MASTER'S THESIS 2022

Highly Scalable Queues and Stacks with Elastic Relaxation

Kåre von Geijer

ISSN 1650-2884
LU-CS-EX: 2022-36

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

Highly Scalable Queues and Stacks with Elastic Relaxation

(Elastically extending a semantically relaxed lock-free framework)

Kåre von Geijer
Kare.kvg@gmail.com

June 22, 2022

Master's thesis work carried out at Chalmers University of Technology.

Supervisors: Philippas Tsigas, Philippas.Tsigas@chalmers.se
Jonas Skeppstedt, Jonas.Skeppstedt@cs.lth.se

Examiner: Flavius Gruian, flavius.gruian@cs.lth.se

Abstract

Traditional concurrent data structures like queues or stacks have inherent bottlenecks due to all operations having to access them through the same points, leading to high contention. By relaxing the semantics of a data structure, we don't have to force each operation to take effect in the same order as they were invoked. This can often lead to reduced contention and increased throughput, at the cost of accuracy.

This thesis builds on an earlier paper which introduced a lock-free framework for such semantically relaxed data structures where the relaxation could be decoupled in two orthogonal dimensions. Our contribution is to be able to change these two relaxation measures during run time for their queue and stack. The two data structures use different ideas but build on creating auxiliary nodes to encode relaxation changes.

By analyzing the algorithms this thesis proves their correctness and bound their relaxation errors. In the empirical evaluations, the elastic stack had almost no throughput penalty and the elastic queue was only a few per cent slower when compared to the original.

Keywords: Concurrent Data Structures, Lock Free, Semantic Relaxation, Elasticity

Acknowledgements

I would like to thank my supervisor Philippas Tsigas for taking me on even though I am not a student of Chalmers. Although his time was limited he allocated time for weekly meetings and introduced me to many interesting ideas and people. Furthermore, I would like to thank my other supervisor Jonas Skeppstedt who not only supported me through this project but was also the one who introduced me to algorithms and multicore programming.

Additionally, I want to thank the other members of the DCS group at Chalmers who have been very welcoming during my visits. A special thanks go out to Dimitrios Palyvos-Giannas, Christos Profentzas and Georgia Tsaloli for the many fun discussions during lunch.

Finally, I want to thank Melinda Rydberg for her support through the ups and downs as well as the many draining train rides, and my parents for letting me use their home when visiting Chalmers and Gothenburg.

Contents

Chapter 1
Introduction

Basic data structures such as queues and stacks are used in a plethora of different applications where things need to be collected and ordered. Their ideas are often simple and easy to implement in sequential programs. But as the amount of multi-threaded applications continues to increase, so does the demand for efficient concurrent data structures. Data structures like the Michael Scott queue[8] or the Treiber stack[12] are great in that they are concurrent and lock-free. But they are inherently limited due to their sequential specifications. Their low number of access points quickly becomes a bottleneck when the load and number of threads increase[2].

A potential solution to this is called *Semantic relaxation*[6] and it relaxes the sequential specification of the data structure to allow different out of order behaviours. The relaxation we will look at is *k-out-of-order*[10] which for a stack allows each pop to return any element among the top $k + 1$ elements, instead of always the topmost one. There has been quite a lot of work in this area recently and several good data structures have emerged[6, 10, 9, 13, 4, 5]. A common tactic is to increase the number of access points by having several normal data structures in parallel which for example can be seen in the MultiQueue[13] and the 2D framework[10]. A problem with them, and most others[6, 4, 5] are that the amount of relaxation has to be specified up front at compile time. This limits the applications as they cannot change behaviour dynamically over time[9].

This thesis presents a solution to that problem for the queue and coupled stack from the 2D framework by extending them to be able to vary relaxation during run time. We call this extension elastic since it is still gives strong relaxation guarantees while also making the data structures more flexible. This extension is done for the 2D queue and the 2Dc stack from the 2D framework[10], and we use a different new method for each of them. We think these methods are applicable to similar data structures and should at the least be usable for the remaining 2D stack and deque. This extension to the 2D framework makes it an excellent choice when you want a highly efficient and scalable relaxed data structure, with a bound on the actual relaxation, which you can have running for a long time and adapt its behaviour depending on workload.

The research question of the thesis is *How can the list-based data structures from the 2D framework[10] be extended, so that the allowed relaxation can be changed during run time?* and to limit the scope we will only investigate the queue and stack. To be able to measure how good such an extension is we use the following points, which all should be fulfilled for an ideal extension.

- The extension should be lock-free.
- The extension should guarantee some notion of correctness and have a bound for worst-case relaxation, like the original data structures[10].
- The extension should have similar throughput to the original data structures when the relaxation is kept constant.
- A change in relaxation should take effect quickly.

The thesis starts with the background needed to understand the work in chapter 2 and goes over some related work in chapter 3. Chapter 4 describes our new ideas and designs and in chapter 5 we prove their correctness and bound their relaxation. Finally, we evaluate our implementation in chapter 6 and end with some discussion and conclusions in chapter 7.

Chapter 2
Background

This section will describe the necessary material to understand the ideas in this thesis. It starts with the fundamentals of concurrent programming, and unless otherwise specified, the source we use for all of these fundamentals is the extensive book *Writing Efficient C Code: a thorough introduction*[11]. Readers who are already familiar with concurrent programming can probably skip that chapter, or only read sub-sections they feel are interesting. After that, we describe the previous work this project builds upon, namely relaxed semantics and the 2D framework.

2.1 Concurrent programming

Concurrent programs are those in which different parts of the code executes asynchronously in different threads (which *can* be mapped to different cores). There are different reasons why to do this, but in this thesis, we focus on the most obvious one which is increasing performance. Many computers nowadays have several cores and unless you have concurrent code your machine will largely only use one of those cores at a time, which is a waste of potential.

In the ideal case, we can let each thread work on completely different parts of the program, not reading and modifying the same data. Writing code for that is just like writing normal code, except you tell each thread which part to run. But in reality, it is required for the threads to communicate and work with shared resources, such as a queue or stack, which is what makes the field complex.

One thing to keep in mind when doing concurrent programming is that the executions are not deterministic. In a single-threaded program, all instructions are (conceptually) executed in order, and thus the same thing will happen every time unless you use some randomness. But when you have several threads the output can depend on the interleaving and timings of the threads and their memory accesses. Thus the program may run fine and pass a test on the first 100 tries, but fail on the 101st due to some hard to spot bug. It can also be very

dependent on the computer used and work fine on your computer, but when someone else tries, it crashes immediately. This makes the need for formal proofs and reasonings about correctness more important in concurrent programs than in normal ones.

2.1.1 Correctness in concurrent programs

In this section, we look at what challenges we will face when writing concurrent code, as well as how to work around them. Here we will approach it from the C language as that is what is used in the project, as well as it being quite explicit with what happens in the machine. But most ideas here are also directly applicable to most other modern languages.

Atomic instructions

A simple example of why concurrent programs are complex to reason about is incrementing a shared variable, such as x++ or x+=1. You might think that this is just an atomic instruction which increments x by one. But what happens is that the thread reads $y := x$, computes $z := y + 1$ and stores $x := z$. This can cause problems as shown in Algorithm 1 where two attempts at incrementing a variable only lead to it being incremented once. Behaviour like this which depends on the timings of different threads is called a race condition and should (almost always) be avoided.

Algorithm 1: Data race when incrementing a shared variable. Two threads try to increment x, but at the end, x will only have increased by 1.

Thread A	**Thread B**
read(x)	
compute($x + 1$)	
	read(x)
	compute($x + 1$)
store($x + 1$)	
	store($x + 1$)

This problem stems from the fact that normal code instructions are not necessarily atomic, but rather are split into several smaller instructions in the hardware. This discrepancy motivates the need for special instructions in the hardware which can give the programmer some promise of atomic behaviour. The most important function is that we want to be able to read and write some memory without any other thread accessing that memory between our read and our completed write. We call it an *atomic* instruction as it can be seen as a taking effect in a single instant.

The most common and basic such instruction is what is called compare-and-swap (CAS). What it does is simply write some value to a memory location if and only if the memory location contains the expected value. The idea of it is visualized in Algorithm 2. Note that this requires extra hardware support and would not work if implemented directly in software as we then would run into the same issue as in Algorithm 1. The most basic use case for it is to read some value (commonly a pointer), calculate the next value (maybe a pointer to some new object), and try to replace the old pointer with the new one.

Using CAS you can implement other things such as mutual exclusion with locks which

Algorithm 2: Compare and Swap idea

```
if x = x_old then
    x ← x_new;
    return true;
else
    return false;
end
```

is very widely used. The idea then is that only one thread can hold the lock at a time and then access some shared state. Since the shared state can only be accessed while holding the lock it makes all such accesses seemingly atomic. A lock is conceptually just a boolean and to acquire it you must update it from ***false*** to ***true*** with a successful CAS, and when you are finished you set it to ***false***. The reason this is so popular is that it is very easy to use. You can basically take any normal data structure, like a queue or stack, decorate it with a lock at all access points and it will be a valid concurrent data structure.

When talking about atomicity the term *linearizability* often comes up[11, 7]. This is a property of a method operating on some state or object. Essentially it means that the method must seemingly take effect instantaneously sometime between its invocation and return. That instant is often called the *linearization point* and can be very useful when proving the correctness of concurrent data structures. An example is a shared counter updated using CAS. Then the linearization points are the moment you successfully complete the CAS for an update, and every time you read it for a read. For something with a lock the linearization point can be viewed as any time between the acquisition and release of the lock, but maybe simplest is to think of the instant of release as the point where all changes become visible at once.

Compiler and hardware optimizations

Modern compilers are very sophisticated and can make the code you write significantly faster using different optimization algorithms. These optimizations make sure to not change the result of the code, but unless specified the compiler assumes the code is to be run in one thread. The same thing goes for modern hardware which also has many optimizations to improve performance. This means that if you have several threads these optimizations can severely break your program. Here we will look at some simple examples of what can go wrong to hopefully give some intuition for what to keep in mind when writing concurrent code on a shared memory system.

One danger is register allocation of shared variables. If the code reads a variable ***x*** and later reads it again without having changed it, then it can just assume it has not changed and use the old value. By saving the value in a local register it thus can avoid the second read from memory. This can cause issues as seen in Algorithm 3 where one thread can get stuck forever, not realizing a variable has changed. Both reads and writes are in danger of this, for example, a thread might not immediately write something to memory, but keep it in a register for a while until it is time to write it to memory. This shows that we must be wary to specify which variables can be changed from different threads. In C this can be done by defining a variable as *_Atomic* (since C11) or *volatile* which is older but has a similar effect.

Algorithm 3: Thread B can get stuck in an infinite loop if it register allocates $x = 0$.

Thread A	**Thread B**
	$x \leftarrow 0$
$x \leftarrow 1$	
	while $x = 0$ **do**
	`/* Can be stuck forever. */`
	end

Another tricky issue is that of ordering operations, in which both the compiler and hardware are involved. The compiler is allowed to change the order of operations as long as it behaves the same, and it assumes the code is for one thread unless otherwise specified. The hardware can start several instructions in parallel and since different instructions take different amount of time the hardware might want to start on future instructions before all the previous ones have been completed. Together they will do this to not affect the single-threaded execution, but threads observing each other can see very strange orderings. Take the example in Algorithm 4 where the instructions can be re-ordered to cause the second thread to seemingly observe an impossible state.

Algorithm 4: By reordering the read of y in Thread B to within the loop, or by re-ordering the writes in Thread A, Thread B can read $y \neq 1$. Both x and y are _Atomic.

Thread A	**Thread B**
$y \leftarrow 1$	**while** $x \neq 0$ **do**
$x \leftarrow 1$	`/* Wait */`
	end
	`/* Can read` $y \neq 1$ `*/`
	read(y)

A healthy way to think about these re-orderings is that once one thread looks at a sequence of actions by another, they can be completely re-arranged without almost any promise of logical order. This is of course disastrous for writing correct concurrent programs and thus there are ways to enforce orderings within concurrent programs. One approach is using something called a *memory consistency model*, which is described further down. The benefit of using them is that you don't have to keep track of all possible re-arrangements, but instead force the compiler and hardware to not violate certain important orderings within the program.

Memory consistency models

To enforce orderings in the program we utilize what is called memory consistency models. These allow us to create happens-before dependencies between different points in the code. A happens-before relation is exactly what it sounds like and guarantees that one thing happens before another thing, for all observers, which is incredibly powerful. If we know x happens before y then when observing x we know everything done before y must also be visible. Take Algorithm 4 again as an example. Here we can specify that the two writes in Thread A must be ordered and that the reads in Thread B must be ordered, and then we would never observe the inconsistent state. But relaxing either one of those orders could cause the inconsistent

observation.

There are several consistency models to choose from and they are all just different ways to specify where there are happens-before dependencies. In languages such as C, the base memory order gives no guarantees except intra-thread dependencies, but you can specify different memory consistency orders for the access to _Atomic variables which can also be called *synchronized variables*.

The most basic and strong order is *Sequential Consistency*[11] (SC). Conceptually it forces every instruction on synchronized variables to occur in the code order, and no two instructions can occur simultaneously. Furthermore, no normal memory access can be reordered past the access to a synchronized variable. A great aspect of SC is that it creates a global order of accesses to the synchronized variables, even if they are not related. So if one thread sees x happen before y then no other thread will see y happen before x. The price it pays for being very strict is that it prevents a lot of possible optimizations and slows down the program if used excessively. However, if used sparingly it can be very powerful as a global order of all accesses to all synchronized variables is useful when reasoning about correctness.

Another ordering is *Weak Ordering*[11] (WO) which as its name implies is less strict. Here all accesses to any synchronized variable are globally ordered and like in SC no normal memory access can be reordered past the access to a synchronized variable. The big difference to SC is that there is no total global ordering of accesses to all synchronized variables. So each synchronized variable has an ordered history of accesses which all threads will observe identically, but when observing the global order of all accesses to synchronized variables one thread might see a different interleaving order between each variable than another thread. This weakening of consistency enables the programs to become a bit faster, but reasoning about the correctness of the program becomes harder to prove.

ABA problem

The ABA problem is a common problem in concurrent programming and has to do with an inherent limitation in CAS. Programs relying a lot on CAS to update data, such as the algorithms in this thesis, constantly have to think about this problem. The problem is that the use case of CAS is often that "If the value of x has not changed, let $x \leftarrow x_{new}$", but what it actually does is instead "If the value of x is $x_{expected}$, let $x \leftarrow x_{new}$". This means that the CAS can complete, even if x has been changed since i was last read, just as long as it has been changed back to the same value. This can have devastating effects, for example when combined with pointers. Say you want to change some pointer to some immutable data, given some aspect of the data and you update it with CAS. Then the pointer might have changed, the original object garbage collected, the memory reallocated as a new object and the pointer to it written in the location we are looking at. In that case, the pointer will still be the same, but what it points to might be completely different, leading our update to succeed on completely false assumptions and could in a good scenario cause a crash.

The most common way to circumvent the ABA problem is to use so called ABA counters together with the pointer value x you want to update. Then each time you want to CAS x you increment the counter and CAS both x and the counter together. Since the counter is always increasing, this makes sure that the CAS only succeeds if there has been no change since you observed it last.

One problem with this is that the counter is of course not unlimited and when it overflows

it can wrap around to 0. Thus it is possible that you observe a counter with the expected value, but that it has just changed so much that it wrapped back around to the same value. This is unlikely, but as you often want to decrease the size of the counter to increase efficiency you can potentially run into problems. This makes all programs using ABA counters in a way not completely safe, but it would be incredibly unlikely a problem would arise unless using a too small counter.

Proving Correctness

There are many ways of proving that something is correct in concurrent programming and no proof strategy has taken hold as the de facto one to use. However, proving linearizability is often the start of proving correctness[7].

One way to prove correctness is to reason about the order between different operations. As earlier discussed, order in concurrent programs is less strictly defined and not all instructions in different threads need to be ordered. But if we assume all memory accesses on shared data are synchronized with sequential consistency then the set of all instructions is a partially ordered set, ordered by happens-before relations.

A known fact is that each partially ordered set can be extended to a totally ordered set[7]. This means that for each such program execution there must be at least one way to extend all instructions to a totally ordered set. Thus when using SC on the shared data it is possible to reason that either x happens before y, or y happens before x. So even if there is no imposed order between two operations (x and y), there must be at least one way to order them consistently with the rest of the history. Another way to look at this is that each concurrent (partially ordered) history can be rearranged into at least one linear (totally ordered) history. This can be a useful tool in proofs by contradiction.

