CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Relaxed Priority Queue & Evaluation of Locks

A semantically relaxed priority queue and an experimentally driven comparison of locks and atomic operations for the sake of relaxation

Master's thesis in Computer science and engineering

ANDREAS RUDÉN
LUDVIG ANDERSSON

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2023

Master's thesis 2023

Relaxed Priority Queue & Evaluation of Locks

A semantically relaxed priority queue and an experimentally driven comparison of locks and atomic operations for the sake of relaxation

ANDREAS RUDÉN
LUDVIG ANDERSSON

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2023

Relaxed Priority Queue & Evaluation of Locks
A semantically relaxed priority queue and an experimentally driven comparison of locks and atomic operations for the sake of relaxation
ANDREAS RUDÉN
LUDVIG ANDERSSON

Supervisor: Philippas Tsigas, Department of Computer Science and Engineering, Chalmers
Advisor: Kåre von Geijer, Department of Computer Science and Engineering, Chalmers
Examiner: Peter Ljunglöf, Department of Computer Science and Engineering, Chalmers

Master's Thesis 2023
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2023

Relaxed Priority Queue & Evaluation of Locks
A semantically relaxed priority queue and an experimentally driven comparison of locks and atomic operations for the sake of relaxation
ANDREAS RUDÉN
LUDVIG ANDERSSON
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

We present a new, lock-free and concurrent priority queue, utilizing some ideas from [1] by Rukundo et al., that relaxes the traditional sequential semantics of the *delete_min* operation to achieve better scalability and performance. This relaxation is such that *delete_min* operations can be k-out-of-order when compared to sequential semantics, for which k has a well-defined upper bound. We experimentally compare our priority queue to established, concurrent priority queues, both with relaxed and non-relaxed semantics, and find that ours performs well.

Furthermore, we conduct tests with locks using data structures from [1] by Rukundo et al., comparing the data structures' performance when making use of a mix of atomic instructions and locks to that of the original design, which only utilizes atomic instructions. This allowed us to use different underlying data structures, and 3 different data structures using locks were tested; a linked list, a dynamic array, and a list of small arrays. We find that in some instances, this leads to increased cache-locality in data access patterns, and therefore an overall increase in performance. As an example of this, a speedup of up to 7.6 times was observed for the largest relaxation of the queue when using an array, compared to the original, with large improvements for other data structures as well.

Keywords: Concurrency, Data Structures, Algorithms, Priority Queue, Semantic Relaxation, Lock-free, Scalability, Performance

Acknowledgements

We would like to thank our supervisor, Philippas Tsigas, for giving us the opportunity to work on this project. Our advisor, Kåre von Geijer, for participating in many productive discussions. And Peter Ljunglöf, our examiner, who has provided elaborate feedback on this text. We would also like to thank all our family and friends for being both encouraging and supportive.

Andreas Rudén, Ludvig Andersson, Gothenburg, 2023-06-29

Contents

List of Figures

List of Algorithms

1

Introduction

In the twentieth century, advances in technology would regularly and reliably result in increased clock speeds of hardware, allowing software to gain performance "for free". Today, this type of gain has slowed down, and instead, we observe advances in technology that bring small improvements to clock speed, but regular increases to available parallelism. As such, exploiting that parallelism is one of the unfinished challenges of modern computer science, according to Herlih et al. in [2].

One tool available when trying to utilize parallelism is concurrent data structures, which are data structures designed to be accessed by multiple processes running in parallel. This creates a concern regarding correctness, and synchronization must often be employed to ensure it. Synchronization could cause congestion of the data structure if multiple processes try to access it at the same time and/or frequently enough. As Attiya et al. point out in [3], creating efficient concurrent data structures is known to be a difficult problem of fundamental importance. To derive more efficient data structures, it is often desired to reduce or entirely remove synchronization. However, it can be shown that it is impossible to eliminate certain synchronizations and retain the exact behavior of classical data structures.

To address the scalability bottlenecks of concurrent data structures, it has been suggested by Shavit in [4] that one can relax what constitutes semantically legal behavior to derive new variants of data structures more suited to multicore systems. This reduces the demands we place on a structure's consistency and liveness conditions, in favor of performance and scalability. One type of relaxation that will be used extensively in this paper is the idea, defined in [5] by Afek et al., to relax the *linearizability* condition with a bound on the introduced nondeterminism. That is, each operation must be linearizable at most at some bounded distance from its strict linearization point. This will be referred to as *k-out-of-order* relaxation, where k denotes the upper bound of the distance an operation may be separated from its classical counterpart.

In the paper [1] Rukundo et al. introduce the idea of constructing a framework for k-out-of-order relaxation, with two tunable parameters to affect the relaxation. Our thesis builds on their ideas, and the details of their paper will be examined in Chapter 2. The core of their idea, however, is the existence of multiple substructures that are functionally identical to each other, but otherwise independent. For example, a relaxed FIFO queue in the framework may consist of a handful of atomic FIFO queues as substructures, where the point of the framework is to let threads choose

a substructure to issue *enqueue* or *dequeue* operations to. This idea may remind the reader of replication, but it should be emphasized that no data is replicated, and no substructure shares any element with another substructure. The framework manages the overall situation and ensures that the k-out-of-order bound is upheld. The design also encourages threads to stay local to a given substructure, to limit the number of *collisions* that may be observed. Therefore, the expected usage is for the number of substructures to be greater than the number of threads, although this is one of the configurable parameters.

The framework mentioned in the previous paragraph was used to successfully implement relaxed counters, stacks, queues, and deques. In this paper, we will expand upon their ideas, and derive an adaptation of their framework that we will use to implement a relaxed priority queue. This is the subject of Chapter 3.

In Chapter 4, we examine some implications of the framework striving to keep threads local to minimize collisions. In particular, the experimental question of replacing the atomic substructures with lock-based counterparts is considered. The reason for this avenue to be considered is, as mentioned, the assumed low *contention* within the substructure, due to how the framework operates. Consequently, the extra work that goes into a correct lock-free implementation may be outweighing the work of an uncontested lock and a simple, cache-friendly, data structure. As such, we will test the performance impacts of replacing the previous substructures with a handful of different cache-friendly data structures and introduce locks to maintain correctness.

2

Background

In this Chapter the necessary background needed to understand the later chapters in the thesis is presented. First, the basics of concurrent programs, lock-free data structures and their correctness is presented (Section 2.1, 2.2). Following that, Section 2.3 introduces the idea of semantic relaxation, after which a large section (2.4) is devoted to explaining the "2D framework" paper [1] that this thesis builds upon. Towards the end of the chapter the definition of a priority queue is given in Section 2.5, and lastly, some related work is presented in Section 2.6.

2.1 Briefly on Concurrent Programs

In contrast to serial computation, where the execution of a program is done by a single *Execution Core* (EC) at any given point of the program's runtime, a parallel computation involves the potential for more than one EC to execute the program code simultaneously. If we set aside the ideal, but trivial case, in which the program has every EC working on an independent part, then we are dealing with the potential of more than one EC wanting to read or write to shared data. In such a situation, the programmer must consider in what order the program's instructions will be executed. In the single EC case, the program's instructions are conceptually executed in order[1], and thus the execution is deterministic. This is not the case for a parallel execution, in which factors such as EC frequency, memory access times and interleaving cause instructions to execute in a non-deterministic order.

With the programmer unable to rely on the order of the instructions, some extra work needs to be done to ensure the program behaves as intended in every *execution ordering*. That is, one must retain the *semantic correctness* for shared objects. The notion of correctness for a concurrent object is based on some form of equivalence to sequential behavior. One type of correctness is that of **sequential consistency**. Intuitively, sequential consistency means that for every run of a program, the method calls appear to happen one at a time in some sequential order that is consistent with the order of method calls in the program's source code. The formal definition of sequential consistency as defined in [6] is as follows:

[1]This is of course not the case with modern hardware, but the perceivable effect remains the same.

Every concurrent object is assumed to have a serial specification that defines two things: A set of methods as ordered pairs of call and response events, and a set of legal method sequences. Now, given an execution σ, let $ops(\sigma)$ be the sequence of call and response events appearing in σ in real-time order[2]. Also, given a sequence s of method events and a process p, let $s|p$ denote the restriction of s to method events by p. With the notation introduced, the definition itself is:

Definition 2.1.1 (Sequential consistency)**.** An execution σ is *sequentially consistent* if there exists a legal sequence τ of methods such that τ is a permutation of $ops(\sigma)$ and, for each process p, $ops(\sigma)|p$ is equal to $\tau|p$.

Another, stricter, notion is that of *linearizability*. Linearizability has a similar property to that of sequential consistency, except instead of the requirement that any valid order must be consistent with the program order, with linearizability any valid order must be consistent with the real-time order. I.e., the real-time order of method calls must be preserved. Formally, [6] defines it as follows:

Definition 2.1.2 (Linearizability)**.** An execution σ is *linearizable* if there exists a legal sequence τ of methods such that τ is a permutation of $ops(\sigma)$, for each process p, $ops(\sigma)|p$ is equal to $\tau|p$ and furthermore, whenever the response for operations op_1 precedes the call for operations op_2 in $ops(\sigma)$, then op_1 precedes op_2 in τ.

In an effort to make the definition more intuitive, we could think of linearizability as being the requirement of a fine-grained implementation of a concurrent object's method to have the same effect as an instantaneous atomic method. With that conceptualization in mind, it may seem rather intuitive that a common way in which we show that code is linearizable is to identify an instant in the implementation where the effect of the method takes place. I.e., a point in the code where the method's impact becomes observable to the rest of the system. Such a point is known as a *linearization point*, or LP. Thus, showing that a concurrent object is linearizable can be equivalent to finding an LP for every method of the object.

Let us consider some examples to better grasp the differences. First, we write a simple program with a single thread operating on a variable x. The code can be seen in listing 1.

```
x.write(1);
x.write(3);
x.read();
```

Listing 1: The code thread 1 executes.

In figure 2.1 we see an execution ordering of this program which is *not* sequentially consistent. This is because thread 1's read operation observes the value 1 instead of 3, which is not consistent with the program order.

Let us consider a program with two threads working on a First In, First Out (FIFO) queue. Initially the queue is empty, and the two threads run the program source code

[2]For simplicity, the implicit clock is assumed granular enough such that no two events are truly ever at the same time. This assumption could be lifted by defining a systematic approach to resolving collisions, for instance, by processor id.

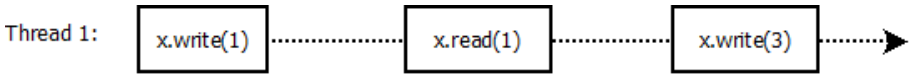

Figure 2.1: Thread 1 reads X=1 which is inconsistent with program order, making this execution *not* sequentially consistent.

given in listing 2. In the execution given in figure 2.2 the real-time order is such that thread 1 is the first to enqueue 1 onto the queue before thread 2 enqueues 3, and as such, by real-time ordering and FIFO semantics, we expect that this means that the dequeue method will yield 1. However, in the execution of figure 2.2 thread 1 and thread 2 can be seen enqueueing and dequeueing in parallel, and thread 1 obtains the element 3 which thread 2 is in the process of inserting, rather than the expected element 1. This violates the FIFO semantics in terms of real-time ordering. On the other hand, by sequential consistency, which is concerned with program order not real-time order, this is a correct execution. But, it is not linearizable.

```
Thread 1:           Thread 2:
q.enqueue(1);       q.enqueue(3);
q.deqeue();
```

Listing 2: Two threads operating on a queue.

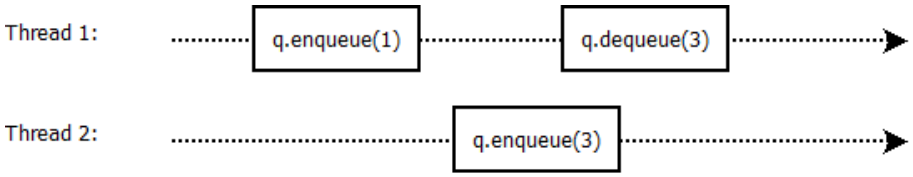

Figure 2.2: An execution ordering of listing 2 which is sequentially consistent, but not linearizable.

Linearizability is the property we will concern ourselves with in this report, so it stands to reason we should consider how we might go about *showing* that something is linearizable. When designing a method for a concurrent object that will write to shared data, the easiest way to ensure correctness and be able to show linearizability is often by utilizing a *lock*. That is, access is compartmentalized to a *critical section*, within which processes are guaranteed exclusive access by guarding the critical section with a lock. A simple pattern of (1) *acquire* the lock, (2) do work, and (3) *release* the lock can be repeated for all relevant methods. This makes it often easy to show linearizability, as we can pick any point between the acquisition of the lock and the release of the lock as our linearization point, and show that every method has such a point.

