics of the proposed design, limit scaling and the performance gains of the algorithm with increase in relaxation.

Other relaxed data-structures proposed include, priority queues [4, 15, 21] and shared memory distributed FIFO queues [11].

3 Relaxed Stack Design Description

In this section, we describe our *2D-Stack* two dimensional design plus other three related distributed stack designs that we implement for the sake of a detailed study. We follow the same *sub-stack* design with different *thread* scheduling techniques. An array of pointers (*stack-array*) is used to access individual *sub-stacks*. Each *sub-stack* is independently accessed through its topmost item

pointer at a given *stack-array* index. Operations on each *sub-stack* follow the lock-free Treiber stack design [8].

The three extra scheduling techniques lead to different stack algorithms that reveal different performance and *accuracy* characteristics. *Random* and *Random-c2*, are simple randomized stack algorithms, they use the *width* dimension to distribute operations by randomly selecting a given *sub-stack* to operate on. *Random-c2* increases *accuracy* by employing the random choice of two technique. Our third algorithm *k-robin* uses the *width* dimension to distribute operations following a strict round robin *sub-stack* selection.

3.1 *2D-Stack*

2D-Stack algorithm uses three parameters to tune its performance: *width*, *depth* and $shift$ which form a count based bidirectional operational region (*window*) in which an operation can occur. Number of *sub-stacks* is defined by the *width* whereas *depth* defines the maximum number of items acceptable for a single *sub-stack* per *window*. We implement a global counter (*Global*) that limits the *depth*, by defining the maximum and minimum number of items per *sub-stack* for a given active *window*. The *window* and *Global* give us the liberty to optimize for both *accuracy* and throughput within tightly defined *accuracy* bounds. A given *thread* randomly selects a *sub-stack* to operate on, tries to keep on the same *sub-stack* for as long there is no contention and without violating the *Global* restrictions. If contention is detected on a given *sub-stack*, the *thread* randomly selects another *sub-stack* to reduce on contention. This allows *threads* to optimistically exploit locality without *thread* to *sub-stack* bidding.

The algorithm operations are depicted in Algorithm 1. *2D-stack* is a shared memory distributed stack, composed of multiple lock-free *sub-stacks*. An individual *sub-stack* is implemented using a linked list whose operations follow the Treiber stack design [8]. Each *sub-stack* has a unique *descriptor* (line 1 to 4) that keeps track of the *sub-stack* information including; pointer to the topmost item and item-counter. A *descriptor* has a dedicated memory location accessed through an array (*stack-array*). Using a CAE[1] instruction we can update the descriptor contents in one atomic step to maintain correctness (line 15 and 27 for $Push$ and Pop respectively).

To perform an operation, a *thread* searches for a *sub-stack* based on the *Global* ($GetIndex$). A *thread* selects a *sub-stack*, then, compares the *sub-stack* item-count with the *Global* (line 61 or 65). The *thread* can then proceed on the selected *sub-stack* only if the comparison evaluates to true (line 46 or 48). Otherwise the *thread* has to search for another *sub-stack*. For each operation, the *thread* starts from the previously known *sub-stack* on which it succeeded (line 44). First the *thread* tries a given number of random hops (line 50), then switches to round robin until a valid *sub-stack* is found, or the *thread* updates the *Global*, after failing on all *sub-stacks* (line 64 or 68).

The *Global* is updated in relation to *depth*. If the *thread* detects contention on a *sub-stack*, a random hop to another *sub-stack* is performed (line 18 or 30). This is to reduce possible contention on consecutive *sub-stacks* that might arise from round robin hops. It also introduces our concept of optimistic locality. A *thread* can operate on given *sub-stack* for as long no contention is detected. A CAE fail signals the presence of another *thread* operating on the same *sub-stack*. To avoid further contention, the *thread* that has failed leaves the successful *thread* to take over the *sub-stack*.

During the search, the *thread* validates each *sub-stack* item-count against the *Global*. The item-count must be less than *Global* for $Push$ or greater than the difference between *Global* and *depth* for Pop (line 45 or 47). If the item-count is zero, then the *sub-stack* is empty. If no valid *sub-stack* is found, the *Global* is updated atomically (line 64 or 68). $Push$ adds whereas Pop subtracts a value ($shift$) (line 62 or 66), $shift$ must be less than or equal to *depth*. Then the search is restarted with a fresh search count. If a valid *sub-stack* is found, the *thread* tries to operate on it, on success the *sub-stack descriptor* is updated (line 15 or 27) otherwise another *sub-stack* is searched for, starting from a random index (line 18 or 30).

[1] Compare and Exchange (CAE) atomically compares 16 bytes of memory content and exchanges it with new content on success.

A successful *Push* increments whereas a *Push* decrements the item-counter by one. 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 *sub-stack* or NULL for empty. An empty *sub-stack* is represented by a NULL item pointer within the *descriptor*. As an optimization strategy, the *thread* keeps track of the *Global* for every hop during the search process, restarting for every *Global* change detected. Keeping track of the *Global* prevents the *thread* from doing useless search with stale *Global* information. Consider a *Push thread* that reads *Global* just before it is incremented by a preceding *Push thread*. The succeeding *thread* will have to check all *sub-stacks* searching for non full *sub-stack* before proceeding to access the updated *Global*. Note that, keeping track of the *Global*, has no added cost apart from the constant one cache miss while accessing an updated *Global*.

Algorithm 1: *2D-Stack*

```
1  Struct Descriptor
2      *item;
3      count; unsigned long
4      version; unsigned long
5  Struct Global
6      count; unsigned long
7      version; unsigned long
8  Function Push(NewItem)
9      while true do
10         {Des,Index} = GetIndex(push,Index);
11         NewItem.next = Des.item;
12         NDes.item = NewItem;
13         NDes.count = Des.count + 1;
14         NDes.version = Des.version + 1;
15         if CAE(Array[Index],Des,NDes) then
16             return 1;
17         else
18             Index=RandomIndex();
19         end
20     end
21 Function Pop()
22     while true do
23         {Des,Index}=GetIndex(pop,Index);
24         if Des.item != NULL then
25             NDes.item = Des.item.next;
26             NDes.count = Des.count - 1;
27             if CAE(Array[Index],Des,NDes) then
28                 return Des.item;
29             else
30                 Index=RandomIndex();
31             end
32         else
33             return Null;
34         end
35     end
36                 ▷ Des stands for descriptor;
37                   ▷ Glo stands for Global;
38 Function GetIndex(Op,Index)
39     IndexSearch = 0; PGlo = Glo; Random = 0;
40     while true do
41         if Index ≥ ArraySize then
42             Index = 0;
43         end
44         Des = Array[Index]        ▷ Read descriptor;
45         if Op == push ∧ Des.count < Glo.count then
46             return {Des,Index};
47         else if Op == pop ∧ (Des.count ≥ (Glo.count
             - depth) ∨ Des.count == 0) then
48             return {Des,Index};
49         else if PGlo == Glo then
50             if Random<2 then
51                 Index=RandomIndex();
52                 Random+=1;
53             else
54                 Index += 1;
55             end
56         else
57             PGlo = Glo; IndexSearch = 0;
58         end
59         if IndexSearch == ArraySize then
60             IndexSearch = 0;
61             if Op == push ∧ PGlo == Glo then
62                 NGlo.count = PGlo.count + ShiftUp;
63                 NGlo.version = PGlo.version + 1;
64                 CAE(Glo,PGlo,NGlo);
65             else if Op == pop (∧ PGlo == Glo ∧
                 (Glo.count - depth) > 0) then
66                 NGlo.count = PGlo.count - ShiftDown;
67                 NGlo.version = PGlo.version + 1;
68                 CAE(Glo,PGlo,NGlo);
69             end
70             PGlo = Glo;
71         else if Random ≥ 2 then
72             IndexSearch += 1;
73         end
74     end
```