2.1.2 Efficiency in concurrent programs

Just writing correct concurrent programs can take a programmer a long way. But to make concurrent programs fast there are several points to keep in mind. Here we will look closer at how to write code to optimize memory accesses and to make sure threads are not blocked unnecessarily. We assume the computer used has several cores.

Memory hierarchies

Memory is not just one big central array of storage locations. Reading from the central memory is slow and thus modern computers have several layers of caches between the main memory and where the computations are done which can store a smaller part of the memory and enable faster access. Caches use a cache coherency protocol to guarantee that there are not two different valid values of a variable in two different caches. When you try to access a variable in a cache, but find that it is not there, or that it is outdated, the computer has to fetch it from further away and we call it a cache miss. These cache misses take a lot of time and can be on the order of hundreds of normal instructions. Therefore it is very important in all high performing code to think about the cache performance.

Each core has its own small cache with fast access, so if a core wants to access a variable written by another core it will get a *cache miss*. A cache miss forces he core to fetch the

memory from some higher lever cache, which takes time and can significantly slow down the program. So threads on different cores working on the same data will inevitably slow each other down. Instead, we want as much as possible for each thread to work on private data which no other thread is modifying, to avoid as many cache misses as possible. This concept is called *data locality* and is important when designing multicore algorithms[10].

Instead of only considering one processor with several cores it is also common for computers to have several processors. Each of these processors then has its own cores and caches. Communicating between two such processors takes much longer than communication within one of them and thus we say that such an architecture uses non-uniform memory access (NUMA). If instead only using one processor we say it has uniform memory access (UMA). In NUMA settings data locality becomes even more important as you want to avoid frequent communication between processors as much as possible. The difference between UMA and NUMA is quite large and it can often be faster to run say 8 threads on one of the processors and let the other be idle than run 8 threads on both processors. Ideally, you would write code in such a way that the threads tried to work in groups according to which processor they belonged, but this is not trivial and not often done.

Non-blocking code

Locks as described earlier are very simple to use, but they have an inherent drawback. When one thread holds the lock all other threads trying to acquire the lock are blocked and must wait for it to release the lock. If the thread holding the lock for some reason gets suspended, or for any other reason runs slower all blocked threads must wait an extra long time. This is often not a problem when there are few threads wanting to acquire the same lock but can cause big performance degradations when the amount of threads increase and the contention on the lock becomes high.

This is the reason the term *lock-free* (also called *non-blocking*) was introduced and is quite important. The definition from[7] is that if one or more threads invoke a lock-free function then at least one of them will make progress in a finite number of steps. Lock-free algorithms are potentially faster than blocking ones as they involve less time spent waiting for other threads. However, they can be more complicated to design as several threads can work on the same things concurrently. It also means that the data structures are often in non-consistent states which the threads together have to make consistent. These algorithms often rely on CAS to swap out key parts of the shared state by some strategy and there are often races between threads to see which one will finish an operation first, prompting the other to potentially discard their work. Classic examples of lock-free algorithms are the Treiber stack [12] and the MS queue [8].

CAS implementation details

Compare-and-Swap is an essential tool when it comes to lock-free programming. The promise of atomicity is quite costly and it scales badly with the size of the memory location to operate on. If used on a single word (commonly 64 bits) CAS usually has quite good hardware support, but on larger objects, the performance may vary. On processors with a reduced instruction set like PowerPC, there is usually no built in direct support for larger than single word CAS[11], and thus you want to stay within a single word as much as possible. On proces-

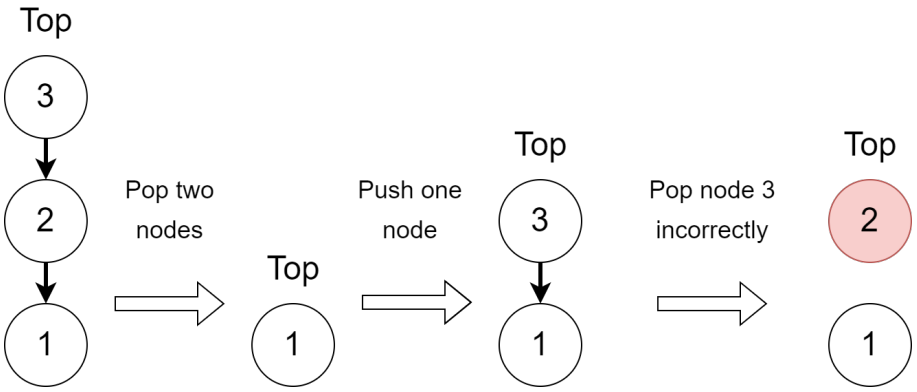


Figure 2.1: Visualization of the ABA problem with the Treiber stack. The numbers on the nodes signify which memory location they have.

sors with more complex instruction sets, like Intel processors, there can be support for wide CAS[1], updating two adjacent words concurrently without taking too much extra time. So the use of a wide CAS might be fast on some computers, but very slow on others. In this thesis we will allow the use of wide CAS as the preexisting code from the 2D framework[10] requires x86 and already using several wide CAS.

2.1.3 Classic lock-free data structures

Here we present two classic lock-free data structures which are used as components in the 2D framework[10] as well as in this thesis.

Treiber stack

The Treiber stack[12] is a very simple lock-free stack. Internally it is just a linear linked list where we add and remove nodes at the head. The shared mutable state is the so called *descriptor* which keeps track of the top node (the head) of the stack.

To add an element to the stack you create a node and set its ***next*** field to the current head of the stack. Then you try to update the descriptor of the head of the stack to point at the new node. If it fails you just update the ***next*** field to the newly observed head and try again until success. To remove an element you just read the ***next*** field of the head, and then try to update the head descriptor to the read ***next*** node with a CAS, retrying until successful.

A naive implementation of a Treiber stack is a good showcase of the ABA problem which was described earlier. Since the ABA problem is important for the design of our data structures, we use this as a chance to further explain it and visualize it with Figure 2.1. Assume thread A wants to pop and reads that the top pointer points to 3 and that the next node is 2 in the first state. Simultaneously, thread B pops both 3 and 2 and finally pushes 3 again. If after that thread A proceeds with the pop and tries to linearize with a CAS it will succeed, since the top is 3 now, just as when the thread observed it last. However, the next node is no longer 2, which is what thread A will set the new top as, leading to the final state in the figure. Since node 2 is no longer a valid node on the stack the whole stack completely breaks. By letting the descriptor contain both the top pointer and a ABA counter we solve this problem.

MS queue

The Michael & Scott queue[8] (MS queue) is a slightly more complex lock-free implementation of a FIFO queue. It is not necessary to understand all of it for this thesis, but an idea of how it works is required as it is used as an internal queue in our proposed elastic queue. The difference from the Treiber stack is that we now must operate on both the head (where nodes are dequeued) and tail (where nodes are enqueued). Like the Treiber stack it is a linked list, but now with one descriptor for each end. The next pointers inside of the list must also now be considered shared to be able to enqueue items. Its descriptors also contain a pointer and an ABA count, but when describing it we will omit the counter to make it more concise.

To enqueue an element we should add it to the tail of the queue. To do this we both need to update the tails ***next*** field as well as the tail descriptor, which is done in two steps as they are separated in memory. First we update the ***next*** field of the tail and if successful we continue to try to update the descriptor. If some other thread managed to first change the ***next*** pointer of the tail, you first help that enqueue to update the tail descriptor to the ***next*** node of the current tail, before retrying the enqueue. So after updating the ***next*** field of the tail the queue is in an inconsistent state until one of the threads fixes it by updating the descriptor to the new tail, allowing further enqueues.

Performing a dequeue is almost identical to popping an element from the Treiber stack. You read the heads ***next*** pointer and try to update the descriptor to it with a CAS, given that it is not NULL as the queue then is empty. Here it also checks if the head and tail descriptors point at the same node, in which case it tries to help the pending enqueue if present before proceeding.

An interesting part of the queue which is exploited in this thesis is that the head does not actually point at the next node to dequeue, but rather the most recently dequeued node (which in turn has a pointer to the conceptual head). This is to avoid the queue ever being completely empty and both descriptors having a NULL pointer. In that case, it would be hard to add an element as it would want to be added to both ends concurrently. If instead the queue is empty both the head and the tail will point at the most recently dequeued node.

2.2 Semantic relaxation

The term *semantic relaxation* is used in concurrent data structures to mean that the original specification (semantics) of the data structure has been relaxed in some way[10]. Commonly this involves allowing operations on the data structure to execute out of order. This can allow an operation on a data structure to not return the expected element (like the topmost one on a stack) but instead a sub-optimal element (like one of the topmost on the stack). By not forcing every thread to access the data structure through the same element we should be able to improve performance[6].

The main relaxation we will look at in this thesis is the *k-out-of-order* relaxation for the remove operations. This is the relaxation used in the original 2D framework[10] and this thesis continues with the same foundation. If applied on a stack for example the relaxation does not force the pop operations to remove the top element, but instead, it can skip up to k elements and return the $k + 1$'th element. Each removed element has a corresponding *rank error*[13] which is how many higher priority elements were skipped in favour of the chosen

one. By designing data structures around this relaxation you can achieve higher throughput as seen in earlier papers[10, 6, 5].

2.3 2D framework

This thesis extends the previously published 2D framework[10], which in turn uses the Treiber stack[12] and MS queue [8]. Here we present the important ideas of the 2D framework which are needed for the ideas in this thesis. However, if the reader wants to get the full picture or another presentation we recommend reading through sections 3, 4 and preferably 5 in the original paper[10]. The framework presents concurrent k-out-of-order relaxed implementations of stacks, queues, deques and counters and gives analytical bounds on the worst-case rank costs. All their data structures use the ideas of multiple sub-structures synchronized with a global window and all updates are done with CAS using sequential consistency (which this thesis will continue using). We will start by introducing their queue and then generalize the ideas to their most successful stack. We will skip the decoupled stack, deque and counter as they are out of scope for this thesis.

2.3.1 2D queue

The main idea of the 2D queue comes from having several classic queues in parallel (call them *sub-queues*), and then the threads can spread out their operations over the sub-queues to decrease contention. If there are two sub-queues and two threads for example each thread could take its own sub-queue and mostly work on it, avoiding contention. This is also what introduces the relaxation as we then are not sure the items are enqueued and dequeued in the correct order. They have chosen to use MS queues for all sub-queues which we continue with in this report, but it could also use another queue.

To not let the rank errors be infinite there must be some way to synchronize the sub-queues so that they have made similar amounts of enqueues and dequeues. This is done by encoding a counter into the head and tail descriptor of each sub-queue which keeps track of how many get respectively put operations have been done on that sub-queue. We call the *row* of a node the count at which it was inserted. Note that this count is the same as an ABA counter which we would need anyway! Then there is a global *window* which specified in which ranges they are allowed to operate on the two ends of the sub-queues. The *put window* specified the maximum count to which nodes can be enqueued on each sub-queue, and the *get window* specifies the maximum count where nodes can be dequeued. For example, you are not allowed to dequeue from a sub-queue with more than n prior dequeues. Then when all sub-queues have reached this amount of dequeues/enqueues the threads together help shift the window up to some higher count which now becomes the limit. This is visualized in Figure 2.2 where several of dequeues are made and the window shifts up one step. The distance the window is shifted up is called ***shift*** and the range of each window is called ***depth***. For the queue the ***depth*** will automatically become the same as ***shift***, so its is referred to as ***depth***.

When one thread wants to do an operation it must search for a valid sub-queue concerning the current window. The search should minimize contention and maximize data locality. The first way this is done is that each thread always goes back to the sub-queue it last successfully updated so that hopefully each thread can do several local updates before having to

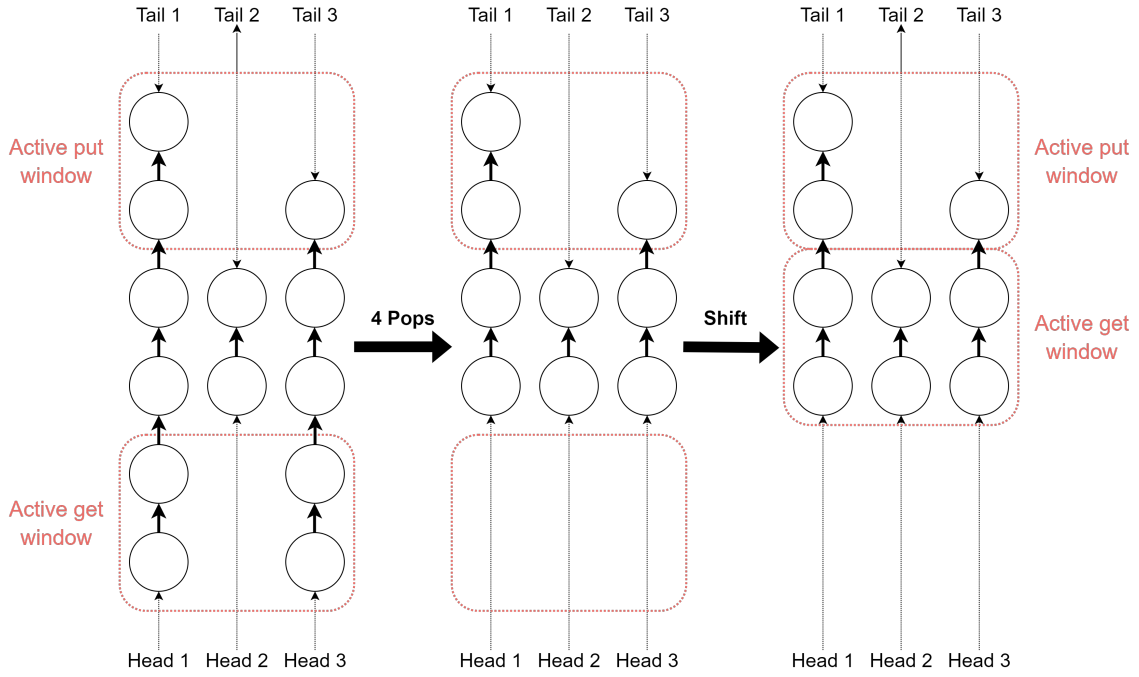


Figure 2.2: A 2D queue where four dequeues lead to it shifting its window up to allow more dequeues on all sub-queues.

switch sub-queue. Furthermore when there is contention on the sub-queue or the descriptor hits the window max then the thread does a few random jumps to try to find a new valid descriptor, and if no one was found it searches for one in a round-robin fashion. If it reads all descriptors in a round-robin fashion and finds that they are all full, it tries to shift up the window by ***depth*** and then again look for a valid sub-queue.

As an emptiness check, the queue can only return NULL if it during its search for a valid sub-queue observes all sub-queues as empty. Note that this is not a linearizable emptiness check as the sub-queues are checked sequentially and not in a single instant. Thus there might at all instants be some queues which are not empty, but when we observe them they are empty and some other queues instead have got elements.

The maximal rank error is decided by the ***width*** and ***depth*** of the window and is given as $(\textit{width} - 1)\textit{depth}$. This follows from all nodes on each sub-stack being completely ordered, while a node can be ordered before or after any other node in each window.

2.3.2 2D stack

The ideas for the 2D stacks are very similar to the queue, but since all operations access the same end of the list there are a few differences. Primarily they present two strategies for windows and sub-counts. The first and best performing one is called 2D coupled (2Dc) window and it only has one window which states in which size ranges the sub-stacks are allowed to be. The second one called 2D decoupled (2Dd) window is more similar to the queue and keeps two counts in each descriptor, one get count and one put count. It then also has two shared windows which separately provide ranges on the number of allowed puts and gets on each sub-stack. The coupled window leads to a bit fewer window shifts and is conceptually simpler so we will only consider it in the rest of this thesis, however it does have

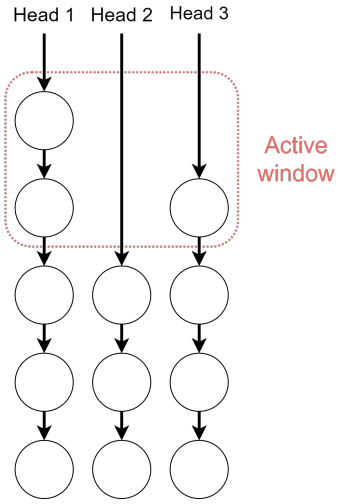


Figure 2.3: The 2Dc stack only has one active window which bounds the size of all sub-stacks both upwards and downwards.

a drawback of being harder to generalize to a deque. A visualization of the 2Dc stack can be seen in Figure 2.3.