The usage of locks is often simple and effective, but sometimes the approach has drawbacks. For more fine-grained control, systems that provide *atomic operations*, instructions that unify the behavior of what is traditionally multiple instructions, come in handy. For example, a processor may implement the atomic instruction *compare and swap*, $CAS(x, e, n)$, as part of its instruction set. $CAS(x, e, n)$ takes

the location of a word[3] in memory x which is read and compared against the expected value e. Then, if and only if the actual and expected values are equal, the memory is updated to the new value n. The instruction will indicate success or failure, such that the programmer will be able to tell if the write happened or not. The instruction takes place in one atomic step, and no intermediate state is visible to the system.

To show linearizability for a method using atomic operations, we typically need to identify a single point in the implementation for which the effects of the method become visible to the rest of the system. This point will most likely be directly following the success of some atomic operation in the code.

2.2 Lock-free

Before considering the main focus of this thesis, the relaxation of semantic correctness, it is important that the reader is familiar with the concept of *lock-free* data structures. In this section a lock-free stack will be gradually developed, to showcase ideas and pitfalls that are of high significance when moving into the main subject of this thesis, relaxation. This section is in part inspired by Shavit's paper [4].

Consider a classic stack, with two methods: *push* and *pop*. Push is used to add elements to the stack and pop is used for taking elements off again. In particular, the intuitive semantics of the stack are such that pop returns the element most recently added by push. That is, a stack has Last In, First Out (LIFO) semantics.

When using a stack in a concurrent setting, the correctness of this LIFO semantic needs to be considered. A simple way to do so is to employ a single lock that guards the critical sections (the parts where the stack is modified), by acquiring it at the start of both push and pop, and releasing it at the end of the respective operations. This is a true and tested solution and would indeed provide a correct concurrent stack. However, it comes with some drawbacks. As an example, threads that need to enter the critical section will need to be blocked if the lock is already held. And, if the thread inside the critical section fails or is suspended, e.g. by the system's thread scheduler, no thread will (temporarily) be able to make progress through the critical section. In a *lock-free* design the algorithm is *non-blocking* and the progress of some thread is guaranteed.

To make a lock-free stack, a processor with atomic instructions can be used to retain LIFO correctness without blocking threads. This could be achieved by instead making it a "race", where simply the first thread to succeed in its atomic instruction is the first thread to make progress (by definition, every thread failing is not possible). Notice that this is non-blocking and guarantees that some thread makes progress, as such it is lock-free. One way to construct this idea into a data structure is to employ a linked list, where the stack is a pointer to the first node in the list; the top element.

[3]Or some size that is a multiple of the processor's word size, dependent on its instruction set.

The lock-free implementation is given in algorithm 1. The algorithm uses an object-oriented design, and *Node* is an implicitly defined class with two data members: a pointer to the *next* element in the list and a data entry *value* which holds the stored value. Note that the algorithm is incomplete. In particular, if any *contention*[4] occurs, at least one CAS will not succeed (the expected value is mismatched). This implies that we need to manage contention. In the stack example, a simple loop will be used that retries the CAS instruction until success. It should be noted that more sophisticated methods exist, such as *exponential backoff*: Every time a CAS fails, the thread delays for a random time before attempting again. A thread will double the range from which it selects its random backoff delay each time the re-attempt fails[5]. This constitutes a correct lock-free stack, and it is simple to show that the stack is linearizable. The CAS operation that succeeds can be selected as the linearization point, and in doing so it can be shown that the stack's behavior is equivalent to a sequential stack where the methods take effect when the CAS instructions succeed.

Algorithm 1: Skeleton of a Lock-Free Stack

```
1.1  Function Push(x):
1.2      oldTop ← Top
1.3      newTop ← new Node{next ← oldTop, value ← x}
1.4      return CAS(Top, oldTop, newTop)
1.5  end
1.6
     Precondition : Top must not be null.
1.7  Function Pop():
1.8      oldTop ← Top
1.9      newTop ← oldTop.next
1.10     if CAS(Top, oldTop, newTop) then
1.11         return oldTop.value
1.12     else
1.13         return error
1.14     end
1.15 end
```

However, in practice the lock-free stack may have a more deceptive error: an *ABA* error. When an atomic operation such as compare-and-swap is in play, the structure is generally of the following form: obtain some data that will be the "expected value", do some work to create the "new value", and then perform a CAS operation to swap them. Between the point of obtaining the expected value and the CAS instruction, the actual value might have changed to be something else, but then changed back again. Depending on the assumptions made, re-observing the old expected value may cause erroneous behavior.

[4]Two or more threads trying to alter the top pointer simultaneously.

[5]The randomness and exponentiality are to circumvent the problem of repeated failures getting stuck in a pattern.

In listing 3 a sample implementation of the pop method from algorithm 1 that is vulnerable to ABA errors is presented. The implementation is given in C, a manual memory language, targeting the gcc compiler. In the implementation, the decision has been made that for algorithm line 1.8 the implementation will copy the memory address of **Top** as the expected value. Furthermore, upon successfully finishing algorithm line 1.10, the node's memory will be freed as it is no longer part of the stack.

```
#define CAS(m, e, n) __atomic_compare_exchange( \
    m, e, n, false, __ATOMIC_SEQ_CST, __ATOMIC_SEQ_CST)

typedef struct node   bool pop(DataType* out)
{                     {
  struct node* next;    while (true)
  DataType value;       {
} node_t;                 node_t* old_top = top;
                          if (old_top == NULL) return false;
volatile node_t* top      node_t* new_top = old_top->next;
                          if (CAS(&top, &old_top, &new_top))
                          {
                            *out = old_top->value;
                            free(old_top);
                            return true;
                          }
                        }
                      }
```

Listing 3: Faulty example implementation in C targeting *gcc* of pop from algorithm 1.

In figure 2.3 an execution showcasing the ABA error is given. Thread X, currently executing pop, is suspended after having copied the address of *top* but before executing the *CAS* instruction. In the interim, before X is resumed, the *top* is popped by another thread, Y. Upon resuming, this should not cause an issue, as we will now observe a new address as *top* and X's *CAS* will fail. However, in this particular execution, push is also called by Y before X is resumed. The memory allocator of the system, having correctly reclaimed the memory for the old top, when processing the request to allocate a new node, chooses to reuse the exact same memory location. This is correct behavior; Y puts that node as the new top of the stack. But, when X is resumed, it will try doing its *CAS*, and succeed, despite its expected value having come from an old node; this is because the new top happens to be on the same memory address, and that is the expected value we used for *CAS*. This causes the actual head to be lost, and for X to return a duplicate of a value Y already popped.

To amend this implementation error, a monotonically increasing counter could be introduced alongside the memory address. By having the counter output unique values, and taking identity to be dependent on both it and the memory address, every

node has a unique identifier, and the re-observation that caused the previous error is impossible. As such, the ABA error is eliminated from the sample implementation. The corrected code has been reproduced in listing 4.

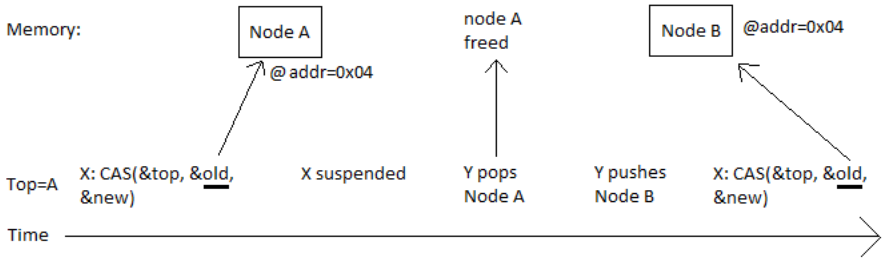

Figure 2.3: Execution showing the problem of the code in figure 3.

```
#define INC(x) __atomic_add_fetch(x, 1, __ATOMIC_SEQ_CST)

typedef struct
{
  struct node_t* ptr;
  size_t counter;
} pointer_t;

typedef struct node
{
  pointer_t next;
  DataType value;
} node_t;

volatile
size_t counter;
volatile
pointer_t top =
  { .next = NULL,
  .counter = 0 };
```

```
DataType pop()
{
  size_t cnt = INC(&counter);
  while (true)
  {
    pointer_t old_top = top;
    if (old_top.ptr == NULL)
      return false;
    pointer_t new_top = {
      .ptr = old_top, .counter = cnt };
    if (CAS(&top, &old_top, &new_top))
    {
      DataType val = old_top.ptr->value;
      free(old_top.ptr);
      return val;
    }
  }
}
```

Listing 4: Fixed version of listing 3.

Even though the lock-free stack works correctly, it still has problems: it scales poorly. This comes about because it has a single point of access which results in a sequential bottleneck. Successful pushes and pops proceed one at a time, ordered by successful CAS instructions. The problem can be alleviated by demanding less of the stack. One such example is a push call which is immediately followed by a pop call. Currently, the algorithm requires the threads to succeed in inserting and removing the element from the stack, going through the sequential bottleneck. Instead of demanding this, a technique known as *elimination* can be used. In this

technique, a thread pushing or popping selects a random index in an *elimination array*. If two threads "meet" at this index while pushing or popping at about the same time, they will exchange the element without accessing the lock-free stack. Only threads which enter the elimination array and do not meet a "partner" will go on to access the underlying stack.

This elimination idea is an improvement, however it still comes with problematic cases. For example, a burst of push calls followed by pop calls will not result in any elimination, and again the stack experiences a sequential bottleneck. This leads to the pursuit of a reduction of demand placed on the stack. In the next section this idea will be expanded upon.

2.3 Semantic Relaxation

One method to reduce the sequential bottleneck (discussed in the previous section) is the reduction of the consistency constraints. In particular, the *relaxation* of linearizability is the idea defined in [5] to relax the linearizability condition, and to do so in such a way that an upper bound can be defined on the introduced nondeterminism. In other words, each method must be linearizable at most at some bounded distance from its strict linearization point. This will be referred to as *k-out-of-order* relaxation.

In the lock-free stack developed in the previous section, the semantics are that of LIFO. To relax these means to abandon the strict ordering of LIFO. Instead, a k-out-of-order stack is correct if it is at most distance k from these semantics. It should be easy to see why this may increase performance and scalability: it is now correct for multiple threads executing pop to make progress at the same time, as they do not need to ensure the element is the literal top. In the next section this idea of semantic relaxation will be put into use.

2.4 2D framework

In this thesis we extend the work done in [1] by Rukundo et al. In their publication they develop a framework for semantic relaxation in which new data structures are derived using multiple independent lock-free data structures, and a shared window to determine which such data structure a thread should operate on. From the framework they derive some variations of a counter, a stack, a queue, and a deque. Sections 2.4.2, 2.4.3, and 2.4.4 are dedicated to an examination of the paper's algorithms. In the next section, the core ideas of the paper will be utilized to semantically relax the stack from the previous section. This serves as an introduction to the more detail-oriented sections that follow after it.

2.4.1 Lock-free stack example, continued

In section 2.3 a lock-free stack was examined. In this section that stack will be utilized to construct a k-out-of-order stack. Given a *width* w the out-of-order stack

contains w instances of the lock-free stack. This can be seen in figure 2.4. In the figure, a thread that wants to push an element onto the stack is incoming. Part of designing the stack will be to direct the incoming thread to one of the w lock-free stacks. This will be referred to as *selecting a substructure*, and algorithms to do so will be presented in later sections. To summarize, the relaxed stack is composed of w lock-free stacks, which will more generally be referred to as *substructures*, and a selection algorithm, which directs threads to a substructure to perform either a push or a pop.

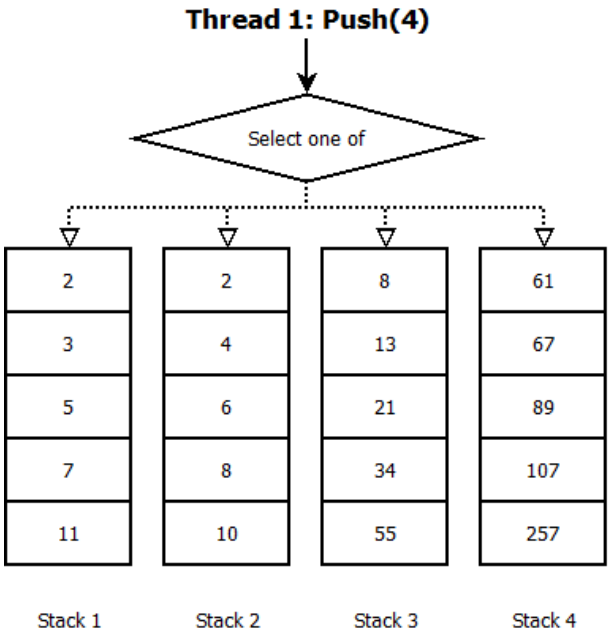

Figure 2.4: Width = 4. Four lock-free stack instances composed into a relaxed stack. A selection algorithm directs the incoming thread to a single stack instance.