3.2 Other Shared Memory Distributed Stack Designs

In this section, we present other distributed stack designs based on known basic scheduling techniques. They are briefly described to give the reader an implementation overview for a better performance comparison with the *2D-stack*.

3.2.1 *Round-Robin*

This algorithm uses two parameters to tune its performance: number of threads and number of *sub-stacks*. Unlike *Random* and *Random-c2* algorithms, *k-robin* provides a deterministic *accuracy* bound, linearizable with respect to *k-Out-of-Order* stack semantics. The algorithm distributes operation following a strict round robin fashion without skipping a *sub-stack*. Each thread has two local independent counters, a *Pop* operation counter and a *Push* operation counter. A *thread* tries to operate on a *sub-stack* indicated by a given operation counter, if successful, it increases the respective counter for the next operation. Otherwise it keeps trying on the same *sub-stack* until it succeeds.

3.2.2 *Random Single Choice*

This is the most basic algorithm designed on top of the *sub-stack* design. It takes a single parameter: number of *sub-stacks* (*width*). For both *Pop* and *Push* operations, a *thread* selects a *sub-stack* uniformly at random to perform its operation. Once the *sub-stack* is selected, the respective operation follows the Treiber stack design.

3.2.3 *Random Choice of Two*

In the bid to improve on *accuracy*, *Random Single Choice* is extended to *Random*. *Random-c2* design is based on the principle of power of random two choices[14], also similar to MultiQueues [6] and Power of choice of two [3]. Like in *Random* the number of *sub-stacks* remains as the only parameter to select for tuning. The algorithm is depicted in Algorithm 2. Each pushed element is tagged with a time-stamp generated using a globally consistent clock (line 5). Time-stamps provide for a logical global ordering of elements. A *Pop* operation randomly selects two *sub-stacks* and proceeds to pop from the *sub-stack* whose element has the highest time-stamp (line 25). A *Push* operation, randomly selects a *sub-stack* and proceeds to push the element onto it (line 4).

Algorithm 2: *Random Choice of Two*

```
1  Function PushItem (*NewItem)
2      *Item;
3      while true do
4          index = RandomIndex(); Item = StackArray[index].item;
5          NewItem→next = Item; NewItem→tag = Timestamp;
6          if CAS(StackArray[index].item, Item, NewItem) then
7              return 1;
8          end
9      end
10 Function PopItem ()
11     *Item; *NewItem; tag1 = 0; tag2 = 0;
12     while true do
13         index1 = RandomIndex();
14         while StackArraySize>1 do
15             index2 = RandomIndex();
16             if index1 != index2 then
17                 tag1 = Item1→tag; tag2 = Item2→tag;
18                 if tag1 > tag2 then
19                     index = index1; break ;
20                 else
21                     index = index2; break ;
22                 end
23             end
24         end
25         Item = StackArray[index].item;
26         if Item != NULL then
27             NewItem = Item→next;
28             if CAS(StackArray[index].item, Item, NewItem) then
29                 return Item;
30             end
31         else
32             return 1;
33         end
34     end
```

4 Correctness

In this section, we prove the correctness of our algorithms. We examine and prove linearizability and lock-freedom for both *k-robin* and *2D-stack*. We do not consider *Random* and *Random-c2* in this section, since the k would be unbounded (proportional to the maximum number of items in the stack) for these two algorithms.

To begin with, we introduce the linearization points of *Push* and *Pop* operations for both stack designs; the linearization points are the same points (same program lines) that the original Treiber's Stack implementation had as lineraiztion points. For *2D-stack*, *Pop* linearizes either by returning *NULL* (at line 33) or with a successful CAE at line 27. *Push* linearizes with a successful CAE at line 15.For *k-robin*, *Pop* linearizes either by returning *NULL* or with a successful CAS that pops an item by modifying the top pointer of a *sub-stack*. *Push* linearizes with a successful CAS that modifies the top pointer of a *sub-stack*.

We prove the linearizability of *2D-stack* and *k-robin* with respect to the sequential semantics of *k-Out-of-Order* stack [13], which provides a relaxed version of the LIFO semantics. Relaxation can be applied method-wise and it is applied only to *Pop* operations in *k-Out-of-Order* stack, i.e. a *Pop* pops one of the topmost k items. *Push* operations add the item to the top of the stack.

4.1 *2D-Stack*

Firstly, we require some notation. *window* defines the active region in which the operations are allowed to proceed (line 45 and 47 for *Push* and *Pop* respectively). The *window* is shifted by the parameter $shift$, $1 \leq shift < depth$. A *window* i (W_i^{up}) has an upper bound (W_i^{up}) and a lower bound (W_i^{down}), that are defined by $W_i^{up} = depth + (i \times shift)$ and $W_i^{down} = i \times shift$, respectively. And, a *window* is active iff $W_i^{up} = Global$. The *width* parameter describes the number of *sub-stacks*. 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 CAE.

Lemma 1 *Given that* $Global = depth + shift \times i$*, it is impossible to observe a state(S) such that* $N_j > W_{i+1}^{up}$ *(or* $N_j < W_{i-1}^{down}$*), where* $1 \leq j \leq width$*.*

Proof: Recall that $Global = W_i^{up}$ defines the active window where the operations are allowed to start. Though, they might linearize while the active window is set to an adjacent window ($Global = W_{i-1}$ or $Global = W_{i+1}$). We can not observe such a state in the initialization, therefore there should exist a point in time that this state (S) is observed for the first time, with $Global = depth + shift \times i$ and $N_j > W_{i+1}^{up}$ (or $N_j < W_{i-1}^{down}$, but we do not consider this symmetric case in the proof since it can be covered with the same arguments as $N_j > W_{i+1}^{up}$). Now, we show that this is impossible by considering the interleaving of operations.