For the 2Dc stack the distance to shift the window ***shift*** will not be the same as ***depth*** as nodes can both be pushed and popped in the window. If ***shift*** = ***depth*** it would always shift up from a (practically) full window to an empty one and then maybe directly shift down. Instead we set ***shift*** = ***depth***/2. The stack searches for valid sub-stack in the same pattern and returns NULL in the same way as the queue. It now uses Treiber stacks [12] for the sub-structures and uses its methods to push and pop. The final difference from the queue is that it needs an ABA counter for the shared window and sub-stack descriptors as their counts now no longer are monotonically increasing.

The real major difference between the coupled window of the 2Dc stack and the decoupled windows of for example the 2D queue is the interleavings of operations and the relaxation behaviour. If we in the queue observe that the window is full we know that it will never become not full again until the window is shifted up. However, in the 2Dc stack a thread might observe all sub-stacks at the top of the window, but the CAS to shift the window up might not go through before the linearization of several pops which made the old window empty. This would lead to a state where all the heads of the sub-stacks are below the bottom of the new active window. Importantly here we allow pushes if the descriptor count is *below* the upper limit and pops if it is *above*, not necessarily inside. This allows sub-stacks to temporarily be outside the active window after shifts, which makes the worst-case relaxation errors a bit worse. This also makes it harder to extend the stack as it not just grows in one direction monotonically but instead goes back and forth.

The maximal rank error for the 2Dc stack is decided by the ***width***, ***depth*** and ***shift*** of the window. It is not as intuitive as the queue but is given as $\left(2\boldsymbol{shift} + \boldsymbol{depth} + \left\lfloor \frac{\boldsymbol{depth}-1}{\boldsymbol{shift}} \right\rfloor \boldsymbol{shift}\right)(\boldsymbol{width}-1)$. This is actually not proven in the original paper due to an oversight, but we give a corrected proof with this bound in Theorem 6.

Chapter 3
Related work

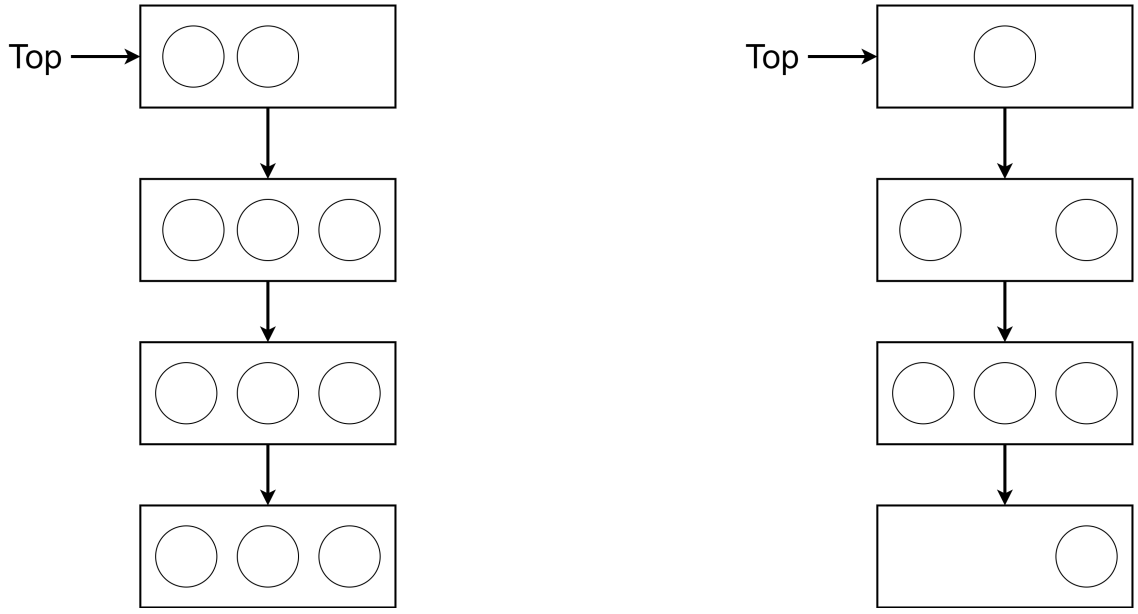
In the last decade, there have been several good articles on semantic relaxation in data structures ranging from theoretical to implementation based. Here we present some related papers which can be good to read for a deeper understanding of the field. All of the papers share the fact that the relaxation behaviour has to be specified up front, and cannot be changed during run-time, just like the original 2D framework[10]. Thus it would be interesting to extend all of them to be able to change the relaxation just like this thesis does for the 2D framework[10].

3.1 K-segment stack

The k-segments stack [6] was introduced in 2013 and is, just like the 2D stack[10], a k-out-of-order relaxed stack. It is a stack of segments each containing k unordered elements, leading to the out of order relaxation. When doing an insertion or deletion it always tries to do it on an arbitrary index in the top segment, avoiding conflicts and decreasing contention. A visualization can be seen in Figure 3.1a. The paper also contains theoretical definitions of several different relaxations as well as concurrent ordering relations.

The complicated part with this stack is the addition and removal of the k-segments. If you want to push an element but the top segment is full you have to push a new k-segment, or remove the top segment if it is empty and you want to pop. When pushing a segment there can be concurrent pop operations on the past top segment, making it not full anymore and can take a state like in Figure 3.1b. On the other hand, if you want to remove the top segment and another thread does a concurrent push to it there is a problem, as then the element would be lost. Therefore the stack has to do extra checks when removing segments and adding elements which counteracts the simplicity of the idea.

The k-segment stack is very similar to the 2D stack with a depth of one. But by not having the segments as explicit storage locations the 2D stack can avoid these problems with removing segments.



(a) Perfectly filled. **(b)** Not perfectly filled.

Figure 3.1: Visualization of the k-segment stack. Both the ideal case is when all non top segments are full, as well as when concurrent operations make them not full.

3.2 Distributed queues

From many of the same people as the k-segment stack came a paper about relaxed queues in 2013 called Distributed Queues in Shared Memory [5]. The paper introduces two distributed queue (DQ) algorithms which both build upon having a shared array of sub-queues to spread out contention, just like the 2D framework[10]. The difference between the algorithms comes from how they select which sub-queue to operate on.

The first algorithm uses load balancers to try to spread out the operations over the queues. One load balancer samples several queues at random and then proceeds to operate on the one with the fewest (most) element for an enqueue (dequeue). To quickly determine the approximate number of elements in a queue they simply take the difference between the ABA counters at the head and tail of the queue. The other load balancer maintains a set of index pairs, each assigned to one or more threads. When a thread wants to do an operation it increments its assigned counter for that operation atomically and proceeds to do the operation on the index the counter was incremented from. Both approaches can achieve good speed-ups and enable the user to choose the amount of relaxation.

The other algorithm instead maintains the invariant that the ABA counter across all heads (or tails) should be at most one apart. This is easy to maintain since if you observe a counter with value n and then one with $n-1$ you know it is safe to operate on, and increment, the second one. This idea does not scale as well under high contention as the load balancers, but it is a very simple idea. One reason it does not scale well is that you cannot change the degree of relaxation and it is always quite low. It is interesting to think about how effective it would be to change the invariant to maintain a difference of for example two, or any positive number, instead of one.

3.3 MultiQueues

An efficient and simple relaxed priority queue is presented in the first MultiQueue paper[9] from 2014. It is very similar to a special case of the first load-balanced distributed queue and is made up of several priority queues in parallel. Inserting elements is done into a random queue. Removing an element on the other hand selects two random sub-queues and tries to remove an element from the one with the highest priority top element. This best of two choices when removing acts as a regulative force to keep the expected rank errors from diverging.

As this algorithm relies heavily on randomness and gives no real promises on how large the worst case rank errors are. The original paper[9] has some analysis but does not answer all the questions. For that, the paper on the Power of Choice [2] came out in 2017 and did a theoretical analysis of a slightly simplified MultiQueue. Its most important result was a theorem that stated that the likely rank error of a removed element is proportional to the number of sub-queues, both in expectation and in high probability. This means that is is unlikely that the rank errors will diverge over time.

An extension to the original paper from 2021 [13] introduced buffers to both ends of the queue to improve cache behaviour. In a normal priority queue, you have to traverse nodes, probably allocated in different cache blocks, to do an operation. Since they work on so many different queues many of these reads of nodes will be cache misses. What they do instead is keep a small buffer of nodes in an array before and after the queue so that most operations only will operate on the buffers and not start to traverse the queue, which should reduce cache misses. This noticeably sped up the algorithm and shows how important it would be to think about cache behaviours in these concurrent algorithms. An influence of this can be seen later in how our elastic queue handles elastic changes.

3.4 Time-stamped stack

The time-stamped stack [4] from 2015 does not use relaxed semantics, but it showcases that a concurrent stack can be seen as correctly ordered even if not totally ordered. It too uses a shared array of sub-stacks to spread out contention, but the key idea here is that after successfully pushing an element it is given a time-stamp. Each thread pushes elements to its own sub-stack, but when removing an element it must search the sub-stacks for the newest element and try to remove it.

One positive is that each sub-stack is totally ordered, and thus they must only look at the top element of each such stack when removing. The other is that if a thread reads a node that has not yet been given a time-stamp it can be viewed as concurrently being inserted, which means it is safe to remove. There can be several such elements at the same time which means they are not totally ordered, but they show that it is always possible to construct a correct linear history from the concurrent history. This shows how important it is to let to of the idea that all events must be totally ordered in a concurrent setting.

Chapter 4
Elastic designs

Here we describe the novel ideas presented in the thesis. They are all extensions to the 2D framework [10] and provide ways to change the relaxation during run time. The 2D framework supports changing the sub-structures to different ones, but we continue using the MS queue and the Treiber stack. There are only two parameters controlling the relaxation, namely the ***width*** and the ***depth*** and our extensions implement the functionality to change those variables in future windows.

One solution to making the width elastic is to redistribute the nodes to new sub-structures when the width changes. This could take a long time and block all other operations on the data structure, reducing its responsiveness. Instead, our designs never alter inserted nodes, limiting the possible designs to ones that can only change how nodes are inserted in the future.

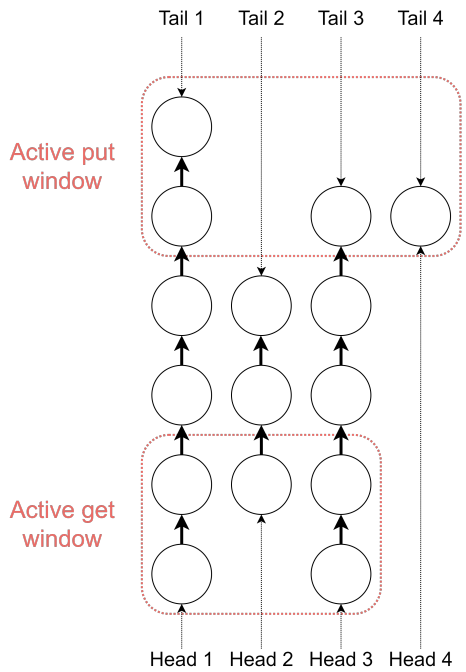
4.1 Elastic 2D queue

The 2D queue is in one way inherently easier than the stack as it always only visits each window once, instead of moving back and forth. Thus we start with presenting the design for it.

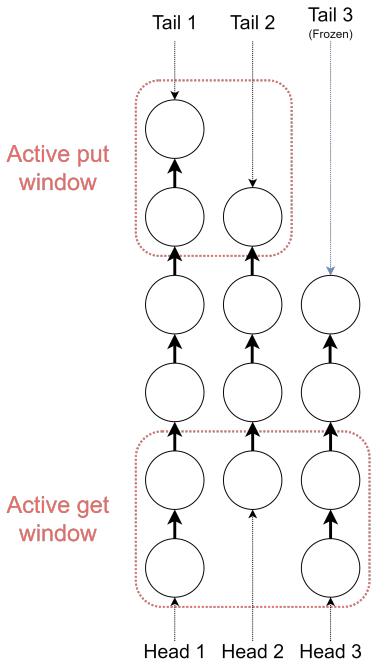
Changing the depth during run time is quite simple for the queue with its separate windows as ***depth*** is only read when shifting the windows up and never written to after initialization. Thus we can do as the original algorithm but allow concurrent updates of the global ***depth***, with a sequentially consistent CAS. Then once one thread reads the new global ***depth*** and uses it for a shift, all future shifts will see that new ***depth***, or a newer one.

To be able to change the width it is not enough to just change a variable somewhere, making it more complicated than the depth. As discussed before we will not try to re-arrange the inserted elements in the sub-queues. Instead the number of sub-queues which are operated upon will change over time.

Since no elements will be rearranged inside the sub-structures and the relaxation must



(a) Floating sub-queue after increasing width.



(b) Frozen sub-queue after decreasing width.

Figure 4.1: Visualization of how we conceptually could try to increase or decrease the width of a 2D queue, starting from width 3. In both cases, the get window must catch up for it to change width.

be kept bounded the only real option is to change the width of the put window. Then over time as the get window catches up, it will also change its width as needed. In the original 2D queue, each thread goes back to the sub-queue it successfully operated on last, independent of which type of operation it was. This becomes problematic for an elastic queue as the put and get windows can have different ***width*** and thus an index can be in bounds for one end of the queue (one operation) while being out of bounds for the other. Instead, we chose to *decouple* the saved indexes so when a thread does an enqueue it goes back to the sub-queue where it succeeded with an enqueue last, and the dequeues go back to the index with the last successful dequeue.

When we increase the width of the window we want to operate on more sub-queues to the right of the previous ones. But these new sub-queues will not have been filled with elements like the others and thus have gaps in them where they were inactive. This is visualized in Figure 4.1a where the width is increased and thus there is a gap under the first element in the fourth queue.

We can do a similar technique for decreasing the width and disallow insertions into the sub-queues we are trying to remove. We can think of it as freezing them which is visualized in Figure 4.1b.

To get this working we want to combine these ideas into a place where we can first decrease the width, then increase and so on, making the sub-queues possibly full of gaps. The conceptual idea should work for both the stack, queue and deque as they all are just versions of linked lists, or some collection of linearly sorted elements. However, they are inherently different in their access patterns and an implementation idea which is good for one of them

might not work well for another.

4.1.1 Implementation

This design builds upon the idea of characterizing the gaps (from changes in width) in the sub-queues as actual nodes, called *gap nodes*. These nodes are inserted during put operations when the width of the put window is changing. Then, when the get window reaches the part of the queue with the gap nodes, it knows to change to a new width. Pseudocode for the structures used, window searches and extra functions can be seen in Algorithm 5, 6, 7 respectively. One large change from the original 2D queue is that its windows only contained the variable ***max***, while ours now also have ***width*** and a couple other variables which will be explained below. The algorithms will be explained below but by having the pseudo code in parallel we think the ideas can be easier to grasp.

Algorithm 5: 2D Elastic queue structs

Struct *Des*
- Node* node
- Int getcount
- Int putcount

Struct *PutWindow*
- Int max
- Int width
- Int active_width

Struct *GetWindow*
- Int max
- Int width

Struct *NormalNode*
- Node* next
- Item content

Struct *FreezingNode*
- Node* next
- Int start_width
- Int start_count

Struct *FloatingNode*
- Node* next
- Int stop_width
- Int stop_count

Algorithm 6: Elastic queue window search algorithm

```
Function WindowSearch(op, index, cont)
    if cont then
        index ← RandomIndex(width);
    end
    if op = put then
        return PutWindow(index);
    else
        return GetWindow(index);
    end

Function PutWindow(put_index)
    Random ← 0;
    SYNC_WIN(LpWin, pWin, put_index);
    while True do
        Des ← putArray[put_index];
        if LpWin ≠ PWin then
            SYNC_WIN(LpWin, pWin, put_index);
        else if Des.putcount < LpWin.max then
            if put_index ≥ PWin.active_width then
                InsertFreezing(Des, put_index);
            else if Des.node is FreezingNode then
                InsertFloating(Des, put_index);
            else
                return {Des, put_index};
            end
        else if hops < LpWin.width then
            HOP(LpWin.width, put_index);
        else
            NWin.max ← LpWin.max +
              depth_global;
            NWin.active_width ← width_global;
            NWin.width ←
              max(NWin.active_width,
              LpWin.width);
            CAS(pWin, LpWin, NWin);
            SYNC_WIN(LpWin, pWin, put_index);
        end
    end

Function GetWindow(get_index)
    Random ← 0;
    SYNC_WIN(LgWin, gWin, get_index);
    while True do
        Des ← getArray[get_index];
        putcount ←
          putArray[get_index].putcount;
        limiter ← min(LgWin.max, putcount);
        if LgWin ≠ gWin then
            SYNC_WIN(LgWin, gWin, get_index);
        else if Des.getcount < limiter then
            if Des.node is not FreezingNode then
                return {Des,index};
            else if Des.node.next.stop_count ≤
              LgWin.max then
                DequeueFloating(Des);
            else
                IncDescriptor(Des, limiter);
            end
        else if hops < LgWin.width then
            if Des.getcount < putcount then
                notempty ← 1;
            end
            HOP(LgWin.width, get_index);
        else if notempty then
            NWin.max ← min(LgWin.max +
              depth_global, pWin.max);
            NWin.width ←
              AdaptiveWidth(NWin.max);
            CAS(gWin, LgWin, NWin);
            SYNC_WIN(LgWin, gWin, get_index);
        else
            return {Des,index};
        end
    end
```

In the queue each gap is made up of two parts, the initial gap node called the *freezing node* and eventually, once the sub-queue gets active again, the ending gap node called the *floating node*. As seen in Algorithm 5 they encode some information about the instant they were enqueued, which will be explained in detail later. When the width decreases we insert freezing nodes on all sub-queues outside the new width, and when the width increases we insert floating nodes into all newly activated sub-queues. Exactly how we insert these is explained in the paragraph below. Figure 4.2 shows an example of how a queue could look after decreasing or increasing width with these gap nodes. Since the width is coupled with the windows we ideally would want to atomically insert all these nodes when the window shifts and changes width, but this is practically difficult and would be too expensive, so we compromise. We introduce what we call a *transitional window* which is a put window which has changed width from the last put window. During this transitional window we make sure that all gap nodes are inserted before shifting away from it. The gap nodes are enqueued during the push operations when the thread finds an index without a gap node where there should be a gap node. Since the threads must visit all indexes before shifting we make sure all gap nodes will be inserted. Before the transitional window the insertions were done at the old width and after the transitional window the insertions will be done at the new width.