So far this imposes no bound on k. This will have to be addressed in how the selection algorithm works. In figure 2.5 there exists, besides the width w, also a newly defined *depth* d and an *offset* o, resulting in the depicted 2D rectangle superimposed on the substructures. This area defines the *operable area* of the relaxed stack. Henceforth this area will be referred to as the *window* into the stack. Elements in any substructure may only be pushed to or popped from if the element is observable within the window. For example, in Figure 2.5, it is *legal* to pop two elements from the first substructure – 3 and 5 – or to push a new element onto it. But for the two middle substructures, no elements can be popped as there are none within the window. For these two it is only legal to push. The last substructure has the opposite situation – it is legal to pop elements, but no new elements can be pushed in currently. With this window construction we have successfully placed a bound on k.

Given the distribution of push or pop operations in a slice of execution, the window may entirely fill up, or become empty. When this happens, the response is to adjust the offset, o. E.g., by increasing its value, the window shifts upwards and new

elements can be pushed again. Atomic instructions are used to adjust the window. If the window is entirely empty preceding a pop, the offset is decreased by a *shift* parameter[6], and similarly if full, the offset must increase before a push can proceed.

The window construction is what the 2D framework uses to bound k, the distance from the true linearization point. The bound for the stack is examined in [7] by von Geijer, who provides an analysis and a proof that shows k is bounded by

$$\left(2shift + depth + \left\lfloor \frac{depth - 1}{shift} \right\rfloor shift\right)(width - 1). \tag{2.1}$$

It is not entirely obvious or intuitive why the bound is so, and the elaboration would entail details that we believe do not provide much insight.[7]

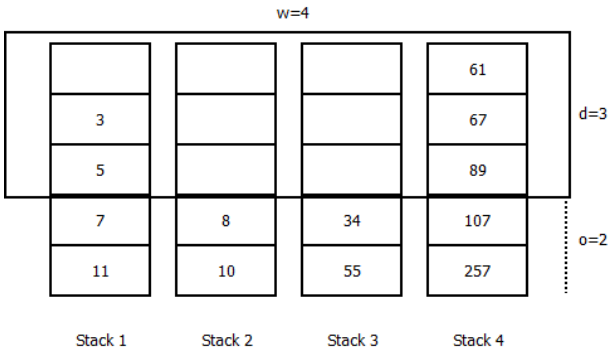

Figure 2.5: Relaxed stack with width 4, and a window depth of 3. The rectangle denotes the part of the stack that can currently be operated on by any incoming threads.

2.4.2 The 2D framework's stack

This section will investigate the details of how the 2D framework's window is operated to create a selection algorithm. This algorithm will then be used to derive a relaxed stack.

In the paper a Treiber stack [8] is used for the substructures. This is the same lock-free stack that was examined in section 2.3. The functionality of the framework is as described in section 2.4.1, with the window, determined by a width, a depth, and an offset, defining the operable area of the lock-free stacks. Threads are directed onto a substructure to perform their push or pop. So far, the functionality is exactly as in the previous stack example. The detail that remains to be filled in is how the selection algorithm works.

To implement the window and the selection of substructures for push and pop operations, the data structure for the window first needs to be defined. It is given in listing 5. The structure has four fields. Two of them are as discussed earlier;

[6] With $1 \leq shift \leq depth$, e.g. $shift = 0.5depth$.

[7] The curious reader is invited to read the cited paper, section 5.2.1.

the window's dimensions are given by *width* and *depth*. The integer *max* fulfills an analogous function to the offset in the previous example, although it denotes the top of the window, rather than the bottom. Thus the observable area of the window is within the range $(max - depth, max)$, and any substructure must have its *count*, the number of elements it holds, within that range after an operation. The last parameter, *version*, is to combat ABA errors (see Section 2.4.1). A single global instance of the window is maintained and read by accessing threads. It should be noted that the width and depth are fixed parameters, and implementations are recommended to maintain them separately from data that will be updated by the CAS instructions.

```
struct Window
{
    width: integer (constant)
    depth: integer (constant)
    max: integer
    version: integer
}
```

Listing 5: The data of a Window structure.

In algorithm 2 the full pseudocode for selecting a valid substructure (i.e. lock-free stack) for pushing or popping has been recreated. The algorithm starts by initializing some data. On line 2.4 the index is chosen to be the last index that the thread reports having accessed, unless the thread also reports contention. If the thread reports contention, it implies pushing or popping from the substructure failed (i.e, the CAS instruction failed). If that is the case, the thread asks for a new substructure, and the index will not be reused, and instead it gets randomly chosen. Following that, a thread-local copy of the global window is taken at line 2.5.

Following initialization, the algorithm will repeat infinitely until a substructure is selected. Skipping over the first if-block for now, consider the lines 2.19-2.23. The substructure referenced by the index is inspected. If the selected operation can be done on the substructure while staying within the window (which is found out by comparing the window parameters to the substructure's count), that substructure, together with the selected index is returned. Otherwise, the version number is inspected. Recalling what was mentioned about ABA errors in an earlier section: the idea is to know if the window has been shifted while we were searching for a valid substructure. If it has been, the selection process needs to start over. The search counter is reset back to zero and the global window is copied anew (line 2.36). If the version has not changed, then the case starting at line 2.25 is taken. This block of code is moving to the next substructure by advancing the index. If the number of *jumps* taken does not exceed the given *max_jumps* configuration parameter, then the algorithm will move to the next substructure by randomizing the index. However, if all jumps are exhausted, the algorithm proceeds to do a linear search by increasing the index modulo the width. Note that this jump functionality exists to avoid the possibility of many threads failing at one step and "moving together" to successive failure points, with only one thread making progress per

step. This jump functionality is a configurable parameter, and whether or not it results in increased performance depends on the other parameters and the type of usage. By default, the number of jumps is kept small ($max\_jumps = 2$).

Lastly, consider the if-block at line 2.8. This branch is only taken once the loop has found all substructures to be invalid candidates, without another thread updating the window during the search. If there exists no valid substructure, an attempt to shift the window happens and then the algorithm starts over. The window shift code is recreated in algorithm 3. At line 3.2 the version of the thread-local window is compared against the version of the global window. This is just a sanity check: if the window has already shifted, no additional shifting attempt is needed. The procedure itself is simple: a new *max* value is calculated and a new window is created. The new window is attempted to be compare-and-swapped to replace the global window, and the thread-local window is used as the expected value. Failure is not checked for, as contention implies that some other thread has already changed the window.

The next Section will examine how a queue can be created with a handful of changes to this procedure.

Algorithm 2: 2D Window

```
2.1  Function SelectSubstructure(op, lastIndex, contention):
2.2      linear_search ← jumps ← 0
2.3      isEmpty ← true
2.4      index ← lastIndex if ¬contention else RandomIndex()
2.5      LocalWin ← Win
2.6
2.7      Loop
2.8          if linear_search = width then
2.9              if op = get ∧ isEmpty then
2.10                 return (X, index)
2.11             end
2.12
2.13             ShiftWindow(op, LocalWin)                    ▷ See alg. 3
2.14             linear_search ← 0
2.15             isEmpty ← true
2.16             LocalWin ← Win
2.17         end
2.18
2.19         X ← SubStructs[index]
2.20         if op = push ∧ X.count < Win.max then
2.21             return (X, index)
2.22         else if op = pop ∧ X.count > (Win.max − depth) then
2.23             return (X, index)
2.24         else if LocalWin.version = Win.version then
2.25             if jumps < max_jumps then
2.26                 index ← RandomIndex()
2.27                 jumps ← jumps + 1
2.28             else
2.29                 index ← index + 1  mod width
2.30                 linear_search ← linear_search + 1
2.31             end
2.32             if X.count > 0 then
2.33                 isEmpty ← false
2.34             end
2.35         else
2.36             linear_search ← 0
2.37             LocalWin ← Win
2.38         end
2.39     EndLoop
2.40 end
```

Algorithm 3: ShiftWindow, helper function

```
3.1  Function ShiftWindow(op, localWin):
3.2      if localWin.version = Win.version then
3.3          if op = push then
3.4              max ← localWin.max + shift
3.5          else if op = pop ∧ localWin.max > depth then
3.6              max ← localWin.max − shift
3.7          end
3.8          newWin ← {version = Win.version + 1, max = max}
3.9          CompareAndSwap(Win, localWin, newWin)
3.10     end
3.11 end
```

2.4.3 The 2D framework's queue

The queue in the framework is defined in a similar way to how the stack works. Some differences need to be considered, however. Firstly, the paper uses an MS queue[8] [9] for the substructures. Secondly, and more importantly, the window functionality runs into some considerations. Unlike the stack, in which there exists a single access point for pushing and popping, the queue has two points of access – elements are *enqueued* at the *tail* of the queue and *dequeued* at the *head* of the queue. As such, positioning a window over the substructures becomes non-obvious. The way the paper resolves this is to simply have two windows, one that defines the operable area for the tail and one for the head.

In algorithm 4 the algorithm of the previous section has been modified to use two windows. Only the differences will be highlit; for the rest of the explanation refer to section 2.4.2. Nevertheless, the first change already appears at line 4.5, where what is copied into the thread-local window is either the global window for enqueuing (WinTail) or that for dequeuing (WinHead). The next change is observed at lines 4.20 and 4.22. Here it can be seen see that the window's *max* field is not compared against the count of the structure, instead there are two constantly growing counters for enqueuing and dequeuing. This is the main difference of this window implementation: it does not shift up and down as the substructures grow/shrink the way the stack did. Instead, it monotonically moves upwards; as does the enqueue and dequeue counters of the substructures. This means that the shift window implementation also needs to change slightly. For the sake of completion, it has been recreated in algorithm 5. As the reader can see, the only functional change is that the operation used changes which window is shifted, and the window is always shifted upwards.

[8]The MS queue is explained in detail in Section 4.1.2

Algorithm 4: 2D Window for queue

```
4.1  Function SelectSubstructure2(op, lastIndex, contention):
4.2      linear_search ← jumps ← 0
4.3      isEmpty ← true
4.4      index ← lastIndex if ¬contention else RandomIndex()
4.5      LocalWin ← WinTail if op = enqueue else WinHead
4.6
4.7      Loop
4.8          if linear_search = width then
4.9              if op = dequeue ∧ isEmpty then
4.10                 return (X, index)
4.11             end
4.12
4.13             ShiftWindow2(op, LocalWin, true)          ▷ See alg. 3
4.14             linear_search ← 0
4.15             isEmpty ← true
4.16             LocalWin ← WinTail if op = enqueue else WinHead
4.17         end
4.18
4.19         X ← SubStructs[index]
4.20         if op = enqueue ∧ X.enqueue_count < WinTail.max then
4.21             return (X, index)
4.22         else if op = dequeue ∧ X.dequeue_count < WinHead.max then
4.23             return (X, index)
4.24         else if
              (op = enqueue ∧ LocalWin.version = WinTail.version) ∨ (op =
              dequeue ∧ LocalWin.version = WinHead.version) then
4.25             if jumps < max_jumps then
4.26                 index ← RandomIndex()
4.27                 jumps ← jumps + 1
4.28             else
4.29                 index ← index + 1  mod width
4.30                 linear_search ← linear_search + 1
4.31             end
4.32             if X.count > 0 then
4.33                 isEmpty ← false
4.34             end
4.35         else
4.36             linear_search ← 0
4.37             if op = enqueue then
4.38                 LocalWin ← WinTail
4.39             else
4.40                 LocalWin ← WinHead
4.41             end
4.42         end
4.43     EndLoop
4.44 end
```

Algorithm 5: Helper function to the queue's window

```
5.1  Function ShiftWindow2(op, localWin):
5.2      if op = enqueue then
5.3          if localWin.version = WinTail.version then
5.4              max ← localWin.max + shift
5.5              newWin ← {version = WinTail.version + 1, max = max}
5.6              CompareAndSwap(WinTail, localWin, newWin)
5.7          end
5.8      else
5.9          if localWin.version = WinHead.version then
5.10             max ← localWin.max + shift
5.11             newWin ← {version = WinHead.version + 1, max = max}
5.12             CompareAndSwap(WinHead, localWin, newWin)
5.13         end
5.14     end
5.15 end
```

2.4.4 The framework's deque

The framework also implements a deque. It uses the substructures found in [10]. Some changes have been made to account for working with the given substructure, however, we believe studying the differences does not illuminate anything that is of importance to this thesis. As such, we invite curious readers to read about it in the paper [1].

2.5 Priority Queue

In Chapter 3 we will develop a relaxed priority queue. As such the definition of a priority queue needs to be examined first.