Without loss of generality, assume *thread* 1 (T_1) has set $Global = depth + shift \times i$ with the CAE (at line 64 or 68) at time t_{l1}. To do this, T_1 should have observed either $Global = depth + shift \times (i-1)$ and then $N_j = W_{i-1}^{up}$ or $Global = depth + shift \times (i+1)$ and then $N_j = W_{i+1}^{down}$. Let this observation of *Global* (at line 39) 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}^{up}, this case will be covered by the first item below). Assume thread 0 (T_0) sets N_j to $N_j > W_{i+1}^{up}$ in this push operation. In this operation, T_0 should observe $N_j \geq W_{i+1}^{up}$ (at line 44) and $Global > W_{i+1}^{up}$ (at line 45). Let line 45 is executed (atomic read) at time t_0. And the linearization of the operation happens at $t_{l0} > t_0$ with CAE (at line 15).

- If $t_{l0} < t_1$, the concerned state(S) can not be observed since, *Global* can not be changed (to $depth + shift \times i$) after $N_j > W_{i+1}^{up}$ is observed.
- Else if $t_{l1} < t_0$, the concerned state(S) can not be observed since, the push operation can not proceed after observing *Global* (at line 45) with such N_j.

- Else if $t_1 > t_0$, then T_0 can not succeed in the CAE (at line 15) because this implies that N_j has been modified (the difference between the value of *Global* that is observed by T_0 and then by T_1 implies this) since T_0 have read the descriptor (at line 44), at least the version numbers would have changed since then, thus leading to a failed CAE (at line 15).
- Else if $t_1 < t_0$, then this implies *Global* has been modified since it had read by T_1 (at line 39), thus CAE (at line 64 or 68) would fail, at least based on the version number.

Hence, the lemma. ■

Lemma 2 *At all times, there exist an i such that $\forall j, 1 \leq j \leq width$: $W_i^{down} \leq N_j \leq W_{i+1}^{up}$.*

Proof: Informally, the lemma states that the size of the *sub-stacks* spans to at most two consecutive windows. Assume that the statement is not true, then there should exist a pair of *sub-stacks* (let k and j) at some point in time such that $\exists i, N_j > W_{i+1}^{up}$ and $N_k < W_i^{down}$. One can not observe such N_j and N_k at the initialization. Then, there should exist a time t that this is observed for the first time. Consider the last push operation at *sub-stack* j and last pop operation at *sub-stack* k that linearize before or at the time t.

Assume thread 0 (T_0) sets N_j and thread 1 (T_1) sets N_k. To do this, T_0 should observe $N_j \geq W_{i+1}^{up}$ (at line 44) and $Global > W_{i+1}^{up}$ (at line 45), let line 45 (atomic read of *Global*) is executed at t_0. And, the linearization of the $Push$ operation occurs at $t_{l0} > t_0$ with the CAE (at line 15). Similarly, for the *Pop* operation of T_1, let line 47 is executed at t_1 and the observed value should be $Global \leq W_i^{down}$. And, let the *Pop* operation linearize at time $t_{l1} > t_1$ with the CAE (at line 27). Now, we consider the possible interleavings.

- If $t_{l0} < t_1$ (or the symmetric $t_{l1} < t_0$ for which we do not repeat the arguments), then for T_1 to proceed and pop an item from *sub-stack* k, it is required that $Global \leq W_i^{down}$ (at line 47). Based on Lemma 1, this is impossible when $N_j > W_i^{up}$.
- Else if $t_1 > t_0$, then T_0 can not succeed in the CAE (at line 15) because this implies that N_j has modified (the difference between the value of *Global* that is observed by T_0 and then by T_1 implies this) since T_0 have read the descriptor (at line 44). At least, the version number would have changed since then, thus leading to a failed CAE (at line 15).
- Else if $t_0 > t_1$, the argument above holds for T_1 too, so T_1 should fail at the CAE (at line 27).

Such N_j and N_k pair can not co-exist at any time, hence, the lemma. ■

Theorem 3 *2D-stack algorithm is linearizable with respect to k-Out-of-Order stack semantics, where $k = (2shift + depth)(width - 1)$.*

Proof: Consider the linearization points of the $Push$ and the Pop operations that insert and remove the item e into and from a *sub-stack* (let *sub-stack* j). Let t_e^{push} and t_e^{pop} denote these points respectively, $t_e^{pop} > t_e^{push}$. 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 N_j become x with the linearization of the push operation that inserts item e. In other words, item e is the xth item from the bottom of the *sub-stack*. Consider a *window* i such that: $W_i^{down} \leq x \leq W_i^{up}$.

Lemma 2 states that the sizes of the *sub-stacks* should reside in a bounded region. Relying on Lemma 2, we can deduce that at time t_e^{push}, the following holds: $\forall i : N_i \geq W_i^{down} - shift$. Similarly, we can deduce that at time t_e^{pop}, the following holds: $\forall i : N_i \leq W_i^{up} + shift$. Therefore, the maximum number of items, that are pushed to *sub-stack* i ($i \neq j$) after t_e^{push} and are not popped before t_e^{pop} is at most $W_i^{up} + shift - (W_i^{down} - shift) = depth + 2shift$. 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 $(depth + 2shift)(width - 1)$ items that are pushed after t_e^{push} and are not popped before t_e^{pop}. Hence, the theorem. ■

4.2 *Round-Robin*

Theorem 4 *k-robin is linearizable with respect to k-Out-of-Order stack semantics, where* $k = (2P-1)(width-1)$*. P stands for the total number of concurrent threads and width denotes the number of sub-stacks.*

Proof: Consider the linearization points of $Push$ and Pop operations that respectively insert and remove the item e into and from a *sub-stack* (let *sub-stack* 0). Let t_e^{push} and t_e^{pop} denote these points, respectively. 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. We denote the number items that are pushed to (popped from) *sub-stack* i by thread j in the time interval $[t_e^{push}, t_e^{pop}]$, with $push_i^j$ (pop_i^j).

Observe that each thread applies its operations in round robin fashion without skipping any index. If the previous successful Pop had occurred at *sub-stack* i, the next Pop occurs at *sub-stack* $i+1(mod\ width)$. The same applies for the push operations.

Without loss of generality, assume that thread 0 has inserted item e to *sub-stack* 0. This implies that $\forall i, width-1 \geq i > 0,\ push_0^0 \geq push_i^0$. Now, take another thread j, we have $\forall i : width-1 \geq i > 0, push_0^j \geq push_i^j - 1$. Informally, another thread can increase the number of items on any other *sub-stack* by at most one more compared to the number number of items that pushes on *sub-stack* 0.