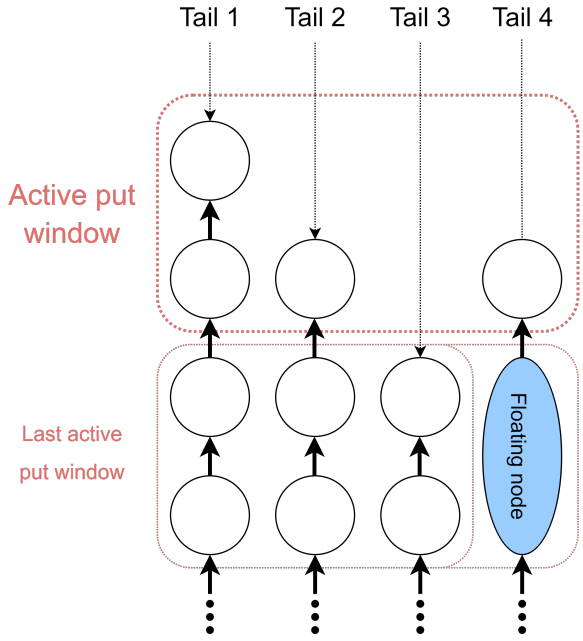
When inserting gap nodes we also include some extra information in them, to eventually be of use for the get window. For both nodes we save the new ***width*** in the fields ***start_width*** and ***stop_width***. We only want to insert one gap node into each sub-queue, but we still want to fill out the window. This is done by increasing the descriptors count directly from the bottom of the window to the top when inserting such a gap node, instead of incrementing by one. For both nodes we save this put count at which the gap is completed in its fields ***start_count*** and ***stop_count***. In the MS queue, other threads can help update the descriptor for halfway completed put operations, and normally the descriptor would just be incremented by one then. Instead, we now have a check if it is a gap node when helping an enqueue, and in that case, we update the count directly to the put window's ***max***.

Importantly each thread must know when it is in a transitional window to insert these gap nodes. When the ***width*** of the put window has increased the threads will find freezing nodes among the descriptors inside the ***width*** and then know it should complete the gaps by pushing floating nodes on those sub-queues. But when ***width*** decreases there are no such hints and the threads need extra information. This is done by encoding another width called ***active_width*** into the put window. The ***active_width*** is equal to ***width*** in all cases except in a transitional window when the width is decreasing, in that case, ***width*** is the old width, and ***active_width*** is is the new width we want to change to. So when *PutWindow* looks for a valid index to insert a normal node and notices it is outside the active_width it should insert a freezing node there before moving on. Respectively when it notices the last node was a freezing node it should enqueue a floating node.

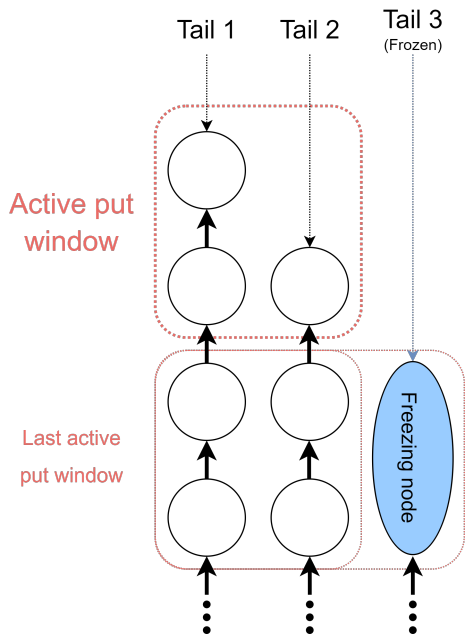
So far we have described how the gap nodes are inserted to change the width of the put window, so now we will move on to how they are dequeued and how the get window adapts its width using them. The idea is that the get window will look for gap nodes when shifting, and possibly change ***width***. We start with how the gap nodes are dequeued, before moving on to how they are used to adapt the ***width***. The important part when dequeuing gap nodes is that we need their information when shifting to the new width they encode, so we should not remove them before we have used this information. Here we utilize the fact that we are using MS queues and that its tail points to the most recently dequeued node (the *last head*). As long

Algorithm 7: 2D elastic queue helper functions

```
Function Enqueue(new_item)
    node ← Node(new_item);
    do
        {Des, index} =
          WindowSearch(put_index, cont,
          get);
    while !TryEnqueue(Des, put_index, &cont,
      node);
Function Dequeue()
    while True do
        {Des, get_index} ←
          WindowSearch(get_index, cont,
          put);
        node ← TryDequeue(Des, index,
          &cont);
        if node.next is not FreezingNode then
            return node.next.item;
        end
    end
Function AdaptiveWidth(max)
    width ← LgWin.width;
    alias edge = getArray[width - 1].node;
    while edge is FrozenNode do
        width ← edge.start_width;
    end
    alias edge = getArray[width].node.next;
    while edge is FloatingNode ∧ edge.gap_stop
      ≤ max do
        width ← edge.stop_width;
    end
    return width;
Function InsertFloating(Des, index)
    floating.stop_width ←
      LpWin.active_width;
    floating.stop_count ← LpWin.max;
    InsertGap(floating, Des);
Function InsertFreezing(Des, index)
    freezing.start_width ←
      LpWin.active_width;
    freezing.start_count ← LpWin.max;
    InsertGap(freezing, Des);
Function InsertGap(node, Des)
    CAS(Des.node.next, NULL, node);
    NDes.put_count ← LpWin.max;
    NDes.node ← Des.node.next;
    CAS(Des, NDes);
Function IncDescriptor(Des, max)
    NDes.node ← Des.node;
    Ndes.getcount ← max;
    CAS(Des, NDes);
Function DequeueFloating(Des)
    NDes.node ← Des.node.next;
    NDes.getcount ←
      Des.node.next.stop_count;
    CAS(Des, NDes);
Macro HOP(width, index)
    if Random < 2 then
        index ← RandomIndex(width);
        Random += 1;
    else
        hops += 1;
        index += 1;
        if index ≥ width then
            index ← 0;
        end
    end
Macro SYNC_WIN(LWin, Win, index)
    LWin ← Win;
    hops ← 0;
    if index > LWin.width then
        index ← 0;
    end
```



(a) Floating sub-queue after increasing width by inserting a gap node.



(b) Frozen sub-queue after decreasing width by inserting a gap node.

Figure 4.2: Visualization of how we increase or decrease the width of a 2D relaxed queue using gap nodes, starting from width 3. We just inspect the tail since the head only adapts when it catches up. In both cases, the last active put window can be viewed as a transitional window as it still knows what the old width was and has to insert gap nodes.

as the last head is not a freezing node we allow a normal dequeue of the next node as long as it is in the window, but if it turns out to have been a freezing node the dequeue operation is retried and the get count for the descriptor is updated to the window max. In actuality, we don't update it to the window max, as it could be higher than the put count for the index which can cause edge case problems, so it is updated to the minimum of the window max and the put count. After having dequeued a freezing node the queue can be thought of as being within a gap and we have easy access to both gap nodes, as the freezing node will be the last head. This design keeps us from unnecessarily having to read the contents of a node before dequeuing it which would introduce extra latency and thus failure chance to each dequeue. A floating node can be dequeued if its field ***stop_count*** is below or equal to the get window's ***max***, and then we update the descriptors count to the read ***stop_count*** and exit the gap.

Now we move on to how the gap nodes are used to change ***width*** of the get window, which also can be seen in *AdaptiveWidth* in Algorithm 7. Each time a thread tries to shift the get window up it looks at the two last heads at the edge of the current width. If the width is shrinking the last head at index $\boldsymbol{width} - 1$ should be a freezing node, which means that the sub-queue has entered a gap. Then the new ***width*** is read from that freezing nodes ***start_width***. This is repeated until we can't find a freezing node at index $\boldsymbol{width} - 1$. Then if the width is increasing the freezing node at index ***width*** should have a floating node as its next node, and that floating nodes ***stop_count*** should be smaller than the next get window's ***max***. Again the new width is read from the floating nodes ***stop_width***. This is also repeated until no such floating node is found. The reason these searches have to be repeated is that the elastic depth can make the get window to be much deeper than the past put windows, which means there could be several width changes in one get window. But by first checking for all possible decreases and then all possible increases we will not miss any so far observable changes for the next window. A larger example of both increasing and decreasing width can be seen in Figure 4.3 which can give a larger intuition for how the gap nodes are placed in the queue.

A hard to spot effect of the elastic queue is what might happen if the queue is almost empty and the width increases. Imagine that there is a transitional put window which has inserted all its normal nodes, but not yet the floating nodes. Then the get window might dequeue all those normal nodes and try to shift up as it has reached the top of the window, but as the floating nodes are not yet inserted it will not notice the increase in width and thus not change its width. If the put window then shifts up, inserts some new nodes in the new width and then stops inserting nodes. Regardless of how many gets are performed, it will never dequeue the nodes in the new width as it will never shift up and notice the gap nodes and the new width. By not letting the get window shift to a higher ***max*** than the put window we eliminate these inconsistency problems.

4.2 Elastic 2Dc stack

The elastic stack is similar in essence to the elastic queue but has a few significant differences. Just like for the queue we only allow the elastic changes to be noticed during window shifts and then encode the required information into the windows, like the depth and width in this case. This way each window always has a unique shape, which makes it easier to reason about correctness.

To make the width elastic we use the same ideas as in the queue where we freeze sub-stacks

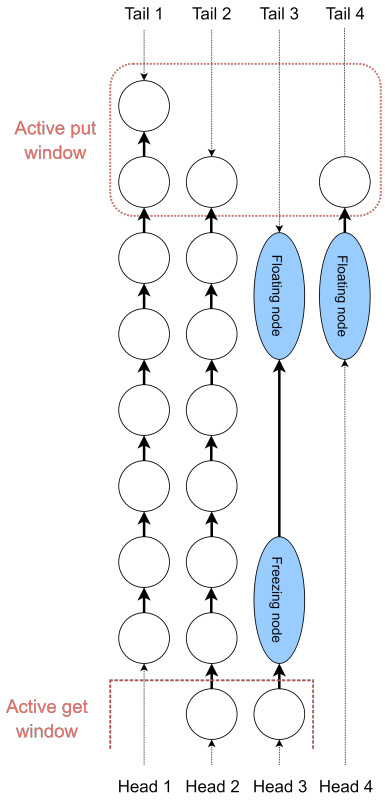


Figure 4.3: A full visualization of how the window can both shrink and expand using gap nodes and how the gaps are represented in the queue.

when we want to decrease width and create floating sub-stacks when we want to increase width. The ends of the queue can be thought to constantly move in one direction, as things are added and removed at different ends. But the stack adds and removed things at the same end, which makes it move back and forth instead. These back and forth movements mean it can enter and leave one different stack width several times when the queue only will enter and move through it once for each change.

4.2.1 Implementation

The implementation of the elastic stack diverges from the queue and does not use the same idea with gap nodes. Instead, it introduces the concept of a *lateral stack* which exists to the side of the 2D stack and tries to push a *lateral node* every time the stack changes width. The lateral stack is like the sub-queues represented as a Treiber stack where the count of the descriptor corresponds to rows in the real stack.

Algorithm 8 shows the structs used for the elastic stack. One can see that the lateral nodes and descriptors are almost identical to the normal ones but instead of encoding inserted items, the lateral nodes encode widths. The window also has a bit more information than for the queue and for example has to encode ***depth*** as it is required continuously for the stack to check if a pop is valid, and not just at shifts. There are some other new fields which will be explained further down. Furthermore, the rest of the pseudo-code can be seen in Algorithms

9, 10, 11.

Algorithm 8: 2D Elastic stack structs

```
Struct Des
    Node* node
    Int count
    Int version

Struct Node
    Node* next
    Item content
    Int next_count

Struct LatDes
    LatNode* node
    Int count
    Int version

Struct LatNode
    LatNode* next
    Int width
    Int next_count

Struct Window
    Int version
    Int max
    Int depth
    Int width
    Int old_width
    Int active_width
    Int pos_last_bottom
    Int upper_bound
```

Algorithm 9: Stack entry operations

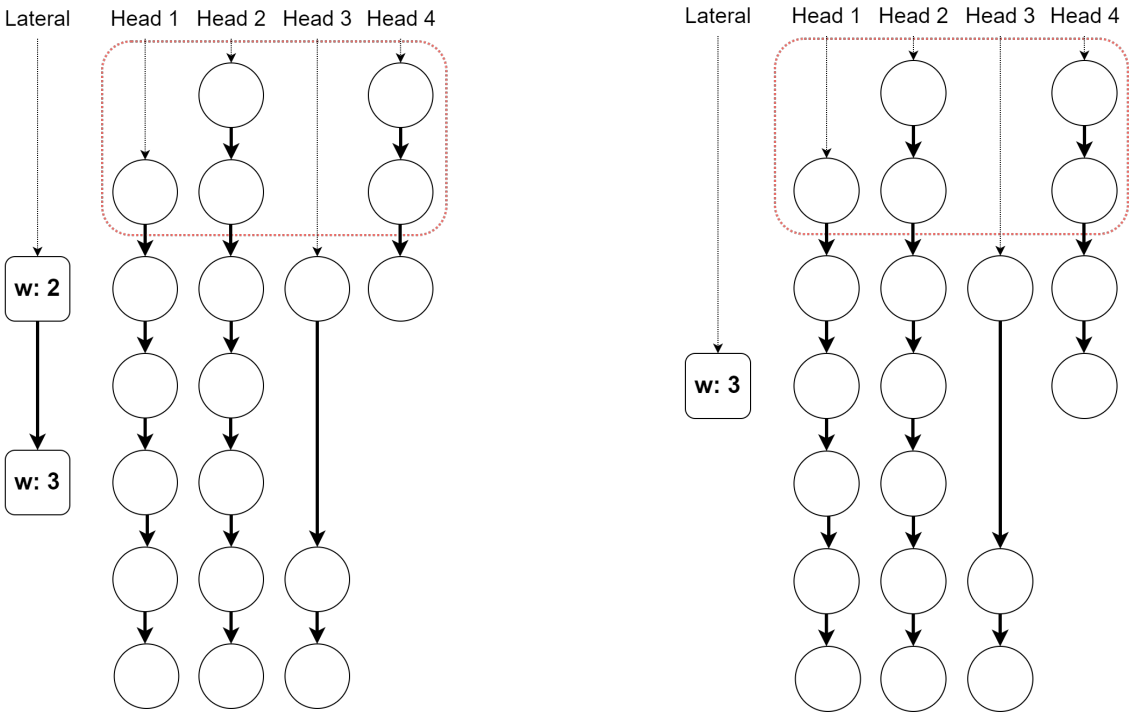
```
Function Push(new_item)
    node ← Node(new_item);
    do
        {Des, index} = WindowSearch(index,
          cont, get);
        node.next ← Des.node;
        node.next_count ← Des.count;
        NDes.node ← node;
        NDes.version ← Des.version + 1;
        NDes.count = max(LWin.max -
          LWin.depth, Des.count + 1);
        cont ← !CAS(Des, NDes);
    while cont;

Function Pop()
    do
        {Des, index} ←
          WindowSearch(index, cont, put);
        if Des.node ≠ NULL then
            NDes.node ← Des.node.next;
            NDes.version ← Des.version + 1;
            NDes.count ←
              Des.node.next_count;
            cont ← !CAS(Des, NDes);
        else
            return 0;
        end
    while cont;
    return Des.node.item;
```