In its simplest form, a *priority queue* is a data structure that provides the external interface of two methods: *insert* and *delete_min* (or max), with the semantics that *insert* adds an element to the queue with an associated priority (both supplied by the user) and *delete_min* removes (and retrieves) an element subject to the following properties:

Property 1: Given a deleted item i, there exists no item $j \in PQ$ such that $prio(i) > prio(j)$.

Where PQ is the set of all elements in the priority queue and $prio(n)$ is the priority level associated with the element n by some previous *insert* method call. Also, it sometimes holds that:

Property 2: Given a deleted item i, for every $j \in PQ : prio(i) = prio(j)$ it holds that in the execution order of operations $insert(i)$ appears earlier than $insert(j)$, i.e. first come, first served.

Property 1 is the defining property of a priority queue, whereas there exist priority queues that do not adhere to property 2.

2.6 Related works

Concurrent priority queues have had a long history of research in papers such as [11], [12] or [13]. With some of the more performant versions being based on skip lists, such as the one presented in [14] by Lindén and Jonsson, which will be examined in detail in Chapter 3.

While there exist many lock-free and wait-free variants of the priority queue, the scalability of the *delete_min* is an inherent problem, due to the sequential nature of its semantics. Attempts to ease this issue include efforts such as that seen in [15] by Calciu et al., where *insert* operations are carried out in parallel, and *delete_min* operations use an elimination array and batching of operations.

Other attempts include the relaxation of linearization requirements, such as seen in the *k-LSM* priority queue described in [16] by Wimmer et al. In the k-LSM priority queue, the authors present a lock-free design which relaxes the linearization requirements on *insert*, to allow threads to batch up to k inserts, and on *delete_min*, to allow deletion of any of the $k+1$ smallest elements from the set of elements visible to all threads.

While k-LSM places an upper bound on the relaxation, k, other relaxation designs which do not place a hard bound on relaxation include the *Multiqueue* presented in [17] by Rihani et al. The Multiqueue is constructed from cp lock-based priority queues, where $c > 1$ is a tunable parameter and p the number of parallel threads. For *insert*, the Multiqueue samples random queues from the full set of queues until it manages to acquire the lock of one. The *delete_min* operation works similarly, but samples two queues and deletes from the one with a smaller minimum. Consequently, the rank error of the deleted element has a stochastic bound.

3

Relaxed Priority Queue

In this Chapter, we present a method to create a relaxed priority queue (PQ) with a fixed number of priorities. First, the general idea is presented in Section 3.1, after which the substructure is described in Section 3.2 and then the algorithm for selecting a substructure is presented in Section 3.3. In Section 3.4 the correctness of the PQ is considered and a relaxation bound is presented. Finally, an alternative substructure is presented in the last section (3.5), and the changes made to accommodate this are elaborated upon.

3.1 The idea

The PQ reuses the idea of a window to select substructures, as discussed in Chapter 2, but adds a third dimension on top of it: each priority level has its own window. I.e., we have *width* number of substructures that store the elements of the PQ. Together with a window *offset* (that slides around) and a defined *depth*, we have denoted an operable area just like before. Within this area, it is valid for threads to insert and delete nodes. However, as mentioned, a third dimension has been added, and that is the *levels*. Elements stored in the PQ are assigned a priority in the range $[0, levels)$. The priority adds an additional requirement that aligns well with the definition of a priority queue (see section 2.5); the *delete_min* operation must delete an element from a substructure that is within some bounded distance to the actual absolute minimum priority level across the PQ. Note that the presented PQ is correct by property 1, as given in section 2.5, within some bound k, but that property 2 is entirely ignored by our implementation. This allows for a wide choice of substructures, and for the sake of simplicity we will use an array of Treiber stacks, which has been previously described in Section 2.4.1.

The interface of the *insert* method requires a priority level p alongside the element to be inserted. In response, the algorithm implementing the window considers the set of substructures, finding a substructure s for which $(max_p - depth) \leq count_p(s) < max_p$, where max_p denotes the maximum number of elements with priority p any substructure may contain, and $count_p(s)$ denotes the number of elements with priority p in substructure s. Once s has been found, a CAS instruction is attempted to insert the new element and increase the $count_p(s)$ by one.

For the *delete_min* method, the window will locate an element that is at most k-out-of-order from the "front", i.e. from the actual lowest priority. Similarly to insert,

a structure s for which it holds that $(max_p - depth) < count_p(s) \leq max$ is located, and a CAS instruction is attempted to remove the selected element and decrease the count $count_p(s)$ by one.

3.2 Substructure

In this section the algorithm of the priority queue's *insert* and *delete_min* will be explored. Our PQ uses a substructure with *levels* Treiber stacks (see Section 2.4.1), such that each level is stored separately in each substructure. As the chosen priority queue semantics define the in-level ordering to be arbitrary (see property 2 in Section 2.5) the choice of the Treiber stack as the underlying substructure can be replaced by any sufficient data structure instead.

The structure of the priority queue as a whole is described in Listing 6. The "PQ" scope will be implicit in presented algorithms, e.g. just *Windows* instead of *PQ.Windows*. Four types are defined. The *Window* type, which defines an operable area (note that *width* and *depth* are constants and thus not listed as part of the type). In total, the PQ consists of *level* window instances. The window will be explained in detail in the next section. The *Substruct* type, which contains an array of descriptors with *levels* elements. In turn, the *Descriptor* type corresponds to a single Treiber stack, the stack presented in Section 2.4.1 using a linked list structure, storing the top element of the stack and a count of total elements. This Descriptor is its own type to facilitate using it as an expected value for compare-and-swap instructions, as will be seen later. Lastly, the *Node* type is the storage type for elements contained within the stacks, it consists of a value held at that node and a link to the next node in the stack.

```
PQ { Windows: Window[levels], SubStructs: Substruct[width] }
type Window { max: int, version: int }
type Substruct { priorities: Descriptor[levels] }
type Descriptor { top: Node, count: int}
type Node { next: Node, data: Arbitrary Type }
```

Listing 6: The priority queue's data structure.

In Algorithm 6 the *insert* method is presented. Note that each thread maintains a thread-local *index* variable, which is initialized to 0 before any method calls. This variable will be passed, together with the current *contention* state, to the substructure selection algorithm. The semantical meaning being; prefer reusing the last successfully used substructure unless there was a "collision" at the compare-and-swap at line 6.8. The algorithm starts by setting this contention variable to **false** and creating a new head *Node* for the stack at line 6.3. The new node is initialized with the given *value*. At line 6.5 the substructure selection function is used. This function will be explained in detail in the next section, but for now it serves to know that it will select a substructure of the PQ such that insertion into it places the new element into the window's operable area. In particular, it returns the index (into the **SubStructs** array) to the selected substructure and a descriptor for the given

priority. This descriptor is a "snapshot" of the state of the stack before the window coverage was confirmed, and will serve as the expected value for the compare-and-swap at line 6.8. This ordering is important for correctness, which will be considered further in Section 3.4. As mentioned earlier the thread-local index is passed in to facilitate thread-locality and reduce collisions. A new descriptor is created, which will serve to replace the old one, effectively increasing the count and inserting into the stack in one atomic operation (as opposed to two). This happens if and only if the compare-and-swap at line 6.8 succeeds. If not, it implies another thread came first to insert or delete from this substructure's priority. Then, *contention* is set to **true** and the substructure selection is started over, and a new descriptor will be obtained.

Algorithm 6: Priority queue's *insert* method.

```
6.1  Function insert(priority, value):
6.2      contention ← false
6.3      newTop ← new Node{data: value}
6.4      Loop
6.5          (desc, index) ← SelectSubstructureForInsert(priority, index, contention)
6.6          newTop.next ← desc.top
6.7          desc' ← Descriptor{top: newTop, count: desc.count + 1}
6.8          if CompareAndSwap(SubStructs[index].priorities[priority], desc, desc') then
6.9              break
6.10         else
6.11             contention ← true
6.12         end
6.13     EndLoop
6.14 end
```

In Algorithm 7 the *delete_min* operation is given. At line 7.4 the substructure selection can be observed. As with insert, the last used *index* and the *contention* flag is passed to the function to prefer remaining within the most recently accessed substructure if there was no conflict. The selection algorithm is explained in detail in the next section. However, it will either return an "empty" descriptor to indicate the PQ is empty as seen on line 7.5, or it will return a descriptor, the index (into *SubStructs*) of the selected substructure, and the selected priority. In the success case, line 7.8 creates a new descriptor to update the top node of the underlying stack and the count in one atomic operation, as with *insert*. This update corresponds to removing the top node and making the second node (if there is one) the new top. This corresponds to a successful compare-and-swap at line 7.9, which means the executing thread succeeded in removing the top node, and its value is thusly returned as the output of *delete_min*. Otherwise, contention is flagged and the procedure starts over with a new substructure selection.

Implementation Detail: The algorithm for *delete_min* may require extra care by a programmer implementing it. If the programmer is using a garbage collector, then no special attention is needed. But, if they instead are using manual memory handling and freeing nodes in *delete_min*, then the *Descriptor* type needs an ABA protection counter. See Section 2.2 for more details.

Algorithm 7: Priority queue's *delete_min* method.

```
7.1  Function delete_min():
7.2      contention ← false
7.3      Loop
7.4          (desc, index, prio) ← SelectSubstructureForDeleteMin(index, contention)
7.5          if desc is empty then
7.6              return PQ is empty
7.7          end
7.8          desc' ← Descriptor{top: desc.top.next, count: desc.count − 1}
7.9          if CompareAndSwap(SubStructs[index].priorities[prio], desc, desc') then
7.10             return desc.top.data
7.11         else
7.12             contention ← true
7.13         end
7.14     EndLoop
7.15 end
```

3.3 Window algorithm

In this section we present the window algorithms for the PQ. Since the structure shares similarities with the algorithms explained in section 2.4, primarily the differences will be discussed in the text. Firstly, in algorithm 8 the insert method of the PQ is presented. At line 8.7 the case where the entire structure has been searched through without finding a valid substructure to insert into is handled. If the window is entirely full, a shift operation is attempted. After the shift attempt the operable area may contain empty slots that can be used for insertion, and the search is restarted entirely. The details of the shift function are given in algorithm 10.

For each considered substructure there are three if-cases present. The first check, at line 8.14, checks if the substructure has free slots within the window area for the given priority. If it does, the window version is verified and the descriptor is returned. Note that the descriptor is copied before the window check, and as such serves as an expected value for CAS in the substructure, as described in the previous section. The case at line 8.16 verifies that the window is up-to-date and advances through the set of substructures if it is. And the last case at line 8.24 handles the case when the local window is not up-to-date by restarting the entire search.

The procedure to select a substructure for *delete_min* is presented in algorithm 9. Let us consider how this differs from the *insert* method. Firstly, the operation needs to not only select a valid substructure, but also to decide which priority level to delete from. This has the implication that, unlike *insert*, one cannot simply look at one window for selection, instead multiple windows need to be considered. At line 9.5 (among others) an important change can be observed: a local copy is made of every global window, since it is not yet known which priority will be selected. Just like before, this is done to detect window shifts by other threads. Further differences from this can be observed at line 9.17, where a new loop is introduced. For every considered substructure, every priority level must be considered in order of highest priority. At line 9.20 the current substructure is checked for elements with priority

i that lay within the operable region, as defined by the window. If this is not the case, note that care must be taken: simply continuing onto the next priority level may be erroneous. First it must be confirmed that the reason there is no valid element is because the structure has none of that priority. If it did, however, since the window could be shifted (see line 9.11) and an element taken out from that priority instead, afterwards, it would be wrong to take out another element of lower priority from X. The check at line 9.30 is to guard against this occurrence. If a valid substructure was found, the descriptor is returned at line 9.23 together with the priority. Note that before the return of the descriptor, every local window of better or equal priority to the selected priority needs to be confirmed. This can be seen at line 9.21. Again, as noted for *insert*, this check happens after the copying of the descriptor, which is relevant since said descriptor serves as the expected value in the substructure's compare-and-swap. Apart from the discussed points, the rest of the algorithm mimics the behavior of previous selection algorithms.

The window shift functions are given in Algorithm 10 and 11 for *insert* and *delete_min*, respectively. The algorithms are simple; in the *insert* case a new window is created from the local window by increasing the *version* and the *max* parameter, and then using a compare-and-swap instruction to update it atomically. The *delete_min* shift is slightly more involved, needing to search through the priority levels to find the first shiftable priority. Higher priorities (i.e. lower values) are preferred, in accordance with priority queue semantics.

3.4 Correctness

In this section we consider the linearizability of our PQ and its lock-free guarantees. The matching sequential priority queue semantics is that of Property 1 as defined in Section 2.5, but not Property 2.