For the pop operations, we have the same relation for all threads: $\forall i, width \geq i > 0,\ pop_0^j \geq pop_i^j + 1$. Informally, a thread can pop at most 1 item less from any other *sub-stack* compared to the number that it pops from *sub-stack* 0. As the interval $[t_e^{push}, t_e^{pop}]$ starts with the push and ends with the pop of item e at *sub-stack* 0, we have $\sum_{j=0}^{T-1} push_0^j = \sum_{j=0}^{T-1} pop_0^j = Y$.

Summing over all threads and *sub-stacks* other than *sub-stack* 0, we get at most $(Y+T-1)(width-1)$ $Push$ operations in the interval $[t_e^{push}, t_e^{pop}]$. Summing over all threads and *sub-stacks* other than *sub-stack* 0, weget at least $(Y-T)(width-1)$ Pop operations. Which leads to the theorem: $k \leq ((Y+T-1)-(Y-T))(width-1) = (2T-1)(width-1)$

4.3 Lock-freedom

All algorithm designs presented in this study are lock-free. This follows from the properties of the lock-free Treiber Stack design except for *2D-Stack*. An operation can fail on CAE only if there is another successful operation ensuring the system progress. For the *2D-Stack*, one should additionally consider if there is a possibility of live-lock due to the update of *Global* that determines the active *window*. The *Global* can be updated repeatedly back and forth if two opposite operations follow each other on an empty or full active *window*. For example, a *Pop* operation might read an empty *window* and update *Global* leading to a full *window*, but before it performs its operation, a subsequent $Push$ reads the full *window* and updates *Global* leading to an empty *window*. This process can continue forever leading to a system live lock. This can however be avoided by setting the $shift$ parameter to less than *depth*. With this setting, a *Global* update can never lead to a full or empty *window* unless if the stack is empty. Therefore, a thread would eventually proceed and reach to the CAE at the end of a $Push$ or a Pop that can only fail due to another concurrent successful operation.

5 Complexity Analysis

In this section, we analyze the *2D-stack* expected step complexity of a sequential process where a single *thread* applies the sequence of operations. The type of an operation in the sequence is determined with an independent coin toss with a fixed probability, where p denotes the probability of a $Push$ operation. With distributed access points, it is possible to make multiple hops on different access points before finding an appropriate point to complete a given operation.

Global regulates the size of the *sub-stacks*. Recall that the number of *sub-stacks* is denoted by *width* and the size of *sub-stack* i by N_i. *Push* and *Pop* operations are allowed to occur at *sub-stack* i, if $N_i \in [Global - depth, Global - 1]$ and $N_i \in [Global - depth + 1, Global]$, respectively. This basically means that, at any time, the size of a *sub-stack* can only variate in the vicinity of *Global*, more precisely: $\forall i, (Global - depth) \leq N_i \leq Global$. To recall, this interval is valid for the sequential process. We refer to this interval as the active region.

We introduce random variables $N_i^{active} = N_i - (Global - depth)$, $N_i^{active} \in [0, depth]$ that provides the number of items in the active region of the *sub-stack* i and the random variable $N^{active} = \sum_{i=1}^{width} N_i^{active}$ provides the total number of items in the *window*.

As mentioned before the *2D-stack* tries to exploit locality, thus, a *thread* starts an operation with a query on the *sub-stack* where the last successful operation occurred. This means that the *thread* hops iff $N_i^{active} = 0$ or $N_i^{active} = depth$ respectively for a *Pop* or a *Push* operation. Therefore, the number of *sub-stacks*, whose active regions are full, is given by $\lfloor (N^{active}/depth) \rfloor$ at a given time, because the *thread* does not leave a *sub-stack* until its active region gets either full or empty. If the *thread* hops a *sub-stack*, then a new *sub-stack* is selected uniformly at random from the remaining set of *sub-stacks*. If none of the *sub-stacks* fulfills the condition (which implies that $N^{active} = 0$ at a *Pop* or $N^{active} = depth \times width$ at a *Push*), then the window shifts based on a given $shift$ parameter. (*i.e.* for a *Push* operation $Global = Global + shift^{up}$ and for a *Pop* operation $Global = Global - shift_{down}$, where $1 \leq shift_{down} \leq depth$). One can observe that the value of N^{active} before an operation defines the expected number of hops and the slide of the *window*.

To compute the expected step complexity of an operation that occurs at a random time, we model the random variation process around the *Global* with a Markov chain, where the sequence of *Push* and *Pop* operations lead to the state transitions. As a remark, we consider the performance of the *sub-stacks* mostly when they are non-empty, since *Pop* ($NULL$) and *Push* would have no hops in this case. The Markov chain is strongly related to N^{active}. It is composed of $K + 1$ states $\mathcal{S}_0, \mathcal{S}_1, \ldots, \mathcal{S}_K$, where $K = depth \times width$. For all $i \in [\![0, K]\!]$, the operation is in state $\mathcal{S}_i$ iff $N^{active} = i$. For all $(i, j) \in [\![0, K+1]\!]^2$, $\mathbb{P}(\mathcal{S}_i \to \mathcal{S}_j)$ denotes the state transition probability, that is given by the following function, where p denotes the probability of a *Push*:

$$\begin{cases} \mathbb{P}(\mathcal{S}_i \to \mathcal{S}_{i+1}) = p, & \text{if } 0 < i < K \\ \mathbb{P}(\mathcal{S}_i \to \mathcal{S}_{i-1}) = 1 - p, & \text{if } 0 < i < K \\ \mathbb{P}\left(\mathcal{S}_i \to \mathcal{S}_{K-(shift \times width - 1)}\right) = p, & \text{if } i = K \\ \mathbb{P}\left(\mathcal{S}_i \to \mathcal{S}_{(shift \times width - 1)}\right) = 1 - p, & \text{if } i = 0 \\ \mathbb{P}(\mathcal{S}_i \to \mathcal{S}_j) = 0, & \text{otherwise} \end{cases}$$

The stationary distribution (denoted by the vector $\pi = (\pi_i)_{i \in [\![0,K]\!]}$) exists for the Markov chain (let $\mathbb{P}(\mathcal{S}_i \to \mathcal{S}_i) = \epsilon$, $i \in \{depth, K - depth\}$, for some *Pop* returning $NULL$), since the chain is aperiodic and irreducible. The left eigenvector of the transition matrix with eigenvalue 1 provides the unique stationary distribution.