Algorithm 10: Elastic stack window search algorithm

```
Function WindowSearch(op, index, cont)
    if contention then
        index ← RandomIndex(LWin.width);
    end
    if op = put then
        return PutWindow(index);
    else
        return GetWindow(index);
    end
Function GetWindow(index)
    Random ← 0;
    SYNC_WIN(get);
    while True do
        Des ← desArray[index];
        putcount ← desArray[index].putcount;
        if LWin ≠ Win then
            SYNC_WIN(get);
        else if Des.count > LWin.max - LWin.depth
          then
            return {Des, index};
        else if hops < LWin.width then
            if Des.count > 0 then
                empty ← 0;
            end
            HOP(LWin.width, index);
        else if empty then
            return {Des, index};
        else
            Shift(get);
            SYNC_WIN(get);
        end
    end
Macro HOP(width, index)
    if Random < 2 then
        index ← RandomIndex(width);
        Random += 1;
    else
        hops += 1; index += 1;
        if index ≥ width then
            index ← 0;
        end
    end
Function PutWindow(index)
    Random ← 0;
    SYNC_WIN(put);
    while True do
        Des ← desArray[index];
        if LWin ≠ Win then
            SYNC_WIN(put);
        else if Des.count < LWin.max then
            return {Des, index};
        else if hops < LWin.active_width then
            HOP(LWin.width, index);
        else
            Shift(put);
            SYNC_WIN(put);
        end
    end

Function Shift(op)
    SyncLateral(LWin.active_width,
      Lwin.old_width);
    NWin.active_width ← width_global;
    NWin.old_width ← LWin.active_width;
    NWin.version ← LWin.version + 1;
    if NWin.active_width = NWin.old_width then
        NWin.depth ← depth_global;
        NWin.max ← ShiftMax(op);
    else
        NWin.depth ← LWin.depth;
        NWin.max ← LWin.max;
    end
    if NWin.active_width = NWin.old_width ∧ op =
      get then
        NWin.pos_last_bottom ← LWin.max -
          LWin.depth;
    else
        NWin.pos_last_bottom ← ∞ ;
    end
    NWin.width ← ShiftWidth(NWin.max -
      NWin.depth + 1, NWin.active_width);
    CAS(Win, LWin, NWin);
```



(a) Ideal case

(b) Messier case

Figure 4.4: Two different elastic stack states after changing width from three to two to four. The messy case shows that the bottom of the gaps does not completely have to line up and that the lateral stack does not have to exactly describe the widths.

The key invariant of each lateral node is that starting below the row it is placed at and continuing until where the next lateral node is placed, all rows in the normal stack have width smaller or equal to the lateral node's ***width***. Furthermore, all rows at or above the lateral descriptor should have width smaller or equal to ***active_width***. To know at which row the next lateral node is placed each one of them has a field ***next_count*** and the topmost node's row is determined by the descriptors ***count***. This information can then be used when shifting the window to see if it must adapt to the width below it. To shift correctly the lateral stack must therefore be in a consistent state before each such window shift and that is ensured by the function *SyncLateral* in Algorithm 11, which all threads have to perform before trying to shift the window. Two possible states of the elastic stack are shown in Figure 4.4.

Since the width is elastic we will have gaps in the sub-stacks when we increase the width and create floating ones. But as we don't insert gap nodes into the sub-stacks we need another way of representing where the gaps are. This is done by adding the field ***next_count*** to all nodes which save which count the previous node was placed at. This means that after a pop the descriptor is not decreased by one but instead updated to the ***next_count*** stored in the node. When pushing nodes it is hard to know if we are creating a gap or not, so we add the constraint that a node can only be inserted at or above ***Window.max − Window.depth***. This automatically creates the gaps from changing the width, but can also create small gaps when a sub-stack happens to be below the bottom of a window.

The main difference from the original stack is that the elastic one has to synchronize

Algorithm 11: 2D elastic stack helper functions

```
Function SyncLateral()
    LatDes ← Lateral.Des;
    if LWin ≠ Win then
        return;
    end
    NDes ← LowerLat(LatDes.node,
      LatDes.count);
    CAS(LatDes, NDes);
    if LWin.active_width ≠ LWin.old_width
      then
        PushPhase(LatDes);
    end
Function LowerLat(node, count)
    lower ← LWin.max - LWin.depth + 1;
    upper ← LWin.pos_last_bottom + 1;
    if count ≤ lower then
        return node, count;
    end
    if node.width ≤ LWin.active_width then
        new_count ← min(node.count,
          lower);
    else if node.width > LWin.active_width
      then
        new_count ← min(node.count,
          upper);
    next, next_count = LowerLat(node.next,
      node.next_count);
    if node.next_count ≠ next_count ∨
      node.next ≠ next then
        node ← copy(node);
        node.next_count ← next_count;
        node.next ← next;
    end
    return node, new_count;
Function PushLat(LatDes, count, width)
    node ← LatNode(LatDes.node,
      LatDes.count, width);
    NDes.node ← node;
    NDes.count ← count;
    NDes.version ← LatDes.version + 1;
    CAS(LatDes, NDes);
Function PushPhase(LatDes)
    lower ← LWin.max - LWin.depth + 1;
    upper ← LWin.upper_bound + 1;
    if LWin.active_width > LWin.old_width ∧
      LatDes.count ≤ lower then
        PushLat(LatDes, lower, old_width);
    else if LWin.active_width <
      LWin.old_width ∧ LatDes.count ≤ upper
      then
        PushLat(LatDes, upper, old_width);
Function ShiftMax(max, new_depth, old_depth,
  op)
    shift ← (new_depth+1)/2;
    if op = put then
        new_max ← max + shift;
    else
        new_max ← max - old_depth - shift
          + new_depth ;
    end
    return max(new_depth, new_max);
Function ShiftWidth(bottom, new_width)
    widths ← [new_width,
      LWin.active_width];
    lat_node.next_count ←
      Lateral.Des.count;
    lat_node.next ← Lateral.Des.node;
    while lat_node.next_count > bottom do
        lat_node = lat_node.next;
        widths.append(lat_node.width);
    end
    return max(widths);
Macro SYNC_WIN(op)
    LWin ← Win;
    hops ← 0;
    empty ← 1;
    if op = get ∧ index > LWin.width ∨ op =
      put ∧ index > LWin.active_width then
        index ← 0;
    end
```

and utilize the lateral stack when shifting. We start with describing how it sets all window variables when shifting before moving on to how it synchronizes the stack:

- ***active_width*** is the width of not frozen sub-stacks and is read from the global ***width***.
- ***depth*** is read from the global ***depth***.
- ***max*** is found by shifting the top or bottom limit of the window by ***shift*** := ***depth***/2 depending on if the window shifts up or down respectively, but it is always at least ***depth***.
- ***width*** is the width of the window, including the frozen sub-stacks, and is found by taking the max of ***active_width***, ***old_width*** and the ***width*** of all lateral nodes with a higher count than the bottom of the window.
- ***old_width*** is the old windows ***active_width*** and is used to see if the window has just changed width.
- ***version*** is the ABA count and is increased by one for each window.
- ***pot_old_bottom*** is set to the bottom of the window if it is shifting down or ∞ if shifting up and is used for lowering lateral nodes.
- ***upper_bound*** is the largest row at which a node can exist and is used for inserting lateral nodes. It is set to ***max*** if shifting down and the maximum of the new ***max*** and old ***upper_bound***.

To keep the lateral stack in a consistent state there must be a unique set of operations to perform on it each window which all threads agree on and help perform. We call this *synchronizing the lateral* and is done in *SyncLateral* in Algorithm 11. It is just like how the elastic queue had to insert several gap nodes before trying to shift. We divide this into two *phases* which are to first lower lateral nodes and secondly push a new lateral node. Both these phases are explained in more detail below but what they do is uniquely determined by the variables in the window. Each is completed with a CAS if needed and they are forced to happen in the correct order. In both phases, nothing may be changed, and then there is no CAS.

The first phase is to lower lateral nodes which is done to maintain all their invariants and to get as tight bounds as possible (we could just always have ***width*** = ∞ but that would be quite slow). If a node is lowered beneath its ***next_count*** it should instead be popped as its domain would be empty. By calculating the count each node would be lowered to, we can see if they can be lowered or stay where they are. The new count depends on its ***width*** as described below.

- ***width*** < ***active_width***: Node is lowered to the window's bottom (***max*** − ***depth*** + 1) as nodes can be inserted to violate their invariants.
- ***width*** > ***active_width***: Node can only be lowered if the window last shifted down and then to the bottom of that last window (***pos_last_bottom***), as the nodes must have been popped from those sub-stacks and cannot be inserted outside ***active_width***.

- ***width* = *active_width***: Just like first case the node is lowered to the bottom of the window (***max* − *depth* + 1**). Are several possibilities for what to do here, but they have minor consequences.

These rules should now be applied to all lateral nodes, but we can save some work by only looking at lateral nodes with counts higher than the bottom of the window, as we don't lower past that. The only special case is if the topmost lateral node has ***width* = *old_width*** and is at row ***upper_bound*** in which case another thread must have completed the push phase already, so we abort. As the algorithm should be lock-free each node down to the last one needing to be mutated is cloned, and then the top of the stack is updated with a CAS. This way we can modify several nodes using only one CAS which is nice as we not only wanted to modify the topmost node.

Finally, after the nodes potentially have been lowered there is a possibility to push a new lateral node if ***old_width* ≠ *active_width***. Only one such node should be pushed which is achieved by not pushing one if the count at which to push it is not higher than the top of the lateral stack. Its count is again determined by its ***width***.

- ***old_width* < *active_width***: The count becomes the bottom of the current window (***max* − *depth* + 1**).
- ***old_width* > *active_width***: The count becomes the upper bounds where nodes can currently exist (***upper_bound***).

Chapter 5
Analysis

In this chapter, we will look at the correctness of the new designs and provide bounds for their relaxation rank errors. By correctness we mean that no element should be returned twice, no element which has not been inserted should be returned and if doing infinite remove operations while bounding the insert operations, we should eventually reach an empty data structure. This can almost be formulated as a pool like in [6], but since the emptiness check is not linearizable the last clause has to be less restrictive than for a pool.

5.1 Elastic queue

Theorem 1. *The elastic 2D queue is correct and its successful enqueue and dequeue operations are linearizable.*

Proof. When it comes to linearizability each successful operation linearizes with a successful CAS. For the enqueue the linearization point is at the update of a tail's ***next*** pointer and for the dequeue it is updating a descriptor. This also means that no normal node can be returned twice as it is still only represented as a single node in the queue.

The only way to return an element is to dequeue it from a sub-queue using CAS. To return an element not inserted we would have to dequeue a node which was never inserted, and the only such nodes are the gap nodes, which we could mistake for a normal node and return some nonsense value. After dequeuing a node we check that it is not a freezing node, in which case we retry, so it can't happen for a freezing node. To return a descriptor from the window search we check that the descriptors node is not a freezing node and since floating nodes only ever occur after freezing nodes it can't happen for them either. Thus we can never return an element not inserted.

The only possibility of not being able to return an inserted element would be to not properly insert it or that it is inserted into a sub-queue which never becomes active. It is always inserted on a valid sub-queue within the active width using CAS, so that is not a

problem. If a node is enqueued in a sub-queue, that sub-queue must have been activated by the put window either at the initialization of the queue or by increasing width and inserting floating nodes into all newly active sub-queues before shifting and adding any more nodes. For the get window to not include such a node it must not have noticed the increase in width during the transitional put window. Since the get window cannot shift above the global put window it can't shift to a count where the node could have been inserted without seeing the floating nodes during the shift and increase its width accordingly. Thus no element can ever be lost.

□

Theorem 2. *The maximal rank error when dequeuing an element x from the elastic 2D queue is* $(\boldsymbol{width}_{put} - 1)\boldsymbol{depth}_{put} + (\boldsymbol{width}_{get} - 1)(\boldsymbol{depth}_{get} - 1)$ *where the subscripts indicate which of x's windows the variable comes from.*

Proof. Let us inspect node x and try to maximise its rank error, which is the number of nodes enqueued before, but not dequeued before it. A way to visualize this problem is to color all nodes in a queue with up to two colors. All nodes which can be enqueued before x are colored green and all nodes which can be dequeued after x are colored red. The maximum rank error is then the maximum number of nodes which are both red and green. We can use Figure 5.1 to visualize this coloring and we will step by step explain the coloring here.

The coloring comes from three different invariants. First, we argue why all nodes of one color are not also of the opposite color: (i) All nodes enqueued in a later put window (dequeued in an earlier get window) than x must then also be enqueued later (dequeued earlier) than x due to the operations being ordered by the windows, thus they can't be green (red). Conversely, now we argue why the colored nodes have their colors: (ii) All nodes in the same sub-stack as x are correctly ordered by the sub-stack and (iii) Nodes in other sub-stacks in the same or earlier put windows (later get windows) can, or if different windows must be enqueued (dequeued) before x, meaning they are green (red).

These three invariants enforce a unique coloring of all of the nodes in the queue, which again can be seen in Figure 5.1. We see that these bi-colored nodes are the ones in the same or earlier put window as x while simultaneously being the same or later get window than x. To maximize their number x should be on the bottom of its put window and the top of its get window. It can easily be seen by trying to move any of the windows while keeping x in them and seeing that the number of bi-colored nodes shrinks. Then their number becomes the size of both the windows, minus their overlap of one row (which has the width of the put window as x is inserted there) which gives the bound.

□

Now that we have the theorem for an elastic 2D queue concerning both width and depth we can examine the special cases where only one of the dimensions is elastic. Note also that the reason the get and put windows can span different sets of rows is because of the elastic depth which can send them out of sync.

Corollary 1. *The maximal rank error when dequeuing an element x from the elastic depth 2D queue is* $(\boldsymbol{depth}_{put} + \boldsymbol{depth}_{get} - 1)(\boldsymbol{width} - 1)$ *where* $\boldsymbol{depth}$ *is the depth of the window where x was enqueued or dequeues.*

Proof. The proof is the same as in the case of the version where the width is also elastic, but now the $\boldsymbol{width}$ of the get and put windows must be the same.

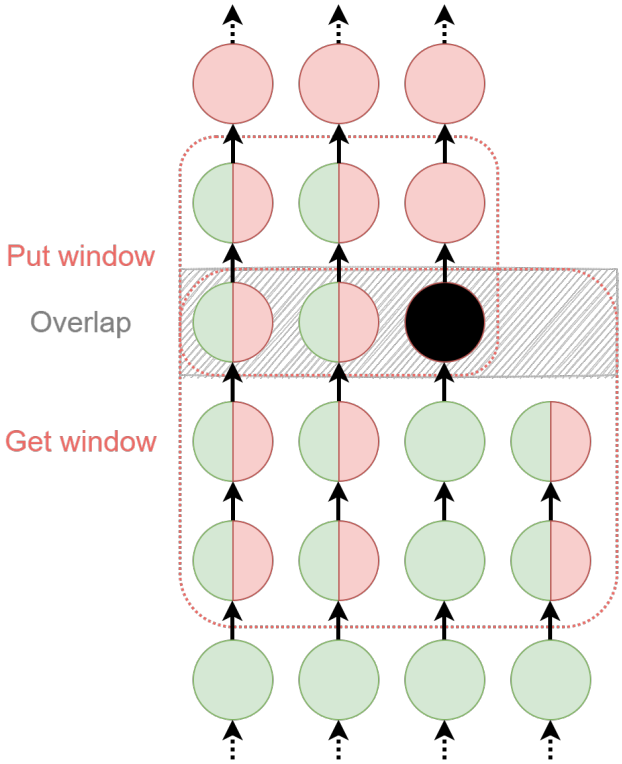


Figure 5.1: The core idea of the proof for the fully elastic 2D queue. The black node is the one inspected, the red nodes can be dequeued after and the green nodes can be enqueued before. The windows are the worst-case scenarios of where they could be placed to cause the maximum possible rank error.

□

Corollary 2. *The maximal rank error when dequeuing an element x from the elastic width 2D queue is $(\mathbf{width}_{put} - 1)\mathbf{depth}$ where $\mathbf{width}_{put}$ is the active width of the window where x was inserted.*

Proof. This follows from the earlier proof of the fully elastic 2D queue, but now the get and put windows must span the same sequence of max values $\{i \cdot \mathbf{depth}\}_i$ since the depth can't change. Thus the window where the node is inserted and removed will have the same width and span the same rows and it becomes just like the normal bound except that the width can vary in different parts of the queue.