Lemma 1. *The priority queue as defined by its operations* ***insert*** *and* ***delete_min*** *is lock-free.*

Proof. Consider the operations of the priority queue: *insert* and *delete_min*. These depend on two methods: *SelectSubstructureForInsert* and *SelectSubstructureForDeleteMin*. By inspection of the algorithms it follows that these four operations are non-blocking, since no thread can block another. Furthermore, only *insert* and *delete_min* of these four can fail to progress. This can happen at the respective compare-and-swap instructions. Consider a thread t that fails the compare-and-swap of *insert*. This failure happens if and only if the obtained descriptor used for the expected value no longer corresponds to the state of the stack. As such, it follows that the failure of t implies that some other thread t' has successfully updated the stack. Such an update corresponds to the compare-and-swap in either *insert* or *delete_min*, which implies that t' succeeded in its operation. As such, it follows that a thread fails to progress in *insert* if and only if some other thread succeeds progressing in *insert* or *delete_min*. The argumentation for *delete_min* is left out, but follows the same reasoning. Thus, we can conclude that the priority queue is lock-free. □

Lemma 2. *The **insert** operation is linearizable.*

Proof. An *insert* operation is linearized at the point of a successful compare-and-swap operation. By the chosen sequential priority queue semantics, the relative ordering of element a, b for which $prio(a) = prio(b)$ is arbitrary. From this it follows that *insert* has no out-of-order considerations with regards to the sequential semantics. □

Lemma 3. *The **delete_min** operation is linearizable with respect to k-out-of-order priority queue semantics, where k is bounded by* $(depth)(levels - 1)(width)$.

Proof. A *delete_min* operation is linearized at the point of a successful compare-and-swap operation. Let e be some element of priority l chosen for deletion by the algorithm and let k be the total number of elements in every priority level $0 \leq i < l$. Next we will consider how to bound the value of k.

Let us assume the selected element has $prio(e) = levels - 1$, i.e. the least important priority level. Then k is equal to the total number of elements in priority $0 \leq i < levels - 1$ in the PQ.

Consider e prior to deletion. Let S denote the substructure that e resides in and in particular let s denote the stack of S that e resides in. Also, let $max(win_i)$ denote the *max* field of the window for priority i. Since algorithm 9 searches through the substructure's stacks in order of descending priority (0 is searched before 1, which is searched before 2, ...), the selection of the priority level l which e belongs to can only happen if every stack for priority levels less than l were observed empty in S. Thus, it follows, at the time the thread-local window copies were obtained (which happened before the search started), for every $0 \leq p < l$ it holds that $max(win_p) = depth$. At line 9.19 the state of s is saved into a descriptor. Let t_0 denote the time the descriptor was taken. After that, the version of every window for priority $0 \leq p \leq l$ is verified unchanged. From which it follows, at t_0, for every priority $0 \leq p < l$, it held that $max(win_p) = depth$. As, such, at t_0 there are at most $(depth)(levels - 1)$ higher importance elements than e in any given substructure. And, since there are at most $width$ substructures, there are at most $(depth)(levels - 1)(width)$ elements less out-of-order than e at t_0.

Now, we have shown that the bound holds at t_0. Consider the linearization point of the CAS, let's call it t_1. The success of the CAS implies that there has been no writes to s between t_0 and t_1. Furthermore, the thread does not participate in any other happens-before relations between t_0 and t_1 (except reading the windows, but they are unchanged and it imposes no new orderings on it). From this, it follows that there will always exist a linearization of the concurrent history in which t_0 is immediately preceding t_1, implying the bound holds at t_1 as well.

Finally, because we made the assumption that e is of the least priority, any selection of e from a higher priority will be bounded by k as well (although this bound is unnecessarily loose for such an e). As such, it follows that the stated $k = (depth)(levels - 1)(width)$ holds generally, for any deleted element.

□

3.5 Using a skip list as the substructure

In addition to the relaxed priority queue already presented above, we also tried using a skip list as the substructure. The skip list used was an already existing concurrent priority queue, developed by Lindén and Jonsson in [14]. This substructure is explained in detail in Section 3.5.1, but before that, the differences in the substructure selection algorithm, and how it interacts with the substructure will be explained. In this version, there are no descriptors holding the count for the priority in the substructure. Instead, there is an array holding all the counts for the substructure, and the structure instead only directs the thread to a substructure, where it will attempt to get the minimal item instead of an already selected priority. This means that in practice, due to the concurrency of the structure, there can be a different minimal element than the expected one. If the found priority is at least as low as the expected one, the item is deleted and returned. If the found priority is higher than the expected one, this means that the expected item was already deleted, and this indicates contention. It is of note that it is only deletion of the expected value such that the new minimum that is found is of a higher priority that is a problem. This is because the ordering within a priority level does not matter, so another item with the same priority is acceptable. A freshly inserted item that has lower priority than the expected value is also allowed, since finding such an element means the relaxation bound still holds. This change of removing the descriptor for each priority and replacing it with an array holding the counts introduces some changes to the functions, both *insert* and *get_min*. These changes replace the CAS operation updating the descriptor with two operations; first the operation is performed on the substructure, and then the count is atomically changed after success of the operation. This introduces some problems discussed in Section 3.5.2.

3.5.1 Substructure

The substructure that was used for the relaxed priority queue is as previously mentioned a concurrent priority queue based on a lock-free skip list, developed by Lindén and Jonsson in [14]. This data structure consists of multiple layers, each a linked list, where the upper layers are exponentially more sparse, allowing many nodes to be skipped when traversing these layers. The bottom layer is a normal linked list that spans all nodes in the priority queue in priority order. This means that the most important item is the first item in the list and can be accessed in constant time, while new items inserted have to be inserted at the correct location to keep the list sorted in priority order. This can be seen in figure 3.1.

The data structure supports two operations; Insert and DeleteMin. When inserting a value, a node is created with the given priority and value, and the skip list is searched for the correct position to insert at. The node is inserted at the bottom layer first, and then the upper layers are updated. The node is considered inserted when the node is inserted in the first layer, and can thus be deleted before the upper

layers are updated.

When deleting the first item in the queue, the node is deleted by setting the least significant bit in the pointer from the previous node pointing to that node to 1. This works because pointers are word aligned, and the least significant bit is thus always 0. If the pointer to the first node is already marked as being deleted, the next node is checked until a node that is not already deleted is found. Multiple nodes are marked as deleted using this pointer trick and the memory for these is reclaimed at the same time later in a batch. This deletion works in a similar way to the inserts, where the bottom pointer is updated atomically to point at the new head, and then the upper pointers are updated. This can be done in any order, but is done from the top down for deletions compared to down up for inserts, as the authors had observed a speedup with that configuration.

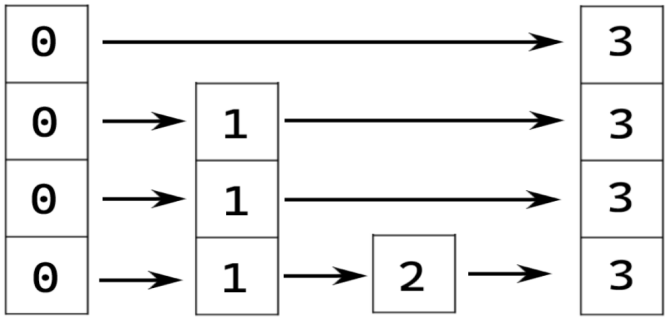

Figure 3.1: Simple example of the general structure of a skip list. A real implementation has pointers pointing to the first node, which here is element 0.

When using this substructure in the relaxed implementation, some changes were made to the substructure. The original implementation had the insertion try again if the CAS operation trying to insert the node at the bottom layer failed, but this was changed to abort the insert and indicate contention, and another substructure is searched for before trying the insert again. When deleting a node, the expected priority found in the window function is checked against the found priority of the node, and if this priority is higher than what was expected, the deletion is aborted since another thread deleted the item first, and this indicates contention. If the deletion would not be aborted at this point, then the relaxation bounds of the structure may not hold.

3.5.2 Comments about linearization

In the first PQ that was presented in this chapter descriptors are used to create a linearization point by updating the count and elements in the PQ at the same time. Since that concept is missing from this second implementation, no strict linearization point exists. This is one of the main limitations that motivated the development of the descriptor-based version. For this version, instead of a specific linearization point,

the effects of an operation take effect before it is visible to the external structure. This is because the counts for the priority in question are updated after the operation on the substructure is completed and there is a window where the effects have taken place, but before the count is updated. This time period introduces the potential for errors when comparing the actual state of the data structure to the correct one, but this error is bounded, since each thread accessing the data structure can only contribute an error of 1 to a count. Due to this, the sum of the error for every count is bounded to be within ± the number of threads.

3.5.3 Analysis of bound

The same reasoning as that found in the proof for the bound of the first version holds for this version of the PQ as well, with only one difference. That difference is that since the substructure is guaranteed to always return the best priority in the substructure, it does not add to the bound, leading to the bound being $(width - 1)(depth)(levels - 1)$. However, due to the issue that no correct linearization point exists and an error can be present, that will be accounted for here.

To reason about this error term, consider the scenario where all priorities in all substructures are filled so that there is only a single spot left for a new item to be inserted in each. Assume now that all threads issue inserts to the same priority in the same substructure at the same time, and that all of these succeed before the count is updated. This results in *threads - 1* too many items in the substructure for that priority, compared to the maximum allowed for the window. This can be repeated for all priorities except the last one, and all substructures except for the substructure where the deletion occurs, resulting in an extra $(width - 1)(levels - 1)(threads - 1)$ items of better rank than the deleted item.

If the issue with the linearization point was to be fixed, then this error term would not be needed.

Algorithm 8: Substructure selection for PQ's *insert*.

```
8.1  Function SelectSubstructureForInsert(priority, lastIndex, contention):
8.2      linear_search ← jumps ← 0
8.3      index ← lastIndex if ¬contention else RandomIndex()
8.4      LocalWin ← Windows[priority]
8.5
8.6      Loop
8.7          if linear_search = width then
8.8              ShiftWindowInsert(priority)                    ▷ See alg. 10
8.9              linear_search ← 0
8.10             LocalWin ← Windows[priority]
8.11         end
8.12
8.13         X ← SubStructs[index].priorities[priority].descriptor
8.14         if X.count < LocalWin.max ∧ LocalWin.version =
              Windows[priority].version then
8.15             return (X, index)
8.16         else if LocalWin.version = Windows[priority].version then
8.17             if jumps < max_jumps then
8.18                 index ← RandomIndex()
8.19                 jumps ← jumps + 1
8.20             else
8.21                 index ← index + 1  mod width
8.22                 linear_search ← linear_search + 1
8.23             end
8.24         else
8.25             linear_search ← 0
8.26             LocalWin ← Windows[priority]
8.27         end
8.28     EndLoop
8.29 end
```

Algorithm 9: Substructure selection for PQ's *delete_min*.

```
9.1  Function SelectSubstructureForDeleteMin(lastIndex, contention):
9.2      linear_search ← jumps ← 0
9.3      isEmpty ← true
9.4      index ← lastIndex if ¬contention else RandomIndex()
9.5      for i ∈ [0, levels) : LocalWins[i] ← Windows[i]
9.6      Loop
9.7          if linear_search = width then
9.8              if isEmpty then
9.9                  return (empty, 0, 0)
9.10             end
9.11             ShiftWindowDeleteMin(priority)                ▷ See alg. 11
9.12             linear_search ← 0
9.13             isEmpty ← true
9.14             for i ∈ [0, levels) : LocalWins[i] ← Windows[i]
9.15         end
9.16
9.17         for i ← 0 to levels do
9.18             ValidWVs ← true
9.19             X ← SubStructs[index].priorities[i].descriptor
9.20             if X.count > LocalWins[i].max − depth then
9.21                 ValidWVs ← (∀j : 0 ≤ j ≤ i : LocalWins[i].version =
                      Windows[i].version)
9.22                 if ValidWVs then
9.23                     return (X, index, i)
9.24                 end
9.25             if ¬ValidWVs ∨ LocalWins[i].version ≠ Windows[i].version then
9.26                 isEmpty ← true
9.27                 linear_search ← 0
9.28                 for i ∈ [0, levels) : LocalWins[i] ← Windows[i]
9.29                 go to 9.41
9.30             else if LocalWins[i].max − depth > 0 then
9.31                 isEmpty ← false
9.32                 break
9.33             end
9.34         end
9.35         if jumps < max_jumps then
9.36             index ← RandomIndex()
9.37             jumps ← jumps + 1
9.38         else
9.39             index ← index + 1  mod width
9.40             linear_search ← linear_search + 1
9.41         end
9.42     EndLoop
9.43 end
```

Algorithm 10: PQ window shift used by *insert*.

```
10.1 Function ShiftWindowInsert(priority, LocalWin):
10.2     if LocalWin.version = Windows[priority].version then
10.3         NewWin ← {version : LocalWin.version + 1, max :
              LocalWin.max + shift}
10.4         CompareAndSwap(Windows[priority], LocalWin, NewWin)
10.5     end
10.6 end
```

Algorithm 11: PQ window shift used by *delete_min*.

```
11.1 Function ShiftWindowDeleteMin(LocalWins):
11.2     for i ← 0 to levels do
11.3         if LocalWins[i].version ≠ Windows[i].version then
11.4             break
11.5         end
11.6         if LocalWins[i].max − depth ≤ 0 then
11.7             continue
11.8         end
11.9         NewWin ← {version : LocalWins[i].version + 1, max :
              LocalWins[i].max − shift}
11.10        CompareAndSwap(Windows[i], LocalWins[i], NewWin)
11.11        break
11.12    end
11.13 end
```

4

Implementing Locks

In this Chapter, the idea of adding locks to the existing data structures found in the previously presented "2D framework" is explored. The motivation behind this is to be able to replace the substructure used, and hopefully gain benefits from better cache locality. Another motivation is that using locks is assumed to work well in the substructure of the 2D framework, as the window selection tries to keep a thread local to a substructure when possible, and thus keeping contention limited and the locks uncontested.