Lemma 5 *For the Markov chain that is initialized with $p = 1/2$ and $shift$, where $l = shift \times width - 1$, the stationary distribution is given by the vector $\pi^l = (\pi_0^l \pi_1^l .. \pi_K^l)$, assuming $K - l >= l$ (for $l > K - l$, one can obtain the vector from the symmetry $\pi^l = \pi^{K-l}$): (i) $\pi_i^l = \frac{i+1}{(l+1)(K+1-l)}$, if $i < l$; (ii) $\pi_i^l = \frac{l+1}{(l+1)(K+1-l)}$, if $l \leq i \leq K - l$; (iii) $\pi_i^l = \frac{K-i+1}{(l+1)(K+1-l)}$, if $i > K - l$.*

Proof: We have stated that the stationary distribution exist since the chain is aperiodic and irreducible for all p and $shift$. Let $(M_{i,j})_{(i,j) \in [\![0,K]\!]^2}$ denotes the transition matrix for $p = 1/2$ and $shift$. The stationary distribution vector π^l fulfills, $\pi^l M = \pi^l$, that provides the following system of linear equations: (i) $2\pi_0^l = \pi_1^l$; (ii) $2\pi_K^l = \pi_{(K-1)}^l$; (iii) $2\pi_i^l = \pi_{i-1}^l + \pi_{i+1}^l$; (iv) $2\pi_l^l = \pi_{i-1}^l + \pi_{i+1}^l + \pi_1^l$; (v) $2\pi_{K-l}^l = \pi_{i-1}^l + \pi_{i+1}^l + \pi_K^l$.

In case, $l = K - l$, then (iv) and (v) are replaced with $2\pi_{(l=K-l)}^l = \pi_{i-1}^l + \pi_{i+1}^l + \pi_1^l + \pi_K^l$.

Based on a symmetry argument, one can observe that, for all l, $\pi_i^l = \pi_{K-i}^l$ the system can be solved in linear time ($O(K)$) by assigning any positive (for irreducible chain $\pi_i^l > 0$) value to π_0^l. The stationary distribution is unique thus for any π_0^l, π^l spans the solution space. We know that $\sum_{i=0}^{K} \pi_i^l = 1$, starting from $\pi_0^l = 1$, we obtain and normalize each item by the sum. ■

An operation starts with the search of an available *sub-stack*. This search contains at least a single query at the *sub-stack* where the last success occurred, therefore we define the rest of the hops as the extra hops (denoted by *Hop*, and they can be at most *width*). In addition, the operation might include the slide of the window, as an extra step, denoted by *Glo*. We denote the number of extra steps with $Extra = Hop + Glo$. With the linearity of expectation, we obtain $\mathbb{E}(Extra) = \mathbb{E}(Hop) + \mathbb{E}(Glo)$. Relying on the law of total expectation, we obtain: (i) $\mathbb{E}(Hop) = \sum_{i=0}^{K} \sum_{op \in \{pop,push\}} \mathbb{E}(Hop|\mathcal{S}_i, op)\,\mathbb{P}(\mathcal{S}_i, op)$; (ii) $\mathbb{E}(Glo) = \sum_{i=0}^{K} \sum_{op \in \{pop,push\}} \mathbb{E}(Glo|\mathcal{S}_i, op)\,\mathbb{P}(\mathcal{S}_i, op)$; where $\mathbb{P}(\mathcal{S}_i, op)$ denotes the probability of an operation to occur in state $\mathcal{S}_i$. We analyze the algorithm for the setting where $shift = depth$ and $p = 1/2$. We do this because the bound, that we manage to find in this case, is tighter, and gives a better idea of the influence of the parameters to the expected performance. For this case the stationary distribution is given by Lemma 5.

Theorem 6 *For a 2D-Stack that is initialized with parameters depth, width, $shift = depth$ and $p = 1/2$, $\mathbb{E}(Extra) = O(\frac{\ln width}{depth})$.*

Proof: Firstly, we consider the expected number of extra steps for a *Push* operation. Given that there are N^{active} items, a *Push* attempt would generate an extra step if it attempts to push to a *sub-stack* that has $N_i^{active} = depth$ items. Recall that the *thread* sticks to a *sub-stack* until it is not possible to conduct an operation on it. This implies that the extra steps can be taken only in the states $\mathcal{S}_i$ such that $i(mod\ depth) = 0$, because the *thread* does not leave a *sub-stack* before $N_i^{active} = 0$ or $N_i^{active} = depth$. In addition, a *Push* (*Pop*) can only experience an extra step if the previous operation was also a *Push* (*Pop*).

Given that we are in $\mathcal{S}_i$ such that $i(mod\ depth) = 0$, then the first requirement is to have a *Push* as the previous operation. If this is true, then the *Push* operation hops to another *sub-stack*, which is selected from the remaining set of *sub-stacks* uniformly at random. At this point, there are $f = \frac{i}{depth} - 1$ full *sub-stacks* in the remaining set of *sub-stacks*. If a full *sub-stack* is selected from this set, this leads to another hop and again a *sub-stack* is selected uniformly at random from the remaining set of *sub-stacks*.

Consider a full *sub-stack* (one of the f), this *sub-stack* would be hopped if it is queried before quering the *sub-stacks* that are empty. There are $width - f - 1$ empty *sub-stacks*, thus a hop in this *sub-stack* would occur with probability $1/(width - f)$. There are f such *sub-stacks*. With the linearity of expectation, the expected number of hops is given by: $f/(width - f) + 1 = width/(width - f)$. Which leads to $\mathbb{E}(Hop|\mathcal{S}_i, Push) = p \times width/(width - f)$ if $i(mod\ depth) = 0$ or $\mathbb{E}(Hop|\mathcal{S}_i, Push) = 0$ otherwise.

From Lemma 5, $\pi_i < 2/(K+1)$ we obtain:

$$\mathbb{E}(Hop|Push) = \sum_{i=0}^{K} \pi_i \mathbb{E}(Hop|\mathcal{S}_i, Push) < \Big(\sum_{f=0}^{width-1} \frac{width}{width - f}\Big)\frac{2p}{K+1}$$
$$< (\ln(width - 1) + \gamma)\frac{width}{K+1} < (\ln width + \gamma)\frac{1}{depth}$$

The bounds for $\mathbb{E}(Hop|Push)$ would also hold for $\mathbb{E}(Hop|Pop)$. Given that there are $K - i$ (system is in state $\mathcal{S}_i$) empty *sub-stacks* then there are $e = \lfloor \frac{K-i}{depth} \rfloor - 1$ *sub-stacks* whose window regions are empty, minus the *sub-stack* that the *thread* last succeeded on. Using the same arguments that are illustrated above (replace f with e and p=1-p), we obtain the same bound.

Window only shifts at $\mathcal{S}_K$ if a *Push* operation happens and at $\mathcal{S}_0$ if a *Pop* operation happens. Hence: $\mathbb{E}(Glo) < \frac{2}{K+1}p + \frac{2}{K+1}(1-p)$. Finally, using $\mathbb{E}(Extra) = \mathbb{E}(Hop) + \mathbb{E}(Glo)$ we obtain the theorem. ■

6 Parameter Selection

One of the goals of the *2D-stack* is to be tunable in order to regulate the trade-off between accuracy and performance. In this section, we investigate the impact of the *2D-stack* parameters to accuracy and performance, to come up with the optimal parameter settings.