□

An important part of these structures is that they are lock-free. Here we show that this is also the case for this elastic version as well.

Theorem 3. *The elastic 2D queue is lock-free.*

Proof. To show that it is lock-free we must show that there will always be progress if some thread tries to make progress for a finite number of steps. The proof of this is that there is always a finite number of predefined operations to be done for each window. A certain number of enqueues/dequeues must be done and in the case of an elastic change there must

also be a certain number of gap nodes which are inserted or deleted before the window again can shift up. Once these actions are done the window can shift, guaranteeing more progress. Each of these actions linearizes with a CAS and can not be undone during the window as the descriptor counts always increase with each CAS. Thus there is a finite number of actions during a window to do, each guaranteeing progress and they all linearize with CAS without having to wait for anything else to complete which could block it. □

5.2 Elastic stack

The correctness on the lateral stack builds on each lateral node encoding an invariant as stated in Lemma 1. We will start with proving that these invariants hold, which will in turn make it easier to prove correctness and bound the rank errors.

Lemma 1. *(lateral node invariant) At the linearization of each window shift it holds that for all lateral nodes* ***lat*** *and normal nodes* ***node*** *it holds that*

$$\textbf{\textit{lat.next\_count}} \leq \textbf{\textit{row}}_{\textbf{\textit{node}}} < \textbf{\textit{row}}_{\textbf{\textit{lat}}} \implies \textbf{\textit{index}}_{\textbf{\textit{node}}} \leq \textbf{\textit{lat.width}} \tag{5.1}$$

(lateral descriptor invariant) Also considering the lateral descriptor ***des*** *it holds that*

$$\textbf{\textit{des.count}} \leq \textbf{\textit{row}}_{\textbf{\textit{node}}} \implies \textbf{\textit{index}}_{\textbf{\textit{node}}} \leq \textbf{\textit{window.active\_width}} \tag{5.2}$$

where $\textbf{\textit{index}}_{\textbf{\textit{node}}}$ *signifies the index of the sub-stack where* ***node*** *resides and* $\textbf{\textit{row}}_{\textbf{\textit{node}}}$ *signifies its row.*

Proof. This proof will be done by induction over the sequence of windows, as the lemma is only guaranteed at the shift instant of each window. In all windows before the first elastic width change, this is trivial as there are no lateral nodes and all normal nodes will be inserted inside ***active_width***.

If we assume we are at the linearization point of an arbitrary window shift, and the lemma held last window shift, we shall prove that it will also hold now. We will only push or pop nodes above the bottom of the window, and we will not modify lateral nodes at or below that row, so their invariants will stay true from the last window. So we will only look at lateral nodes which are above the bottom of the window and we call those the *active lateral nodes*. There can be many such active lateral nodes, especially if the depth of the window has just increased. First, we examine what effect the lowering phase has (where we try to lower lateral nodes), and then the push phase (where we potentially push a lateral node).

It is now possible to push normal nodes inside the current window which means that all active lateral nodes with smaller ***width*** than the ***active_width*** are initially invalidated. If the lowest active lateral node is one of those it will be lowered to the bottom of the window, and all others will be popped. If one was successfully lowered its invariant will still hold as we know it held before this window until the next lateral node, and now it must still hold on the subset of rows beneath the window as they can't have any new nodes inserted. Lowering these nodes will not destroy the invariants of nodes above with higher ***width***. This is because the rows which were previously affected by the lowered nodes were smaller than ***active_width*** and all laterals above the bottom of the window will after the lowering phase have ***width*** > ***active_width***.

Now we instead inspect how active lateral nodes with ***width*** > ***active_width*** are lowered. Some of them will stay where they are and their invariants will not be violated as we cannot

push nodes outside ***active_width***. But if the last window shifted down we try to lower the to the bottom of the last window. Here we inspect what the possible results of these lowerings have on the invariants depending on the node's width ***lat_width*** compared to the old window's active width ***old_width***.

- If ***lat_width*** $\leq$ ***old_width*** then the lateral node must in the last window have been lowered to that window's bottom. Thus we will not succeed in lowering it and its invariant is unaffected.

- If ***old_width*** $<$ ***lat_width*** then for the last window to successfully shift down it must have observed all sub-stacks within its ***width*** to be at the bottom of the window. If the lateral node was above the bottom of the old window, the old window's ***width*** must have been at least ***lat_width***. Since the old and new windows do not allow pushes outside ***active_width*** those stacks must still be at or below the bottom of the old window. Therefore it is safe to lower the lateral node to there and we know the width above that point must be $\leq$ ***max***(***old_width***, ***active_width***). However, if ***old_width*** $>$ ***active_width*** then the descriptors invariant must *not* hold after this lowering.

Now we need only to inspect the active lateral nodes with ***width*** $=$ ***active_width***. If above the bottom of the window they are tried to be lowered there and if possible the invariant will not be violated. If it is lowered or popped the node above it (or descriptor of topmost) must have ***width*** $\geq$ ***active_width*** and thus its invariant will not be violated either.

That were all cases of how lateral nodes are lowered and what invariants then can become violated. But there can also be problems not to do with nodes being lowered. If ***active_width*** $\geq$ ***old_width*** then since the descriptors invariant held last window it will also do it this one as we now only have a more generous invariant. If instead ***active_width*** $<$ ***old_width*** then there is a problem before the new lateral node is pushed as the topmost rows are bounded by ***old_width*** and not ***active_width***.

So after the lowering phase, there were only two cases which could lead to violated invariants and in both cases it is because ***active_width*** $<$ ***old_width*** and thus the descriptors invariant breaking. If ***active_width*** $<$ ***old_width*** then the push phase will push a lateral node at ***upper_bound*** width ***width*** $=$ ***old_width***. This makes it so that no nodes are possible to reside above the inserted node and thus the descriptor's invariant is not violated as there are no nodes to apply it to. The pushed node will also cover the same region as the descriptor did last, which we knew was smaller than ***old_width***.

The last paragraph depended on ***upper_bound*** being the highest row we possibly could have nodes at, and here we quickly show that that assumption holds. When shifting down we set ***upper_bound*** to the maximum of the old and new window's ***max***. This is because if shifting down all nodes must be below or at the top of the window, and in the next window they can only be inserted higher if it has a higher top. If it shifts up we set it to the maximum of the previous ***upper_bound*** and the new ***max***. This is because the new ***max*** either exceeds the old bound or not. So assuming the lateral stack was consistent at the last shift, meaning the width of the window must not miss any nodes above its bottom, there cannot be any node above ***upper_bound***.

Finally we examine the other case where ***old_width*** $<$ ***active_width***. Here a node is only pushed for efficiency and not correctness, but we must make sure its invariant holds. If we can push the node at the bottom of the window it must be the topmost node. All rows below it until the next node would last window have been less than ***old_width*** since there cannot have

been any node above this at the last shift. Since the descriptor invariant held last window this node's invariant must also hold.

So the invariants must hold this next window shift and induction gives they must always hold.

□

Corollary 3. *During a window, there cannot be nodes above the row* ***upper_bound****.*

Proof. This was proved in the third to last paragraph of Lemma 1. □

Theorem 4. *The elastic 2Dc stack is correct and its successful push and pop operations are linearizable.*

Proof. Each successful push and pop linearizes in the same way as the method would on a normal Treiber stack[12]. This means that no element can be returned twice and we only remove nodes which have earlier been inserted.

The only possibility of not being able to return an inserted element would be for its node to be at a larger index than the window's ***width*** and that all sub-stacks within ***width*** are empty at some point during the window. But due to Lemma 1 and how the window determines its ***width*** in Function *ShiftWidth* (Algorithm 11), there cannot be a row inside the window with a node outside ***width***. These two sentences contradict each other, so it is impossible for the first to be true, meaning the stack must be correct.

□

Theorem 5. *The maximal rank error when popping an element* x *from the elastic 2Dc stack is* $(2\boldsymbol{depth} + 2\boldsymbol{shift} - 1)(\boldsymbol{width} - 1)$*, where* ***depth****,* ***shift*** *and* ***width*** *are the maximum of those window variables between the push and pop of* x*.*

Proof. To maximize the number of nodes pushed after x but not popped before x we need to find two bounds. The first one is the lowest bottom (x_{lower}) of a window $\boldsymbol{max} - \boldsymbol{depth}$ which could have taken place after the push of x, as nodes pushed after x must have been pushed above that. The second is the highest possible row nodes can reside when x is popped (x_{upper}).

To find x_{lower} we consider the instant of pushing element y, which happens while x is somewhere on the stack. This means that the current ***upper_bound*** due to Corollary 3 has to be larger or equal to the row of x. The upper bound is calculated at shifts as its old value, or the ***max*** of the window if shifting down. Thus the lowest row y can be pushed on is $\boldsymbol{row}_x - \boldsymbol{depth} - \boldsymbol{shift} + 1 = x_{lower} + 1$ where ***depth*** and ***shift*** must have been in some window since the push of x. We could make more detailed and strict assumptions about ***depth*** and ***shift*** but we choose not to make the theorem clearer. For example, ***depth*** must only really be included if it is the ***depth*** from the window x was pushed.

Finding x_{upper} is simpler as we can use 3 directly. As x can only be popped in a window where x is above its bottom, we have that $x_{upper} = \boldsymbol{row}_x + \boldsymbol{depth} + \boldsymbol{shift} - 1$.

Together we have $x_{upper} - x_{lower} = 2(\boldsymbol{depth} + \boldsymbol{shift}) - 1$ which that must then be the maximal number of elements pushed after x and not popped before it on a single sub-stack. Multiplying with $\boldsymbol{width} - 1$ gives the bound, which is done as the sub-stack x is pushed on must be ordered.

□

We will not give a separate result for the stack where only the depth is elastic, as we don't have a tighter bound there than Theorem 5.

5.2.1 Revisiting 2Dc stack

For the the final Corollary where the stack only has elastic width we will essentially use the same bounds as the normal stack. However, the bound in the original paper has a small oversight in its proof and has a too low bound. Therefore we have revisited the analysis (Theorem 5 in the original paper[10]) and here will give a slightly worse, but accurate, rank error bound. Its starts with a few lemma to build up to the final bound in Theorem 6 The new lemmas and theorems take inspiration from the original paper, but build the theory differently to account for the changes and are quite different from the original ones.

These proofs are more consistent in style with the original paper than the rest of this thesis. Here we introduce the differing notations.

- ***Global*** is the global window's ***max***,
- W_i denotes the window with $\boldsymbol{max} = \boldsymbol{shift} \times i$,
- $W_i^{max} = \boldsymbol{shift} \times i$,
- $W_i^{min} = \boldsymbol{shift} \times i - \boldsymbol{depth}$,
- N_j is the number of nodes on sub-stack j.

Lemma 2. *Given that* $\boldsymbol{Global} = \boldsymbol{shift} \times i$*, it is impossible to observe a state (S) such that* $N_j > W_{i+1}^{\max}$ *(or* $N_j < W_{i-1}^{\min}$ *).*

Proof. This is proved in the main article. □

Lemma 3. *If* x *is pushed to sub-stack* j *while observing* W_i*, then*

$$W_i^{min} - \boldsymbol{shift} + 1 \leq N_j \leq W_i^{max} \qquad \forall j \in [0, \boldsymbol{width}) \tag{5.3}$$

where N_j *is the number of items on the sub-stack* j *after the push.*

Proof. By Lemma 2 we get bounds on the sub-stack in the moment of observing W_i. Now we add the constraint that the push must be valid by $N_j \leq W_i^{max}$ which becomes the new upper bound. Add 1 to both bounds from the new push and the interval is found. □

Lemma 4. *If* x *is popped from sub-stack* j *while observing* W_i*, then*

$$W_i^{min} + 1 \leq N_j \leq W_i^{max} + shift \qquad \forall j \in [0, width) \tag{5.4}$$

where N_j *is the number of items on the sub-stack* j *before the pop.*

Proof. Same proof as above, but now use the criteria for valid pop and don't remove 1 as N_j is before the pop. □

Lemma 5. *If* x *is pushed to sub-stack* j *while observing* W_i *and then popped while observing* $W_{i'}$*, the difference between the two windows is in the interval*

$$-2\,\boldsymbol{shift} - \left\lceil \frac{\boldsymbol{depth} - 1}{\boldsymbol{shift}} \right\rceil \boldsymbol{shift} \leq W_{i'}^{max} - W_i^{max} \leq \left\lfloor \frac{\boldsymbol{depth} - 1}{\boldsymbol{shift}} \right\rfloor \boldsymbol{shift} \tag{5.5}$$

Proof. Assume x is inserted at position $W_i^{max} - \theta$ where θ is how far it is from the top of the window. From Lemma 3 we get the bounds $\theta \in [0, \mathit{depth} + \mathit{shift} - 1]$. Using Lemma 4 and that the pop must be valid we get

$$\begin{aligned} & W_{i'}^{min} + 1 \leq W_i^{max} - \theta \leq W_{i'}^{max} + \mathit{shift} \\ \iff \quad & -\mathit{shift} - \theta \leq W_{i'}^{max} - W_i^{max} \leq \mathit{depth} - 1 - \theta. \end{aligned}$$

But as shift is constant the difference of the windows can only be a multiple of shift. Furthermore, the limits of θ are known. This gives us the final bounds

$$\begin{aligned} & -\mathit{shift} - \left\lceil \frac{\theta}{\mathit{shift}} \right\rceil \mathit{shift} \leq W_{i'}^{max} - W_i^{max} \leq \left\lfloor \frac{\mathit{depth} - 1 - \theta}{\mathit{shift}} \right\rfloor \mathit{shift}. \\ \iff & -\mathit{shift} - \left\lceil \frac{\mathit{depth} + \mathit{shift} - 1}{\mathit{shift}} \right\rceil \mathit{shift} \leq W_{i'}^{max} - W_i^{max} \leq \left\lfloor \frac{\mathit{depth} - 1}{\mathit{shift}} \right\rfloor \mathit{shift} \\ \iff & -2\,\mathit{shift} - \left\lceil \frac{\mathit{depth} - 1}{\mathit{shift}} \right\rceil \mathit{shift} \leq W_{i'}^{max} - W_i^{max} \leq \left\lfloor \frac{\mathit{depth} - 1}{\mathit{shift}} \right\rfloor \mathit{shift} \end{aligned}$$

.

□

Theorem 6. *2Dc-Stack is linearizable with respect to k-out-of-order stack semantics, where* $k = \left(2\mathit{shift} + \mathit{depth} + \left\lfloor \frac{\mathit{depth}-1}{\mathit{shift}} \right\rfloor \mathit{shift}\right)(\mathit{width} - 1)$.

Proof. Assume x is pushed to sub-stack j' while observing W_i and later popped while observing $W_{i'}$. Using Lemma 2 we can get bounds for the other sub-stacks at those moments.

$$\begin{aligned} &\text{At push} && \forall j \neq j' : \ N_j \geq W_i^{min} - shift \\ &\text{At pop} && \forall j \neq j' : \ N_j \leq W_{i'}^{max} + shift \end{aligned}$$

As k is the maximal number of items pushed after x as well as popped after x these inequalities are key. All sub-stacks can vary their items in this period between those limits. But the sub-stack j must of course end up with the same items when x is pushed as popped, as x must be at the top for both.

So the largest number of new items pushed after the push of x, still on the stack after the pop of x becomes the sum of the possible difference of all but the j'th sub-stack. This together with Lemma 5 gives

$$\begin{aligned} k &= \mathit{max}\,(\mathit{width} - 1)(W_{i'}^{max} + \mathit{shift} - (W_i^{min} - \mathit{shift})) \\ &= \mathit{max}\,(\mathit{width} - 1)(W_{i'}^{max} - W_i^{max} + \mathit{depth} + 2\mathit{shift}) \\ &= (\mathit{width} - 1)\left(\mathit{depth} + 2\mathit{shift} + \left\lfloor \frac{\mathit{depth} - 1}{\mathit{shift}} \right\rfloor \mathit{shift}\right). \end{aligned}$$

□

5.2.2 Final elastic stack analysis

Corollary 4. *The maximal rank error when dequeuing an element x from the elastic width 2Dc stack is $\left(2\mathit{shift} + \mathit{depth} + \left\lfloor \frac{\mathit{depth}-1}{\mathit{shift}} \right\rfloor \mathit{shift}\right)(\mathit{width}-1)$, where **width** is the maximum width between the push and pop of x.*

Proof. This follows from the earlier Theorems 6 where it is shown for the original stack, and from the proof of Theorem 4 where the elastic stack adapts its width correctly. The elastic width does not affect the original theorems, except by changing the number of possible sub-stacks.