First, the way in which the locks were added to the data structures is presented in Section 4.1. After this the newly developed substructures, that the addition of the locks enabled, are presented in Section 4.3.

4.1 Adding locks to the data structures

When performing any operation on one of our developed data structures, a substructure is selected using the window function, and the corresponding lock is attempted to be acquired. If acquiring the lock fails, this indicates contention and the window function is called again to find a new substructure to attempt to lock. This is repeated until a lock is successfully acquired. When a lock is successfully acquired, the change to the substructure is made with the knowledge that there are no other threads also attempting to change the same substructure. After the change is made, the lock is released and other threads can acquire it.

The algorithms for doing this are quite similar to the existing algorithms from [1], covered in 2.4. The two algorithms below, 12 and 13, are general add and remove algorithms showing the way that locks are used to separate the substructures from the outer layer while still ensuring that the semantics are followed.

Algorithm 12: Generic Add using locks

```
12.1  Function Add(item, ref index):
12.2      possible work before the loop
12.3      Loop
12.4          index ← Window(index)
12.5          if trylock(index) fails then
12.6              continue
12.7          end
12.8          if SubStructs[index].count >
               global_window.max then
12.9              unlock (index)
12.10             continue
12.11         end
12.12         SubStructs[index].add(item)
12.13         unlock(index)
12.14         return
12.15     EndLoop
12.16 end
```

Algorithm 13: Generic Remove using locks

```
13.1  Function Remove(ref index):
13.2      possible work before the loop
13.3      Loop
13.4          index ← Window(index)
13.5          if trylock(index) fails then
13.6              continue
13.7          end
13.8          if SubStructs[index] is empty
               then
13.9              unlock (index)
13.10             return empty
13.11         end
13.12         if SubStructs[index].count <
               global_window.min then
13.13             unlock (index)
13.14             continue
13.15         end
13.16         val ←
               SubStructs[index].remove()
13.17         unlock(index)
13.18         return val
13.19     EndLoop
13.20 end
```

4.1.1 Stack

In the case of a linked list being the substructure (we will come back to the other substructures tested later), when pushing to the stack, the node is created first, and then the loop of calling the window function and trying to acquire the lock starts. For popping, the window search and lock loop is run first, and when the lock is acquired, the node is removed from the stack, and only after releasing the lock, the value is read from the node, which is freed, and the value is returned. With the other two data structures, there is no work before acquiring the lock, or after releasing it, so the structure of both is the same; acquire the lock, perform the operation, and release the lock.

4.1.2 Queue

The queue has two access points per substructure, so when the substructure is a linked list, two locks are used. These two locks are on either side of the queue, with one for enqueue operations, and the other for dequeue operations. This ensures that when the width is small compared to the number of threads using the data structure, the amount of contention is kept lower than if only a single lock had been used. Having two locks per substructure means the enqueue and dequeue operations have to work in such a way that the dequeue operation does not delete a node that the enqueue operation can potentially reference. This was already solved in the existing implementation of the 2d-framework, which uses a concurrent MSqueue by

Michael and Scott [9]. Adapting that structure to our needs was easy, as the original publication contains a version of the same data structure using locks that was used as inspiration. The queue by Michael and Scott has a dummy node as the head (the node to be removed), and dequeue operations read the value of the *next* node, but delete the dummy node. With this strategy, the node to be removed isn't the node that enqueue operations reference (the tail), since there is always a node in between. When the queue is empty, there is still a dummy node in the structure for enqueue to reference, and this node is not deleted since it has no 'next'.

4.1.3 Deque

The deque is similar to the queue in having two access points, however, unlike the queue both ends can be subject to enqueue and dequeue operations. To account for this, two decoupled windows are used, one for enqueue/dequeue on the left end, and one for enqueue/dequeue on the right end. Short of this addition, the deque follows the generic structure for the general add and remove.

4.2 The lock used

To actually perform the locking process, there were a few different possible locks that could be used. However, it was noted that the only use of locks was done by *trying* to acquire the lock, which simplified the possible implementation a lot. To implement the locking of a boolean representing the lock, two atomic operations in **gcc** were used. **___atomic_test_and_set** tries to set the boolean to true, and the return value shows whether it successfully did so or not. The **___atomic_clear** operation simply clears the boolean which allows other threads to set it using the first operation.

4.3 The Substructures

The benefit of having a lock, or multiple locks, on each substructure is that the underlying data structure can be changed from the original lock-free linked list based version to some other data structure where a lock-free implementation is hard, or impossible. There could also be some performance benefits gained from other data structures that offset the lost performance of adding the locks. In this project, we implemented three different lock-based substructures; a linked list, a dynamic array, and a list of small arrays. The linked list was included mostly as a comparison point to the original lock-free version of the framework, where as little as possible was changed, to be able to in isolation see the effects of adding locks to the code.

4.3.1 (Cyclic) Dynamic array

The motivation behind using an array instead of a linked list was to gain benefits from data locality on subsequent accesses to the same data structure from the same

thread. The existing framework already keeps each thread local to the same substructure as much as possible to minimize contention, so a cache-friendly structure like an array should benefit from this too. Since the max number of items in the data structure is not known at compile time, a dynamic array is used, where the array is resized to be twice as large when it fills up.

When using an array, there is no work to be done before or after doing the push/pop operation, so the structure of each operation is the same; acquire the lock, perform the operation on the array, and release the lock.

For the queue and deque, we use an implementation based on a cyclic dynamic array. The index of the head and tail elements are kept in two variables, and these are incremented or decremented when enqueueing or dequeuing items from the array. When any of these counters reach the beginning or end of the array, they wrap around and continue from the other end, allowing the array to be reused after reaching the end, assuming some space has been freed up at the other end. When the head catches up to the tail the array is full and needs to be re-sized by copying all of the elements into a new larger array, in the correct order. An example situation when using the cyclic array can be seen in figure 4.1.

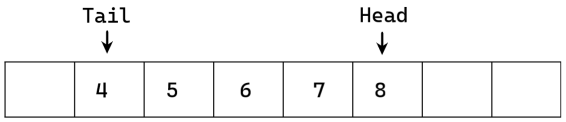

Figure 4.1: A cyclic array with 5 elements. If two more elements were to be inserted at the head, it would then wrap around and insert the next item at index 0 of the array. If yet another item would be attempted to be inserted, the head would catch up to the tail, and the array would need to be resized by making a new array and copying over the old array.

4.3.2 List of small arrays

One of the substructures we have tested for the version using locks is a linked list where elements are small arrays, with sizes aligned to some multiple of the system's cache line size[1]. The idea behind this structure is that it is assumed that it still benefits from the cache locality of an array, but unlike the dynamic array, where growing an array usually involves moving large amounts of data in memory, the addition of elements to the list of small arrays is allocating relatively small amounts of memory at a time, and there exists no need to copy data around to facilitate growth. In figure 4.2 the structure of the list of arrays is given. The presented structure assumes the ability to grow and shrink both the tail and head. If we instead were implementing this substructure for a stack, we could leave out the head's offset, and instead push and pop only at the tail.

The four supported operations are *enqueue_tail*, *dequeue_tail*, *enqueue_head* and *dequeue_head*. The behavior of the methods will be to update the *tail_block* and

[1]We used 128 bytes, which was two cache-lines in our system.

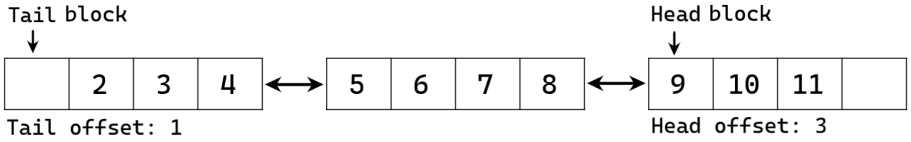

Figure 4.2: A list of small arrays storing 10 elements spread over 3 arrays. If two more elements were to be enqueued at the head, it would then create a new array for the next element, with space for 4 elements. An array is deleted when all the items in that array are deleted from the list of arrays.

head_block pointers when needed by allocating or freeing new blocks. In listing 7 the pseudocode for the four methods has been specified.

```
enqueue_tail(x):
  if tail_offset == 0:
    create new block nb
    nb.next = tail_block
    nb.prev = NULL
    tail_block.prev = nb
    tail_block = nb
    tail_offset = ELEMS_PER_BLOCK
  --tail_offset
  tail_block.data[tail_offset] = x

enqueue_head(x):
  if head_offset == ELEMS_PER_BLOCK:
    create new block nb
    nb.next = NULL
    nb.prev = head_block
    head_block.next = nb
    head_block = nb
    head_offset = 0
  head_block.data[head_offset] = x
  ++head_offset
```

```
dequeue_tail():
  if tail_offset == ELEMS_PER_BLOCK:
    ob = tail_block
    tail_block = ob->next
    tail_block->prev = NULL
    tail_offset = 0
    delete ob

  return tail_block.data[tail_offset]
  ++tail_offset

dequeue_head():
  if head_offset == 0:
    ob = head_block
    head_block = ob->prev
    head_block->next = NULL
    head_offset = ELEMS_PER_BLOCK
    delete ob

  --head_offset
  return head_block.data[head_offset]
```

Listing 7: Pseudocode for the four methods of a list of small arrays

5

Results

To evaluate the performance of all of the developed data structures, both the priority queue and the lock-based data structures were compared against existing versions in benchmarks. For the priority queues developed, these were compared to two existing relaxed priority queues, MultiQueue and k-LSM, and also the concurrent priority queue used as the substructure for the skiplist-based relaxed PQ, while the lock-based structures were compared to the original lock-free versions from [1].

The benchmarks randomly perform operations that add or remove values from the data structure. For our tests, the percentage of operations adding and removing elements was set to 50% each, and the data was randomly generated integers. When the data structure needs multiple values as is the case for the priority queue, the same integer is used for both the priority and the value. The data structure is primed with some initial elements to avoid the situation where the data structure is empty when trying to remove an element, as this would affect the measured performance.

5.0.1 Hardware and settings

The benchmarks were run on a server with two sockets, each with an 'Intel (R) Xeon (R) E5-2687W v2 CPU @ 3.40GHz ', with 8 cores and 16 threads each. These CPUs have 32k L1 caches (data and instruction), 256k L2 cache, and 25.6M L3 cache each. During the benchmarks, a max of 8 cores and 16 threads were used.

The server was running Ubuntu 22.04.2 LTS and the code was compiled using **gcc** with optimization level -O3. During the benchmarks, two thread configurations were run; 8 cores with and without multithreading.

The two developed priority queues were tested with 8 discrete priority levels.

The dynamic array is initialized with a size of 4, and each time it is resized, the new size is 2x the previous.

The arrays in the list of small arrays can hold 14 items each.

5.1 Priority Queue

Each of the two graphs in figure 5.1 contain 4 different widths for the two developed priority queues, with these widths being 1x, 2x, 4x, and 8x the number of threads

used for the benchmark. In addition to these, the substructure used for the skip list was tested without the relaxation framework, as a comparison. This was run with a batch deletion size of 16 only and is shown as a horizontal line in the graph. This batch deletion size is the same size used when the skip list is used as a substructure. Two existing relaxed priority queues, MultiQueue [17] and k-LSM [16], were also tested. MultiQueue is also shown as a horizontal line as only one configuration was run for each thread configuration, while k-LSM was tested for multiple relaxations. The performance of k-LSM for low relaxations was very low, so to better show the results of the newly developed data structures, the first data points for k-LSM are not included.

When looking at the graphs, note that the scales on the y-axis are different between the two graphs.

It can be seen in the graphs that there is quite a large performance difference between the two versions developed, with the version using the skip list substructure only reaching about 25% of the performance of the version using an array of stacks, when running on 8 threads. The gap between the two versions is not quite as large when running on 16 threads.