Empirically, we observed that the contention is inversely proportional to *width* see Figure 1. As a simple model, we split the latency of an operation into the contention and contention-free operation costs, denoted by $op = op_{cont}/width + op_{free}$. Based on this rough estimation, 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 after some point one can not really gain throughput by increasing the *width* although it keeps losing in terms of the accuracy. The situation is a bit more complex because of the extra steps that the algorithm might take as *width* increases (See Theorem 6). In this case, we update the latency of an operation with an additional factor of $(O(\frac{\ln width}{depth}))$. Meaning that, after some point, the gains from the contention factor ($\lim_{width\to\infty} op_{cont} \to 0$) might be surpassed by the extra steps and one would observe a decrease in throughput with the increase of *width*. This is counter-intuitive in terms of the trade-off between the accuracy and performance. To keep the trade-off alive, we turn our attention to the *depth* parameter (*depth* relaxation). This parameter can be used to exploit data locality, which might have a very significant impact on the throughput, especially in a *NUMA* setting. Operations done by the same *thread* in isolation at the same *sub-stack* can yield very high throughput.

In Figure 1, the red curve (L1) depicts the case where we only use the *width* relaxation for all K ($depth = 1$). The other curves diverge from that one at some point where we start *depth* relaxation. With respect to the performance, we see that it is reasonable to apply *width* relaxation in the beginning until $width = 4P$ (P stands for the number of threads), where we obtain enough disjoint access parallelism. After this parameter saturates, one can continue to relax in the *depth* dimension to increase the performance via better locality and fewer extra steps. In terms of accuracy, we observe that the expected *error-distance* increases almost linearly with the *width* parameter, whereas it increases almost logarithmically with the increase of the *depth* parameter (4P for $k > 200$). One can almost recklessly relax on the *depth* dimension to exploit the locality.

Now, we target to minimize the window maintenance cost. In a sequential process, this cost is not significant. But, in a concurrent execution, this could be very costly as each update impose a cost to all *threads*. For the small values of the relaxation, the window gets updated more frequently. For this we optimize the process by tuning the $shift$ parameter. Without having a general solution, we illustrate the optimal value of the $shift$ parameter where $p = 1/2$ (p denotes the probability of a $Push$), for the sequential process that we consider. We will show that $shift = depth/2$ (assuming *depth* is even) is optimal for $\mathbb{E}(Glo)$. Intuitively, these two metrics (Hop and Glo) seems to be correlated because the window shifts only after the maximum number of hops. And, it is more probable to observe a window $shift$ in an interval with operations that complete after many extra steps. We believe, the minimization of $\mathbb{E}(Glo)$ would also reduce $\mathbb{E}(Hop)$, for all values of p.

Lemma 7 *For the Markov chain that is initialized with $p = 1/2$ and $shift$, where $l = shift \times width$ the stationary distribution is given by the vector $\pi^l = (\pi_0^l \pi_1^l .. \pi_K^l)$: (i) $\pi_i^l = \frac{i+1}{(l+1)(K+1-l)}$, if $i < l$; (ii) $\pi_i^l = \frac{l+1}{(l+1)(K+1-l)}$, if $l \leq i \leq K-l$; (iii) $\pi_i^l = \frac{K-i+1}{(l+1)(K+1-l)}$, if $i > K-l$.*

Proof: In Section 5, we have stated that the stationary distribution exist since the chain is aperiodic and irreducible for all p and $shift$. Let $(M_{i,j})_{(i,j)\in[\![0,K]\!]^2}$ denotes the transition matrix for $p = 1/2$ and $shift = depth/2$, that is given in Section 5. The stationary distribution vector π^l fulfills, $\pi^l M = \pi^l$, that provides the following system of linear equations: (i) $2\pi_0^l = \pi_1^l$; (ii) $2\pi_K^l = \pi_{(K-1)}^l$; (iii) $2\pi_i^l = \pi_{i-1}^l + \pi_{i+1}^l$; (iv) $2\pi_l^l = \pi_{i-1}^l + \pi_{i+1}^l + \pi_1^l$; (v) $2\pi_{K-l}^l = \pi_{i-1}^l + \pi_{i+1}^l + \pi_K^l$. In case, $l = K-l$, then (iv) and (v) are replaced with $2\pi_{(l=K-l)}^l = \pi_{i-1}^l + \pi_{i+1}^l + \pi_1^l + \pi_K^l$.

Based on a symmetry argument, one can observe that, for all l, $\pi_i^l = \pi_{K-i}^l$ the system can be solved in linear time ($O(K)$) by assigning any positive (for irreducible chain $\pi_i^l > 0$) value to π_0^l.

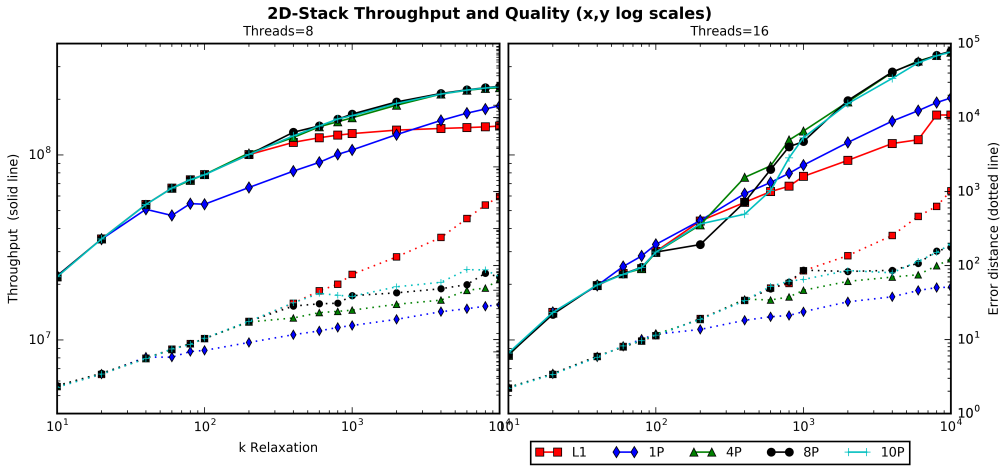

Figure 1: 2D-stack throughput and observed quality for different *depth* and *width* configurations

The stationary distribution is unique thus for any π_0^l, π^l spans the solution space. We know that $\sum_{i=0}^{K} \pi_i^l = 1$, starting from $\pi_0^l = 1$, we obtain and normalize each item by the sum.

Theorem 8 *Given* $p = 1/2$, $shift = depth/2$ *minimizes* $\mathbb{E}(Glo)$, *where* $K = width \times depth$ *and* $l = width \times shift$.