The only other difference in functionality between the elastic width stack and the normal one is that the elastic one does not allow pushes under the bottom of the window. This can not worsen the bound as it only in some cases creates empty spaces in the stack which could have been taken up by nodes to swap order with x. But it is still possible to construct an example with the given bound, so it cannot be lowered.

□

Theorem 7. *The elastic 2Dc stack is lock-free.*

Proof. The stack searches for valid indexes the same way as the original and does not have any blocking capabilities. The lateral synchronization has a maximum of two CAS to complete. When one thread has completed the lowering no other node will find any nodes to lower, and cannot try to lower anything. The push of a new node is also just a potential CAS. A thread cannot destroy the push of a new node by lowering it as if it is narrower than ***active_width*** it would try to lower it to the same row and if it is wider the special case will prevent the thread from proceeding with the lowering. The only other new thing is the calculations for the window shifts but they are just a bounded number of computations and then a CAS. □

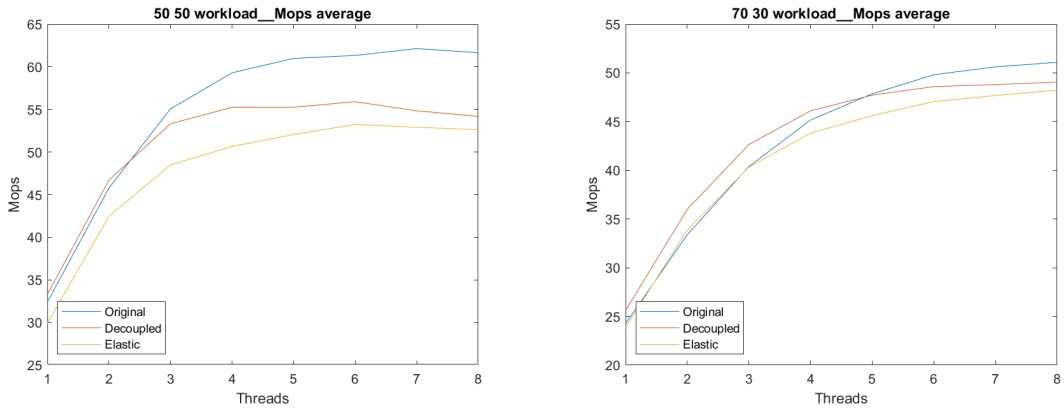
Chapter 6
Evaluation

Using experiments we evaluate the performance of our new elastic data structures and compare them to the original versions. As the original paper compares its data structures with other state of the art implementations we only focus on comparing with the original 2D implementations. The code is written in C and builds on the code from the original paper [10], but with some modifications. The elastic code and the original are kept as similar as possible, without incorporating any elasticity into the original versions to guarantee fairness. The original code in turn builds on the ASCYLIB library as well as its SSMEM memory management framework [3].

During the experiments, we will mostly measure throughput (operations per second) and relaxation rank errors. We want to capture how much the throughput is slowed down after an elastic change, during an elastic change and if there are no elastic changes at all. We further want to see how this depends on the number of threads and the amount of relaxation. For rank errors we want to make sure it stays inside the analytical bounds and that it hopefully should not be much worse than the original. Additionally, we want to see how the rank error changes over time after an elastic change.

As we care about the case of high contention we mostly do tests where all threads are just inserting and deleting elements without any work in between and unless otherwise specified the insertion and deletions have equal probabilities. The data structures also start with 2^{14} nodes to avoid empty returns as much as possible in the benchmarks, which is a standard practice[10, 13]. Each experiment is run for five seconds and the result is obtained as the average of ten such runs.

To measure rank error we create a shared normal data structure and a lock in parallel to the relaxed one like in[10]. Then all CAS operations to insert or remove an element from the structure are encapsulated in the lock and the element is inserted into both the relaxed and normal structure if successful. Then the rank errors can simply be measured as distances on the normal data structure. This is not a perfect measurement, as the locking to a certain extent changes the dynamics of the data structure, but there does not seem to be consensus in the field for how to measure rank errors.



(a) Even workload

(b) 70% enqueues

Figure 6.1: Plots of how the original, decoupled and elastic queues scales with threads using ***width*** = 16 and ***depth*** = 16.

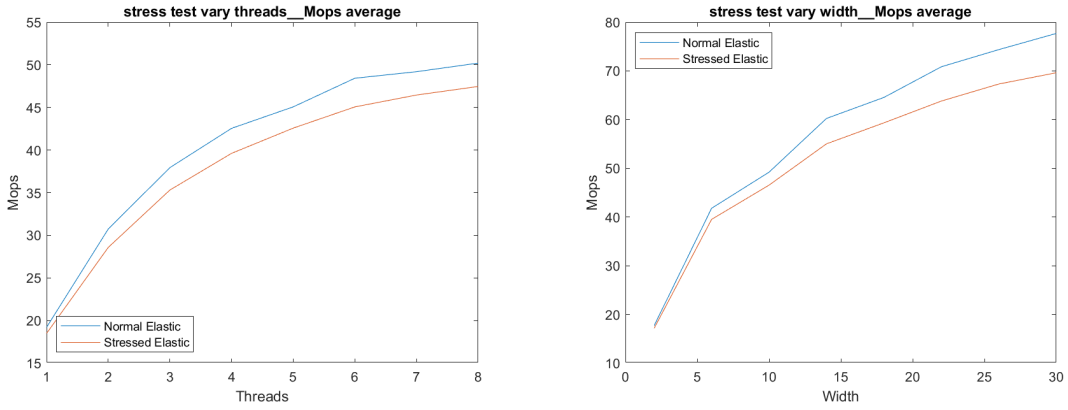
The experiments are run on an x86-64 Intel Xeon E5-2687W v2 machine which has two sockets. Each socket has an eight-core Intel Xeon processor running at 3.4 GHz, where each core can run two threads and with cache sizes L1d: 32 KB, L1i: 32 KB, L2: 256 KB, L3: 25.6 MB. The machine runs on Ubuntu 18.04.6 LTS. During runs, each software thread is pinned to a hardware thread and they run on one socket with one thread per core. As compiler we used gcc 7.5.0 with optimization flag -O3.

6.1 Elastic queue

When benchmarking the performance degradation of the elastic version compared to the original we want as fair of a comparison as possible. One big difference between them however is that the search indexes for the elastic were decoupled. So the decoupled has one index to go back to for the enqueues and one for the dequeues, while the original only has one index for both operations. To make the comparisons fairer we introduce the *decoupled queue* which is exactly like the original queue but also has decoupled search indexes like the elastic queue. In our evaluations, we will compare all three queues to get a better view of what is causing the difference in performance. In theory, the elastic queue should never be able to be faster than the decoupled queue, as it just has some extra work, but it might be faster than the original in settings benefiting the decoupled queue.

In Figure 6.1 we can see how the elastic queue compares to the decoupled and original when it comes to scaling with threads at medium relaxation. Here the elastic queue does not do any elastic changes. Recall that the decoupled queue is when the enqueue and dequeue operations save different indexes of where they succeeded last and try to continue operating on. As expected the elastic is a bit worse than the decoupled and they both are worse than the original. We can also see that both the elastic and decoupled perform relatively better with the skew workload in Figure 6.1b which probably is due to them then being closer to the original then since at 100% skew workload the decoupled and original are identical and they are as far apart as possible at even workload.

To characterize the cost of a width shift in the elastic queue you can constantly shift



(a) Scaling with threads (16 width)

(b) Scaling with width (8 threads)

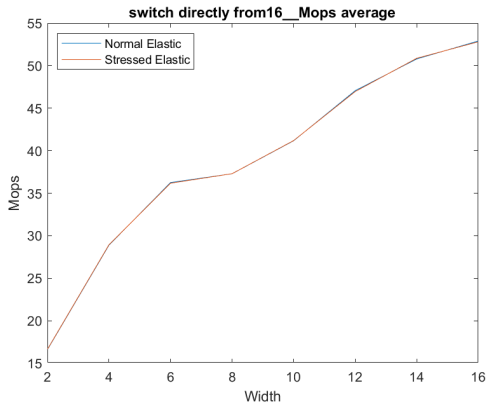
Figure 6.2: Comparison of how the elastic queue performs with no width changes and with the maximal amount possible. The stressed queue changes each window shift between ***width*** and 2***width***, making its active width always ***width***.

between two widths x and y each window shift. Then the active width will always be the smaller width and the indexes in between the widths will constantly be filled with alternating freezing and floating nodes. In Figure 6.2 we can see how doing this between ***width*** and 2 ***width*** affects the throughput of the elastic queue. Overall it is quite a low performance cost and changing width this often is not expected, so the actual cost of changing width seems to be very low.

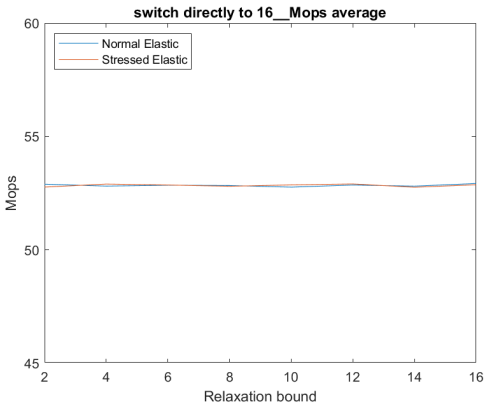
To see the long-lasting consequences of shifting the width of the elastic queue we can insert the initial elements and then change the width before starting to measure the throughput. This will mean that the put window will quickly shift up and notice the new width which it then will run with for the rest of the experiment. Results from this can be seen in Figure 6.3 which essentially confirms the fact that there are no long term performance effects of changing the width. As long as the transition period has passed and the get window has caught up to the new width it will perform just like a normal elastic queue which is not changing width.

One feature of the original queue was that it scaled monotonically with relaxation. In Figure 6.4 we see how the elastic and decoupled queue compares with the original under scaling relaxation. Notably the elastic and decoupled perform quite similarly for the high relaxations, while the original has slightly different behaviour. As expected the throughput increases monotonically for all of them when increasing ***depth*** (Figure 6.4a) and so does the observed rank error (Figure 6.4b).

To see more clearly how the relaxation changes over time after an elastic shift we can look at how it changes for the individual removed elements. When we measure the relaxation we can save all of the rank errors of the removed items and then plot a moving average over these errors in the order they were removed to see the elastic changes effect in action. This can be seen in Figure 6.5 where we for two cases first change ***depth*** and then ***width*** after a third and two thirds of the experiment respectively. It behaves for the most part as expected and has three seemingly stable regions corresponding to the three window shapes. But one can also see a small transition period when initially changing ***depth***. This stems from the fact that we for a period are deleting elements which were inserted in the old ***depth***, since the ***depth***

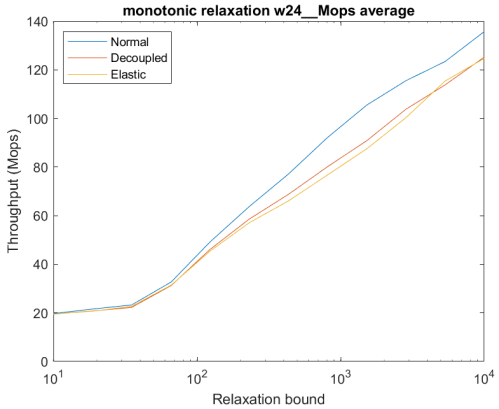


(a) Directly changing from width 16

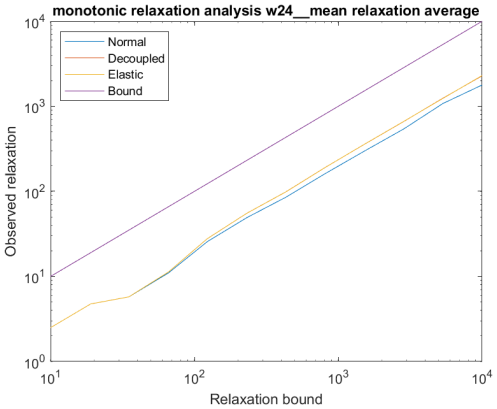


(b) Directly changing to width 16

Figure 6.3: Comparison of how the elastic queue performs when directly elastically changing to a width at the start of the experiment as compared to starting at the width.

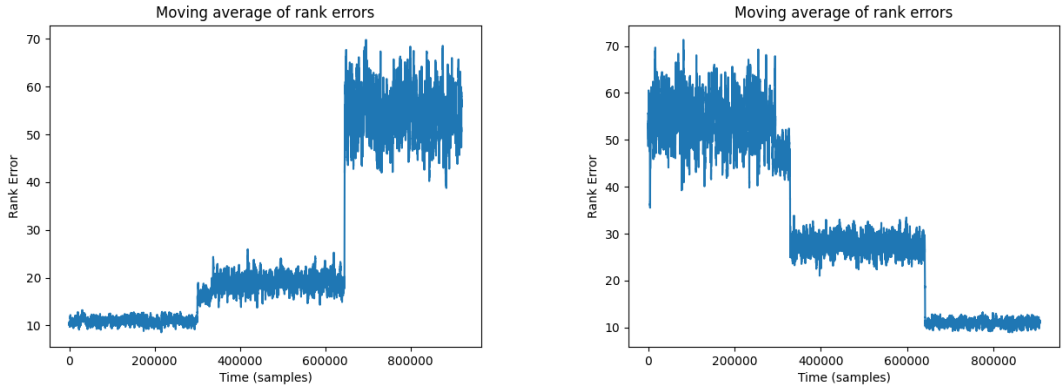


(a) Throughput



(b) Observed relaxation

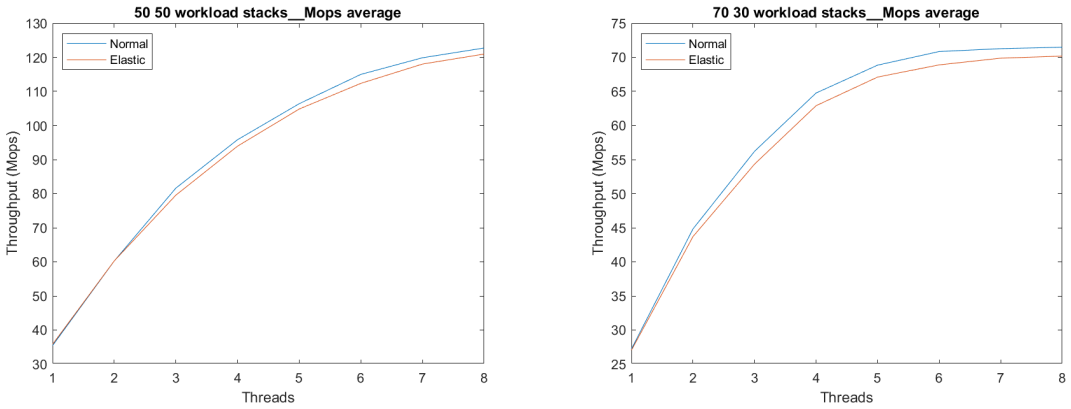
Figure 6.4: Plots of how the queues scale with relaxation. The relaxation is scaled by first increasing the width to 24 and then increasing the depth. The experiments are using eight threads and 50-50% enqueues and dequeues.



(a) Depth and width from 8 to 16

(b) Depth and width from 16 to 8

Figure 6.5: Moving average of rank errors for dequeued nodes in elastic queue. In both cases, we first do a change in depth after 1/3 time and then one in width after 2/3 time.



(a) Even workload

(b) 70% puts

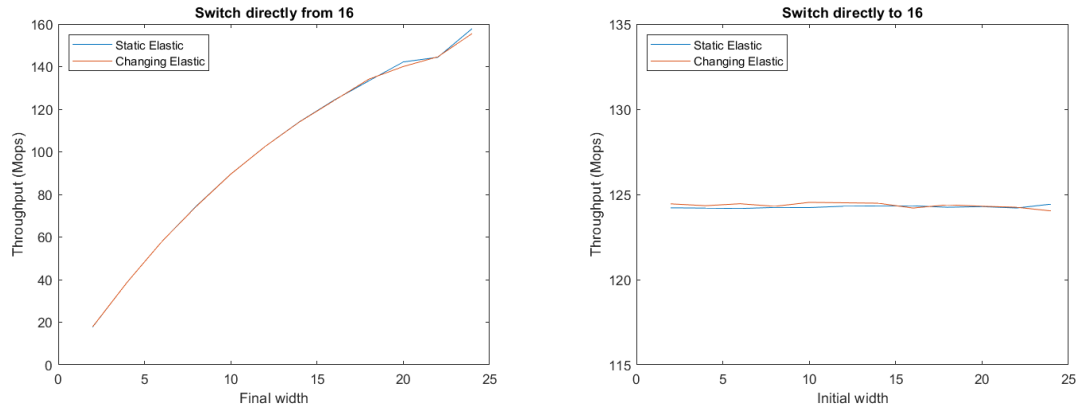
Figure 6.6: Plots of how the original and elastic stacks scale with threads using ***width*** = 16 and ***depth*** = 16.

is updated at both ends of the queue simultaneously. Since the change in ***width*** must travel through the queue no such transition period is seen for the changes in width.