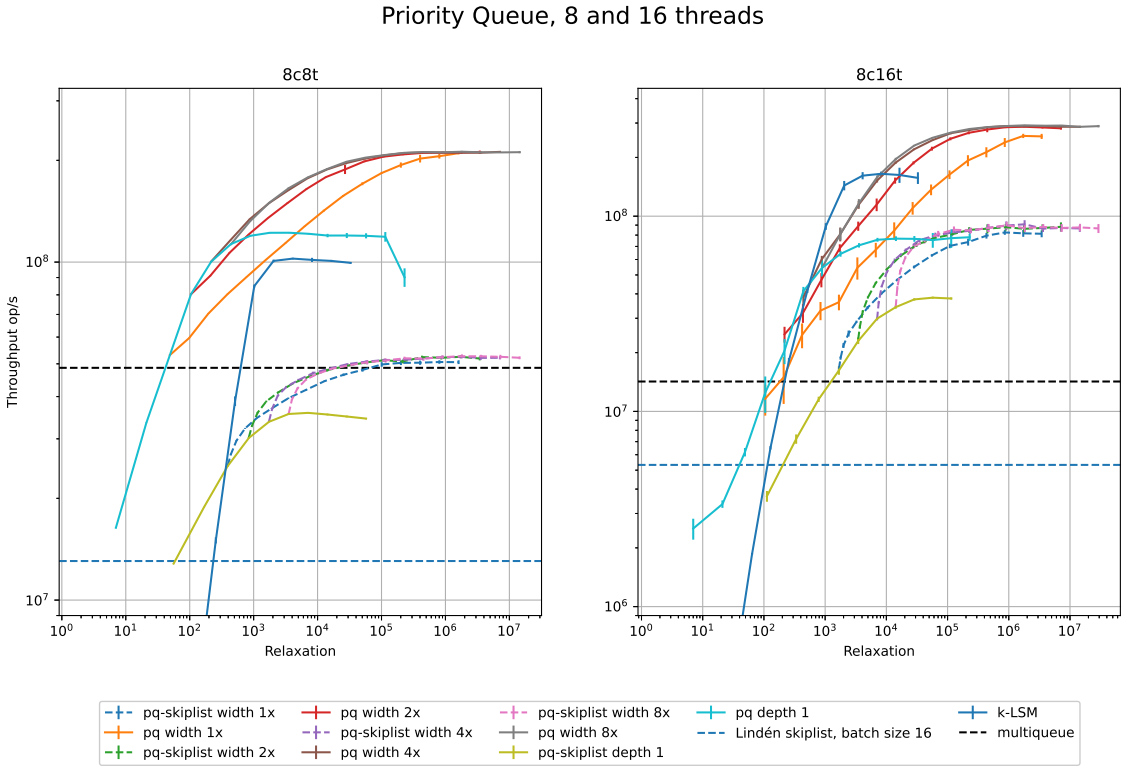

Figure 5.1: Comparison of the two developed priority queues to the concurrent skip list, MultiQueue, and k-LSM. Two different thread configurations, 8 cores with 8 threads and 8 cores with 16 threads.

5.2 Results of adding locks

In the benchmarks for the locks, there were four different configurations, 3 different new substructures (linked-list, array, and list of small arrays), and the original lock-free version to compare against. The original version is shown in the graphs as a dashed line compared to a solid line for the new versions.

5.2.1 Stack

5.2.1.1 Coupled Window

The results for the stack, shown in figure 5.2, show quite different results when looking at the lower part of the relaxation on each width compared to the higher end. With low depth, the original implementation performs best, no matter the width, but for high depths, the newly developed versions using locks overtake the original in performance.

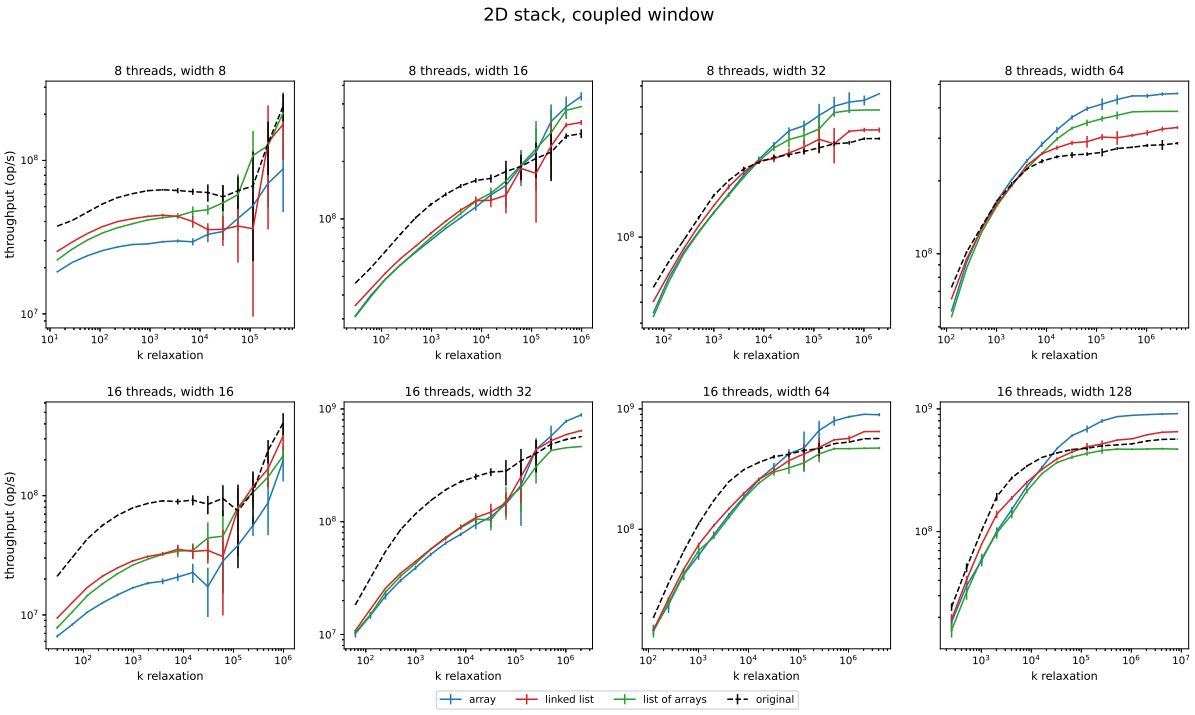

Figure 5.2: Comparison of performance for Stack, coupled window, for different widths going from 1x the number of threads up to 8x the number of threads.

5.2.1.2 Decoupled Window

The results for the decoupled window in figure 5.3 are quite similar to the coupled window when the width is high, but for low widths, there seems to be a larger performance difference between the original and the new versions using locks.

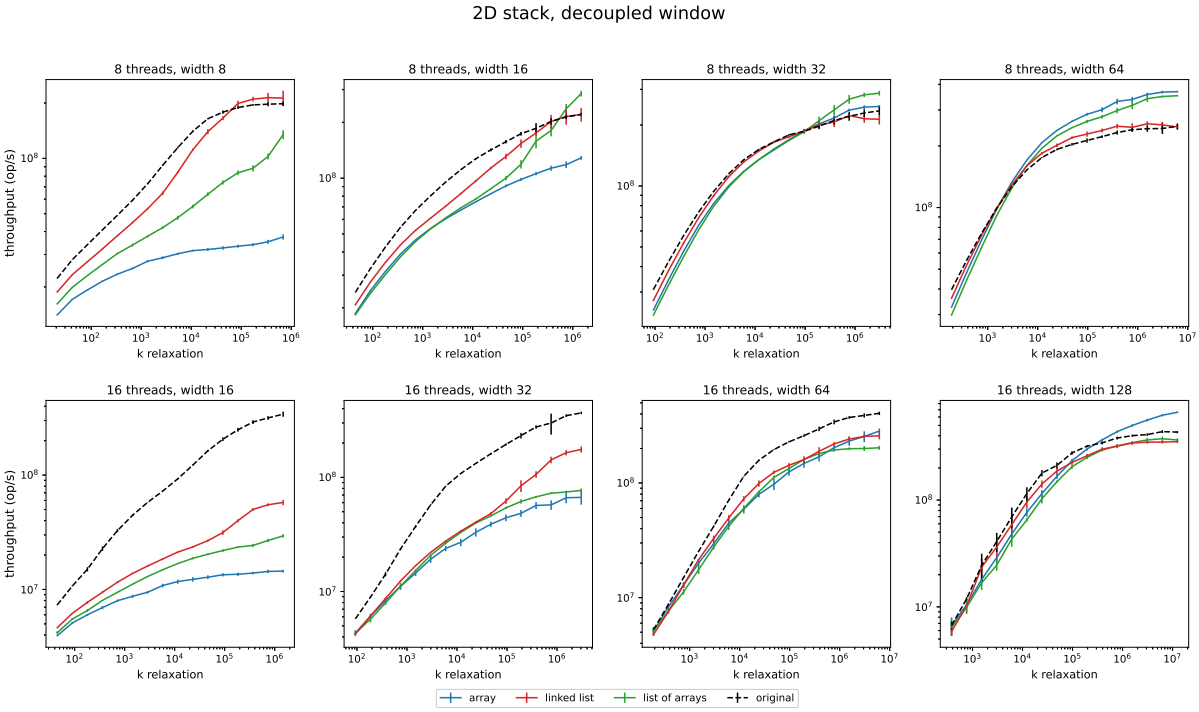

Figure 5.3: Comparison of performance for Stack, decoupled window, for different widths going from 1x the number of threads up to 8x the number of threads.

5.2.2 Queue

The results for the queue are shown in figure 5.4.

An interesting result for the queue is the performance of the linked list with a very low width and depth when running the 8 cores, 8 threads configuration. For these settings, the performance of the linked list implementation is far above that of both of the array-based data structures that we added. This is likely due to the extra access points to the substructure, provided by the use of two locks per substructure in the version using locks, which can also be done on the lock-free original.

When looking at the higher relaxations, especially the higher depths, the array-based data structures show some quite impressive improvements over the original implementation. In the 16 threads, 128 width configuration, the speedup of the array compared to the original reaches an impressive 7.6x speedup for the largest relaxation. This is the highest speedup of any program and version tested.

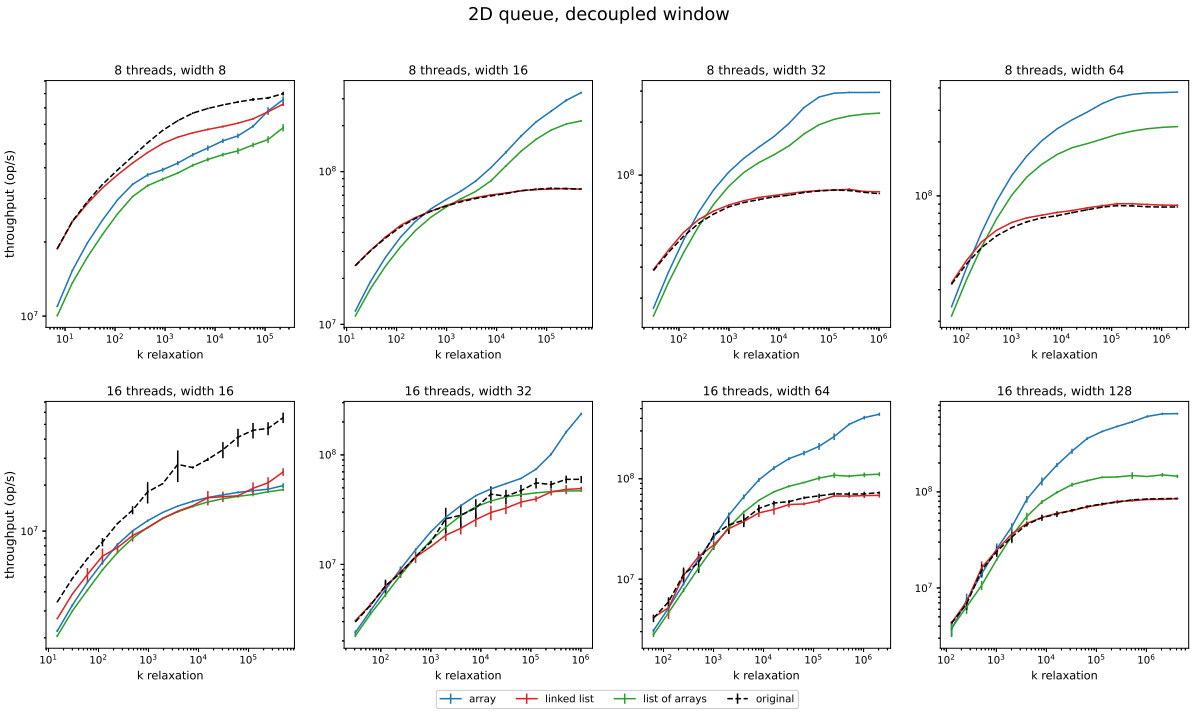

Figure 5.4: Comparison of performance for the queue with different relaxation, for different widths going from 1x the number of threads up to 8x the number of threads, shown from left to right. The top row uses 8 cores and 8 threads while the bottom row uses 8 cores and 16 threads.