Proof: The global barrier is updated either with the transition $\mathcal{S}_0 \rightarrow \mathcal{S}_{shift \times width}$ generated by a *Pop* operation or $\mathcal{S}_K \rightarrow \mathcal{S}_{(K-shift \times width)}$ generated by a *Push* operation. Therefore, our objective is to minimize: $\mathbb{E}(Glo) = p\pi_K^l + (1-p)\pi_0^l$. With $p = 1/2$ and the symmetry of the Markov chain states $\pi_i^l = \pi_{K-i}^l$, the objective reduces to minimizing π_0^l. Based on Lemma 7, we have that $\pi_0^l = \frac{1}{(l+1)(K+1-l)} = \frac{1}{-l^2+Kl+K+1}$. The quadratic function $f(l) = -l^2+Kl+K+1$ has its maximum value at $l = K/2$ since $f'(l) = K - 2l = 0$. Having $l = K/2 = shift \times width$, we obtain the optimal value: $shift = K/(2width) = depth/2$.

7 Experimental Evaluation

We evaluate the performance of our implementations together with other existing stack designs including; the *k-segment* relaxed stack [13], *Elimination-Stack* (*elimination*) [12] and *Treiber-Stack* (*treiber*) [8]. All experiments run on an Intel Xeon CPU E5-2687W v2 machine with two 8-core 3.40GHz Intel Xeon processors (16 cores, 2 threads per core). We pin one thread per core, filling one socket at a time up-to 16 threads before we switch to hyper-threading. Two NUMA settings are tested; intra-socket (1 to 8 threads) and inter-socket (9 to 16 threads). Threads select operations uniformly at random (*i.e.* with probability 1/2) from *Pop* and *Push* operations. Memory is managed using SSMEM [9]. To simulate high contention, we put no computational load between operations. For each experiment, the stack is initialized with 2^{15} items, run for five seconds obtaining an average of five repeats. Throughput is measured in terms of operations per second, whereas *accuracy* is measured in terms of error distance from the LIFO semantics.

To measure the *accuracy*, we adopt and modify a similar method in [4, 15]. A sequential linked list is run alongside the stack, for each *Push* or *Pop* a simultaneous insert or delete is performed on the list respectively. Items on the stack are duplicated on the list and can be identified by their unique labels. Insert operations happen at the head of the list similar to the push whereas the delete operation searches for the given item deletes it and returns its distance from the head (error distance). We then calculate the expected error distance for a given experiment run for 5 seconds with 5 repeats. Scalability is tested on relaxation, by changing (weakening continuously)

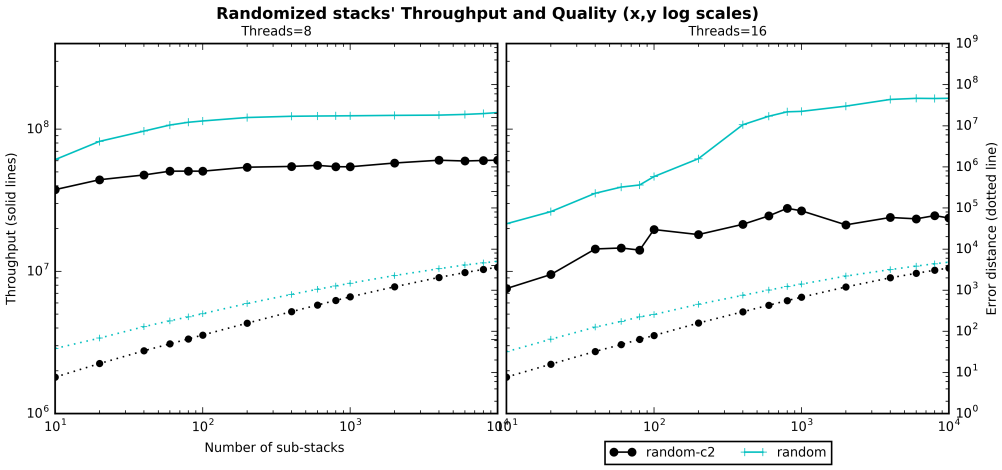

Figure 2: Randomized algorithms' throughput and observed quality with increasing *sub-stacks*

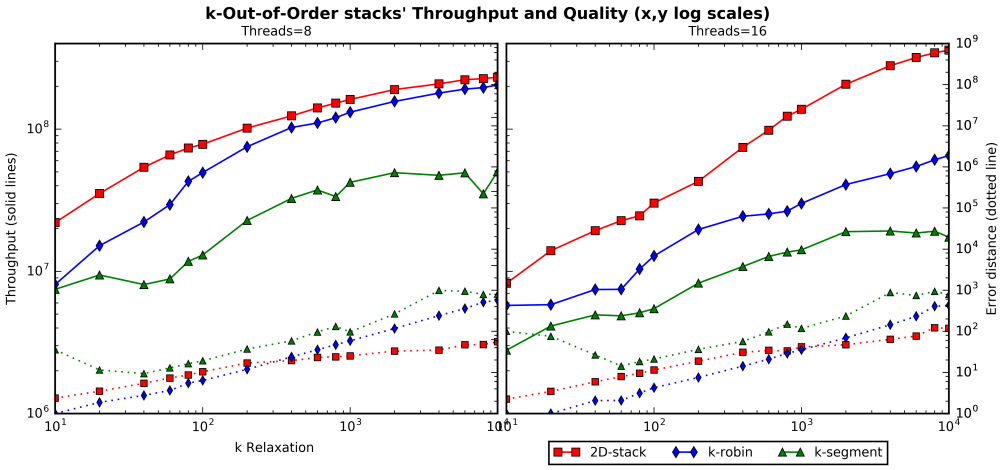

Figure 3: Throughput and observed quality as the k bound for relaxation increases.

relaxation and on concurrency, by changing (increasing) the number of *threads* for different NUMA settings. Experiment results are then plotted using logarithmic scales, throughput (solid lines) and error distance (dotted lines) sharing the x-axis.

Based on the analysis presented in Section 6, that was also confronted by our experimental observations, we select $4P$ (P stands for number of threads and $width = 4P$) as the transition point, from *width* to *depth* relaxation and $shift = depth/2$ as the optimal performance configuration for *2D-stack*. In Figure 3, we evaluate the performance of all algorithms, that are linearizable with respect to *k-Out-of-Order* stack (*k-robin*,*2D-stack* and *k-segment*), at the different relaxation levels. Randomized algorithms (*Random* and *Random-c2*) do not exhibit *k-Out-of-Order* bounds, for that, they are evaluated with respect to number of *sub-stacks* in Figure 2. We observe that *2D-stack* consistently outperforms the other algorithms followed by *k-robin* for both settings where the number of threads changes from 8 to 16 (NUMA setting). Under low degree of relaxation, *2D-stack* avoids contention by hopping to another *sub-stack* on a failed *CAE*. This highly improves performance compared to *k-robin* that keeps retrying on the same *sub-stack*. As the relaxation increases, *2D-stack* combines contention avoidance with locality exploitation, a parameter exclusive to the *2D-stack* design as explained in Section 6. While for the other algorithms the *accuracy* reduces almost linearly with the increase in relaxation, *2D-stack* maintains good *accuracy* with $width > 4P$ ($k > 200$ for $P = 8$ and $k > 600$ for $P = 16$). At this point, the algorithm switches from *width* to *depth* by