6.2 Elastic stack

For the elastic stack, we benchmark its performance against the normal stack. In Figure 6.6 we see how the elastic compares to the original algorithm without any elastic changes under even and skewed workload (Figure 6.6b). They perform very similarly, suggesting that the extra work from the elasticity is almost negligible when no elastic changes are made.

To measure the lasting costs of a change in width for the stack we change the width directly to x after inserting the initial elements, and then compare it to a stack which started at width x. This is seen in Figure 6.7 where we on the left directly switched from width 16



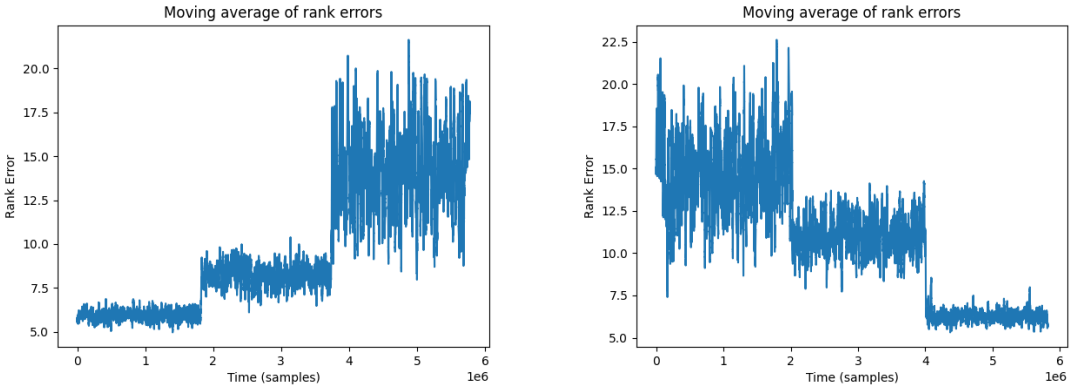
(a) Directly changing from width 16

(b) Directly changing to width 16

Figure 6.7: Comparison of how the elastic stack performs when directly changing elastically to a width at the start of the experiment as compared to starting at the width.

to other widths, and on the right switched from other widths to **16**. In it, we see that there is no large difference at all and that the stack seemingly has no major lasting consequences of changing width.

In Figure 6.8 we present the rank errors over time for a queue with elastic changes. To see more clearly we use a moving average of 10 000 points. In the plots, the depth is first changed and later the width. Comparing the results for the queue in Figure 6.5 to the stack in Figure 6.8, we see that the stack lacks the small middle ground of rank errors after changing the depth. This is because for the stack the depth change does not have to traverse the data structure to reach both windows.



(a) Depth and width from 8 to 16

(b) Depth and width from 16 to 8

Figure 6.8: Moving average of rank errors for dequeued nodes in elastic stack. In both cases, we first do a change in depth after 1/3 time and then one in width after 2/3 time.

Chapter 7
Conclusion

In this part, we discuss some of the problems we faced when designing the algorithms. This can hopefully be useful for people who want to extend this work by providing some context and motivation for the design choices. Furthermore, we will discuss some of the many possible ways to continue this work and lastly some conclusions.

It was immediately obvious that the elastic width was harder to implement, compared to the elastic depth. The simplest solution seemed to be to have some pointers or descriptors in parallel to the structure for how wide each row should be. This is similar to the idea that eventually became the elastic stack. A similar thing could probably have been done for the queue but there is one major issue to do with the direction of information. In a stack, the gaps are always created and removed from the same (topside) direction which means the nodes can have a ***next_count*** field to encode where the gaps are. But for the queue, we dequeue the gap from the other side, and we would need to modify an already insert node when completing the gap if we want to use the same idea. This is not that appealing as it would require the nodes to be mutable, which gap nodes circumvent.

Initially, we also wanted to reuse the gap nodes for the stack. There are however some issues and many of them come from the fact that the stack does not grow monotonically in one direction like the queue. In the queue, the put window has a transitional window where it inserts the gap nodes before the change and thus we know that after that window shifts up the gap nodes will be in place. But for the stack that does not work as easily as we can shift both up and down. That means that we might have to push and pop gap nodes for the same width change several times as we shift up and down. Especially the freezing nodes would have to be popped when popping elements under them, but then add the nodes when shifting further up. However, we believe it is possible to find a nice solution with gap nodes for the elastic stack as well and that they in a way are stronger than the lateral stack as they are harder to miss.

By using our elastic 2D data structures one can adjust the amount of relaxation dynamically and hopefully be able to increase performance. This leads to shorter running times which in itself is useful as it allows things which are dependent on the result to get it faster.

But it also leads to lower energy consumption which is good for the environment.

In this thesis, we have not focused that much on the elastic depth. This comes down to the fact that just implementing elastic depth is easy, especially for the queue. You just have to change one variable to being shared (_Atomic/volatile) and have some way to update it from outside. This has no negative impact on performance. Although, it can lead to some unexpected relaxation behaviours as seen in Corollary 1. According to the original paper[10] the performance gain from increasing the width is not that noticeable after two to three times the number of threads, and instead starts to decrease due to more hops. Changing the depth has no such drawbacks and is suitable for changing back and forth when the extra throughput is needed. It is possible that the best idea is to use elastic structures with only elastic depth, as the extra code needed for elastic width can cause a bit of performance degradation, like in Figure 6.1a.

The elastic stack seems to be fast, but we feel the algorithm has a few problems. One is that it is too complicated as compared to the core idea. One symptom of this is that the window has too many, and quite convoluted, variables. For example, we might be able to remove the field for the potentially last bottom if we instead lower nodes to the current window max (which should always be larger). The proof for the elastic stack is also long and wordy and it would be nice to make it more concise and better structured. A possibility would also be to use some proof assistant like in [6].

7.1 Future work

The original 2D paper[10] has more data structures than we covered in this thesis and it would be nice to create elastic extensions for all of them. The 2Dd stack on its own is not that exciting on its own as we now have a solution for the 2Dc stack which outperformed the 2Dd stack. The counter could be interesting to make elastic, but it is quite different from the other data structures. Primarily we want an elastic extension to the deque, which in a way is a combination of a stack and queue. We feel that by using the techniques and ideas presented in this thesis it should be possible to create an elastic deque. Probably it would need gap nodes as it, much like the queue, can enqueue and dequeue gaps from different directions.

This thesis has only focused on the design of these new data structures, but it is also of great interest to know how effective they can be in different applications. There have been benchmarks used in earlier papers[5] where relaxed queues are used in graph algorithms, such as minimal spanning trees. Relaxed priority queues have been used for efficiently finding the shortest path in a graph[13]. It would be very interesting to look at use cases for these elastic data structures. In graph algorithms like shortest path the amount of nodes in the data structure can be high at the beginning and then become lo during the last part. Here it would be interesting to look at whether the relaxation could be kept high at the start but lower in the end, and in that way lead to increasing performance.

A more analytical direction to continue in would be to define other semantic relaxation properties for the data structures. There could be a tighter definition than *k-out-of-order* for the queue as it is impossible to return the $k + 1$'th topmost element several times in a row. This is because the relaxation is bounded within the windows and eventually you must return the topmost elements before shifting away. For example, the definition of *lateness*[6] could be incorporated which means that there can pass at maximum k get operations between each

correctly ordered one.

A theoretically simple improvement for this thesis is to optimize the elastic queue as its throughput is lower than we expected. In each enqueue/dequeue it has to check the content of the node to look for gap nodes before proceeding with the CAS, but this should not have to fetch extra memory as we already have to read the node for its next pointer. The shifts do take longer, but like the elastic stack they should take up a small portion of the time and thus not be that detrimental either. There is little use guessing about the reason for the performance cost without thoroughly profiling the program, but we don't see any real reason why our elastic queue idea has to be much slower than the normal queue.

Another direction would be to look further into optimal ways to couple the put and get search indexes. For the elastic queue, we had to decouple the indexes as the respective widths could be different. As its operations operate on different ends of the sub-queues it should not decrease the data locality to decouple the indexes. But in actuality, it has to check the tail when trying to dequeue from the head to check if the sub-queue is empty. But coupling the indexes will also probably increase the number of hops (which is changing which sub-structure to inspect) since a get operation can change the index even if the put operations want to stay. This means it is a trade-off as can be seen when we made the workload skew in Figure 6.1b. For example, the queue can save its ***hop*** count for each operation between function calls if the indexes are decoupled. In the elastic stack, we had a coupled index as both operations shared the same window. But even there the get operations can operate on the frozen sub-stacks which the put ones cannot, which might make it beneficial to decouple the indexes in some way.

This thesis has only tried to extend the 2D framework, but there are other relaxed data structures that could be made elastic. Similar techniques can probably be used for some of them, like some of the DQs[5]. Other data structures like the priority queues of MultiQueues[13] require more work as they are not linear in the same way, and thus pose an interesting problem.

7.2 Final words

This thesis has extended semantically relaxed queues and stacks to be able to vary their relaxation during run time. This was done in two different ways which both are promising when it comes to doing similar extensions in the future. Firstly we introduced the concept of *gap nodes* which are auxiliary nodes inserted into the data structure to easily keep track of elastic changes. Secondly, we introduced the *lateral* stack which was a parallel linear structure to the elastic one which kept track of elastic changes.

The main research question of the thesis was to try to create elastic extensions for the stack and queue, which we succeeded in. Additionally, we introduced four points on which to measure how good an extension would be. The first criteria was that they should be lock-free, which ours are. Secondly we proved the correctness of our extensions and gave strict rank error bounds, even though the proofs could be more concise and formal.

Third, our experimental results show that the elasticity in the queue can add a little overhead and reduce the performance by a few percent. This is acceptable, but it would preferably perform a little bit better. The elastic stack on the seems to be almost as fast as the original and therefore fulfills the criteria.

Finally, our results also showed that the changes in elasticity don't reduce the throughput that much. However, the actual change in relaxation only happens at window shifts and increasing the width for the queue has to wait a bit extra as the floating nodes take up space instead of normal nodes. This was done to make the analysis easier, but it could be improved further so the changes takes effect more quickly.

In conclusion we succeeded with the goal of the thesis. It can be continued by improving the proofs, increasing the performance of the elastic queue and making the changes take effect quicker. An interesting area of future work is to use the introduced techniques to extend other data structures, like the 2D deque[10].

References

[1] Intel 64 and ia-32 architectures kernel software developer's manual volume 2b: Instruction set reference, m-u. `https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-2b-manual.pdf`. Accessed: 2022-06-12.

[2] 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, page 283–292, New York, NY, USA, 2017. Association for Computing Machinery.

[3] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Asynchronized concurrency: The secret to scaling concurrent search data structures. ASPLOS '15, page 631–644, New York, NY, USA, 2015. Association for Computing Machinery.

[4] Mike Dodds, Andreas Haas, and Christoph M. Kirsch. A scalable, correct time-stamped stack. *SIGPLAN Not.*, 50(1):233–246, jan 2015.

[5] 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. CF '13, New York, NY, USA, 2013. Association for Computing Machinery.

[6] Thomas A. Henzinger, Christoph M. Kirsch, Hannes Payer, Ali Sezgin, and Ana Sokolova. Quantitative relaxation of concurrent data structures. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, page 317–328, New York, NY, USA, 2013. Association for Computing Machinery.

[7] Maurice Herlihy, Nir Shavit, Victor Luchangco, and Michael Spear. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2021.

[8] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '96, page 267–275, New York, NY, USA, 1996. Association for Computing Machinery.

[9] Hamza Rihani, Peter Sanders, and Roman Dementiev. Multiqueues: Simple relaxed concurrent priority queues. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '15, page 80–82, New York, NY, USA, 2015. Association for Computing Machinery.

[10] Adones Rukundo, Aras Atalar, and Philippas Tsigas. Monotonically Relaxing Concurrent Data-Structure Semantics for Increasing Performance: An Efficient 2D Design Framework. In Jukka Suomela, editor, *33rd International Symposium on Distributed Computing (DISC 2019)*, volume 146 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 31:1–31:15, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[11] Jonas Skeppstedt and Christian Söderberg. *Writing efficient C code : a thorough introduction*. Skeppberg, 2nd edition, 2020.

[12] R.K. Treiber. Systems programming: Coping with parallelism. Technical report, International Business Machines Incorporated, Thomas J. Watson Research Center, 1986.

[13] Marvin Williams, Peter Sanders, and Roman Dementiev. Engineering multiqueues: Fast relaxed concurrent priority queues. *CoRR*, abs/2107.01350, 2021. arXiv:2107.01350.

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2022-06-03

EXAMENSARBETE Highly Scalable Queues and Stacks with Elastic Relaxation
STUDENT Kåre von Geijer
HANDLEDARE Philippas Tsigas (Chalmers), Jonas Skeppstedt (LTH)
EXAMINATOR Flavius Gruian (LTH)

Effektiva, flexibla och inexakta köer

POPULÄRVETENSKAPLIG SAMMANFATTNING **Kåre von Geijer**

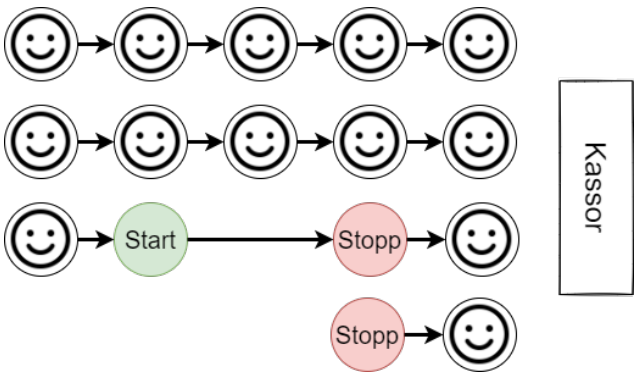
För att undvika flaskhalsar i program vill man ha effektiva och skalbara strukturer för att lagra och ordna data. Detta arbete har byggt vidare på två typer av effektiva men inexakta köer så att de flexibelt kan ändra sin precision vid behov.

Nya datorer blir effektivare varje år men nu blir de inte längre så mycket snabbare, utan får ofta istället fler kärnor. En kärna är som en egen minidator och detta gör att man kan köra allt mer instruktioner samtidigt i datorer. Men för att köra saker samtidigt måste man anpassa hur man skriver program.

Ett exempel är grundläggande datastrukturer som köer. De representerar ungefär en kö i mataffären där folk ställer sig i slutet av kön och kallas fram till kassan från början av kön. Men i vårt fall är det en kö av information istället för personer. När ett program kör på flera kärnor kan man se det som att det finns väldigt många kassor. Att bara ha en kö blir då långsamt då alla kassor måste tävla om att ropa fram den längst fram i kön hela tiden. Istället har vi ofta flera köer, kanske en till varje kassa, vilket gör det hela smidigare men inte längre lika rättvist (exakt) då köerna rör sig olika snabbt. Detta är i princip grundtanken med en tidigare publicerad artikel om effektiva men inexakta datastrukturer (tänk köer) som detta arbete bygger vidare på.

Detta arbete vidareutvecklar dessa datastrukturer och möjliggör dem att ändra hur inexakta de är under körning, vilket kan liknas vid att ändra antalet köer. På detta sätt kan man anpassa inexaktheten efter hand beroende på belastning, önskad precision, tillgängliga kärnor och så vidare för att få ett mer flexibelt program.

Vår lösning för kön illustreras i bilden nedan och kan liknas vid att ställa in start och stopp skyltar i olika köer när man vill öka respektive minska antalet. Dessa ställs in längst bak i köerna och flyttas fram allt eftersom andra köer rör sig framåt. Vi presenterar en lösning för andra typer av köer som bygger på att man håller koll på förändringar i en extra kö vid sidan av dem ursprungliga.



Våra resultat visar att det inte är stor skillnad i prestanda mellan våra mer flexibla köer och de ursprungliga om man inte tillåter antalet att ändras. Dessutom bevisar vi deras korrekthet och begränsar hur inexakta de är beroende på deras parametrar. Att de nya datastrukturerna inte är mycket långsammare än de vanliga, men nu är flexibla, gör att de kan öka totala prestandan över längre körningar om man utnyttjar deras flexibilitet.