5.2.3 Deque

The relaxed deques developed and the original all show some quite odd results. This section will focus on presenting interesting features when comparing the developed versions to the original, while comments on the general oddities are presented in Chapter 6. Due to these oddities, and thus uncertainty about the results, this section is a bit short.

When looking at the results for the deque in figure 5.5, the original performs a lot better than the versions using locks for low widths, with the versions using locks quickly surpassing the original in performance once width increases. This indicates that the lock-free version doesn't scale as well, since even the version using locks and linked lists improve a lot more in performance compared to the original for larger widths.

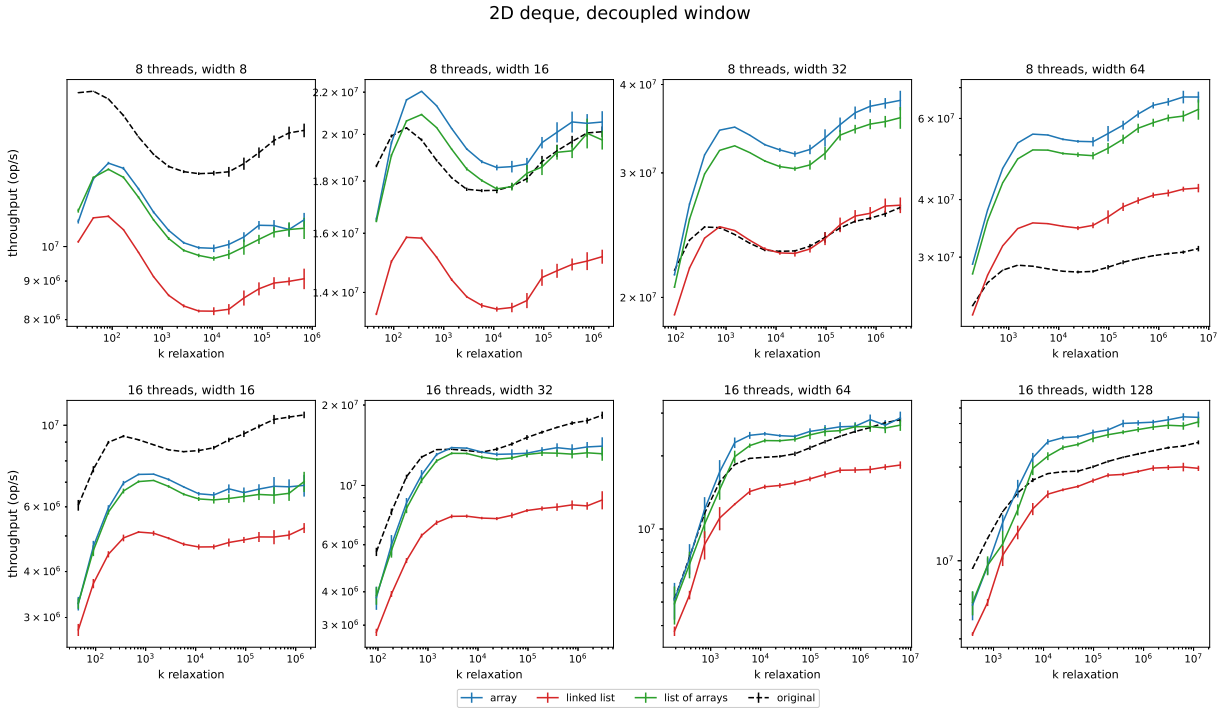

Figure 5.5: Comparison of performance for Deque, decoupled window, for different widths going from 1x the number of threads up to 8x the number of threads.

6

Conclusion

Our developed priority queue shows promising performance in our tests, as presented in the results. However, it should be noted that due to the PQ's fixed level of priorities there is a functional difference to those priority queues which can have arbitrary "key values". Furthermore, our relaxation depends on the fixed number of supported priorities, which may make our PQ unsuited for some use cases.

A possible use case where our PQ would be useful is if the possible priority levels are easily enumerable, such as for a scheduler with a low number of possible priorities. If the priorities are not easily enumerable, such as if the priority is a float, then our approach wouldn't work at all. This could be the case for some kind of graph algorithm where the distance to a node is used as the priority.

Also, due to lack of time, we have not properly been able to test how performance scales with the fixed number of priorities. It is quite likely, due to the selection algorithm, that a much larger number of fixed priority levels would result in a drastic performance downgrade. This, however, has not been tested. Similarly, in our priority queue test, Multiqueue which is likewise tunable (by varying e.g. the number of copies per thread), was only tried in its "default" configuration. To summarize, we believe further testing of how the tunable parameters alter the relative performance of the priority queues in our result, would be of interest.

In the results of the substructures using locks, the array appears to be the clear winner, with the list of arrays being second and the linked list being definitively last among the new substructures. The result of the linked list performing worse was as we predicted, as part of the idea behind adding locks was seeing the effect of cache locality, for which the array structure has the advantage. Previously we mentioned using a list of small arrays as allowing us to combine cache locality together with avoiding the large delays associated with the dynamic array's grow and shrink operations. It should be noted that due to how the tests were conducted, this benefit was not expected to manifest. This is because the tests prefill the structures with a large amount of data and then perform insertion and deletion with equal probability. This results in the dynamic array never needing to grow (and similarly it was not made to shrink), and as such the results are biased towards the dynamic array. This means while we believe it possible to conclude that the cache locality of the array versions is a clearly distinguishable factor, we are unable to draw conclusions about the performance of the dynamic array versus the list of arrays for all but our limited test case.

When shifting the focus to compare the new data structures to the original versions, we can see that there are cases for all types of data structures where the new substructures outperform the original when using high relaxation. This is especially clear for the array, but the other data structures also outperformed the original in some cases. One surprising result was that the linked list using locks outperformed the original in some benchmarks, for example, the stack with a coupled window, where the high width and depth results indicated that no performance was lost when adding the locks, and even a small improvement in performance was seen.

During the development of the substructures using locks, a few different locks were tested to see if performance varied. For some reason the performance of the mutex lock in the **pthread** library performed a lot worse than any of the other locks in that library or the lock that was ultimately used. Since it was realized that the only use of locks is in a try-lock manner, this low performance was not further investigated and instead the method of handling the lock described in Section 4.2 was used.

It could be clearly seen in the results for the deque in the 2D framework that the behavior is quite odd. Since this is not only for our new implementations, but also the original, it is thought to be related to how the substructure selection algorithm works, but this has not been investigated further.

6.1 Future Work

The developed priority queues only have support for a fixed number of priorities. Further development to remove this limitation would be of interest. Our current idea how this may be done is to still retain the notion of one window per priority, but instead of precreating them one would dynamically allocate the windows for a priority the first time it is needed. This means the windows could no longer be kept in an array but instead a data structure with fast key-based lookup would be needed, e.g. a tree or hash-map. When a thread would *insert* into a new priority level, a window would be created and added to the lookup structure. Once the priority level becomes empty (from *delete_min*), the corresponding window would probably either need to be removed, or considered for deletion at some later stage if not reused, to avoid the overall size of the priority queue becoming problematic. Furthermore, the substructures would need to also be made more dynamic. For example, instead of an array of stacks as our first presented version was using, a tree or hash-map of stacks could be employed, where the key would be the priority level. For the version that uses Lindén's skip list, the substructure already supports arbitrary priority keys.

During this work, the relaxation bounds have been calculated for the developed priority queues, but they have not been tested empirically. This would be of interest to test, especially to compare the two versions, and to others, since the one based on a skip list can return a better value than expected, which could potentially affect the rank errors observed.

In the testing of locks, the data structures of the original paper were used for the experimentation. However, we never had the time to try adding locks to the newly

developed priority queues (and replacing the substructure to be something else). This could be of interest to do. Even if there is performance degradation, it would be possible to choose a substructure that defines local ordering within priority levels. While this would not ensure an over-all order, it may alleviate the potential problem of some elements being "stuck" for a longer duration. For some uses this may be an interesting trade-off.

As mentioned earlier, the tests used did not let us truly see if our ideas about the list of small arrays are true or false. Unfortunately, we did not have the time to address this shortcoming. In order to do so, we would have liked to develop tests that try varying types of usage cases, including but not limited to: tests with no prefill, tests with small total size, tests with high ratio of insertions, tests with high ratio of deletions and tests with "streaks" of insertions and deletions.

In addition to the ways that we implemented locks for the existing data structures in the 2d framework, there were a few other ideas that were never implemented due to time restrictions. One of these was to have a lock on every node in the list of arrays. In the case of the queue or dequeue, this would enable concurrent add/remove operations on each end. This was not able to be implemented in the time scope of the project.

The results for the list of small arrays were for a configuration using 2 cache lines for each node, meaning 14 elements on our machine. It might be of interest to look at what the performance impact would be of changing this parameter to a larger value, and if that perhaps could close the gap to the cyclic array when there are more opportunities for cache locality.

6.2 Final words

In this thesis we have developed a new, lock-free and concurrent priority queue that compares well with other established relaxed priority queues, such as k-LSM and Multiqueue, but with the caveat that the number of priorities must be fixed beforehand. And the relaxation and performance will vary with this fixed number of priorities.

By introducing locks, and the use of arrays, extra performance can be gained for relaxed data structures, especially when there is low contention and a thread is allowed to continue working on the same array.

Bibliography

[1] A. Rukundo, A. Atalar, and P. Tsigas, "Monotonically relaxing concurrent data-structure semantics for performance: An efficient 2d design framework," *CoRR*, vol. abs/1906.07105, 2019. arXiv: 1906.07105. [Online]. Available: http://arxiv.org/abs/1906.07105.

[2] M. Herlihy, N. Shavit, V. Luchangco, and M. Spear, *The Art of Multiprocessor Programming*, 2nd ed. Morgan Kaufmann, 2020.

[3] H. Attiya, R. Guerraoui, D. Hendler, P. Kuznetsov, M. M. Michael, and M. Vechev, "Laws of order: Expensive synchronization in concurrent algorithms cannot be eliminated," *SIGPLAN Not.*, vol. 46, pp. 487–498,

[4] N. Shavit, "Data structures in the multicore age," *Communications of the ACM*, vol. 54, no. 3, pp. 76–84, 2011.

[5] Y. Afek, G. Korland, and E. Yanovsky, "Quasi-linearizability: Relaxed consistency for improved concurrency," in *Principles of Distributed Systems: 14th International Conference, OPODIS 2010, Tozeur, Tunisia, December 14-17, 2010. Proceedings 14*, Springer, 2010, pp. 395–410.

[6] H. Attiya and J. L. Welch, "Sequential consistency versus linearizability," *ACM Transactions on Computer Systems (TOCS)*, vol. 12, no. 2, pp. 91–122, 1994.

[7] von Geijer, Kåre, *Highly Scalable Queues and Stacks with Elastic Relaxation*, eng, Student Paper, 2022.

[8] R. K. Treiber, *Systems programming: Coping with parallelism*. International Business Machines Incorporated, Thomas J. Watson Research, 1986.

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

[10] M. M. Michael, "Cas-based lock-free algorithm for shared deques," in *Euro-Par 2003 Parallel Processing: 9th International Euro-Par Conference Klagenfurt, Austria, August 26-29, 2003 Proceedings 9*, Springer, 2003, pp. 651–660.

[11] J. Biswas and J. C. Browne, "Simultaneous update of priority structures," Texas Univ., Austin (USA). Dept. of Computer Sciences, Tech. Rep., 1987.

[12] A. Israeli and L. Rappoport, "Efficient wait-free implementation of a concurrent priority queue," in *Distributed Algorithms: 7th International Workshop, WDAG'93 Lausanne, Switzerland, September 27–29, 1993 Proceedings 7*, Springer, 1993, pp. 1–17.

[13] N. Shavit and I. Lotan, "Skiplist-based concurrent priority queues," in *Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000*, IEEE, 2000, pp. 263–268.

[14] J. Lindén and B. Jonsson, "A skiplist-based concurrent priority queue with minimal memory contention," in *Principles of Distributed Systems: 17th International Conference, OPODIS 2013, Nice, France, December 16-18, 2013. Proceedings 17*, Springer, 2013, pp. 206–220.

[15] I. Calciu, H. Mendes, and M. Herlihy, "The adaptive priority queue with elimination and combining," in *Distributed Computing: 28th International Symposium, DISC 2014, Austin, TX, USA, October 12-15, 2014. Proceedings 28*, Springer, 2014, pp. 406–420.

[16] M. Wimmer, J. Gruber, J. L. Träff, and P. Tsigas, "The lock-free k-lsm relaxed priority queue," *ACM SIGPLAN Notices*, vol. 50, no. 8, pp. 277–278, 2015.

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