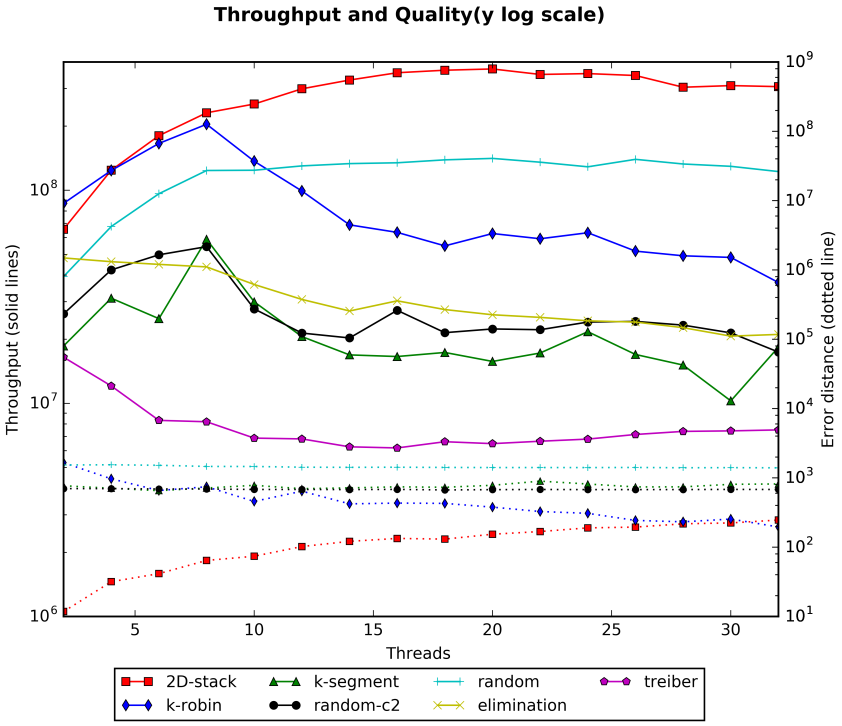

Figure 4: Throughput as concurrency (number of threads) increases.

increasing the depth. This change has a smaller negative impact on the *accuracy*, compared to the other algorithms. *2D-stack* continuously trades off *accuracy* for throughput by switching between relaxation dimensions for different relaxation levels. *k-segment* is mostly affected by the high cost of maintaining segments coupled with increased number of hops as relaxation increases.

We now configure the algorithms to obtain the maximum throughput performance for both intra and inter-socket settings, Figure 4. Based on the results that we observed and discussed before (Figure 2), *width* for *Random* and *Random-c2* translates to 10^3 *sub-stacks*. For *k-robin*, *2D-stack* and *k-segment* this translates to a $k = 10^4$, Figure 3. Two "non-relaxed" algorithms *elimination* and *treiber* are also included in the experiment to compare the power of relaxation to improve performance compared to other strict semantics efficiency improvement techniques. We generally observe that, *2D-stack* is able to maintain the increase in throughput also while increasing the number of threads, even for the NUMA settings. Under inter-socket setting, *Random* maintains almost a constant throughput as we increase concurrency with no throughput increase whereas the throughput performance for other algorithms drop. As the number of threads increase, *Random*, *Random-c2* and *k-segment* maintain almost constant *accuracy* due to the fixed number of *sub-stacks*. *k-robin* and *2D-stack* vary the number of *sub-stacks* as the number of *threads* change. *k-robin* reduces number of *sub-stacks* with the increase in number of *threads* to keep the *k*-bound, this improves *accuracy* but hurts throughput due to the increased contention. As observed, *2D-stack* maintains high throughput also when the number of threads increases for different NUMA settings. Overall, *2D-stack* shows a full control to leverage the semantics relaxation to reach very high throughput in a continuous way. A property that is missing in other solutions.

8 Conclusion

The aim of this work is to design an efficient lock-free stack algorithmic that can continuously relax semantics to improve throughput through exploiting disjoint access parallelism and locality.

We have achieved this through our two dimension relaxation stack design that exploits disjoint access parallelism in one dimension and locality in the other. The *2D-stack*, uses an efficient widow based synchronization technique, that manages to keep the relaxation bound without receding the significant performance achieved through locality. *2D-stack* significantly outperformed all the other stack implementations due to its capability to switch relaxation dimensions leading to a monotonic trade of accuracy for better performance. In addition to *2D-stack*, we have implemented and tested a set of other possible relaxed stack designs. Together with step complexity analysis, we also provided tight accuracy bounds for two algorithms presented in this report including *2D-stack*.

References

[1] Yehuda Afek, Guy Korland, Maria Natanzon, and Nir Shavit. Scalable producer-consumer pools based on elimination-diffraction trees. *Euro-Par 2010-Parallel Processing*, pages 151–162, 2010.

[2] 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.

[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, New York, NY, USA, 2017. ACM.

[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] Yossi Azar, Andrei Z. Broder, Anna R. Karlin, and Eli Upfal. Balanced allocations. *SIAM J. Comput.*, 29(1):180–200, September 1999.

[7] Gal Bar-Nissan, Danny Hendler, and Adi Suissa. A dynamic elimination-combining stack algorithm. *CoRR*, abs/1106.6304, 2011.

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

[9] 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.

[10] 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.

[11] 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, New York, NY, USA, 2013. ACM.

[12] Danny Hendler, Nir Shavit, and Lena Yerushalmi. A scalable lock-free stack algorithm. *Journal of Parallel and Distributed Computing*, 70(1):1–12, 2010.

[13] Thomas A Henzinger, Christoph M Kirsch, Hannes Payer, Ali Sezgin, and Ana Sokolova. Quantitative relaxation of concurrent data structures. In *ACM SIGPLAN Notices*, volume 48, pages 317–328. ACM, 2013.

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

[15] 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.

[16] Nir Shavit and Gadi Taubenfeld. The computability of relaxed data structures: Queues and stacks as examples. *Distrib. Comput.*, 29(5):395–407, October 2016.

[17] 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*, pages 54–63. ACM, 1995.

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

[19] Håkan Sundell, Anders Gidenstam, Marina Papatriantafilou, and Philippas Tsigas. A lock-free algorithm for concurrent bags. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 335–344, New York, NY, USA, 2011. ACM.

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

[21] Martin Wimmer, Jakob Gruber, Jesper Larsson Träff, and Philippas Tsigas. The lock-free k-lsm relaxed priority queue. In *ACM SIGPLAN Notices*, volume 50, pages 277–278. ACM, 2015.