CHALMERS
UNIVERSITY OF TECHNOLOGY

Efficiently Calculating Relaxation Errors

Algorithms and Techniques for Measuring Rank and Delay Errors in Relaxed FIFO and LIFO Queues

Elis Kleen
Victor Olin

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2025
www.chalmers.se

Master's thesis 2025

Efficiently Calculating Relaxation Errors

Algorithms and Techniques for Measuring Rank and Delay Errors in Relaxed FIFO and LIFO Queues

Elis Kleen
Victor Olin

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Division of Computer Systems and Networks
Chalmers University of Technology
Gothenburg, Sweden 2025

Efficiently Calculating Relaxation Errors.
Algorithms and Techniques for Measuring Rank and Delay Errors in Relaxed FIFO and LIFO Queues.
Elis Kleen
Victor Olin

Supervisor: Kåre von Geijer, CSE
Examiner: Philippas Tsigas, CSE

Master's Thesis 2025
Department of Computer Science and Engineering
Division of Computer Systems and Networks
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2025

Efficiently Calculating Relaxation Errors.
Algorithms and Techniques for Measuring Rank and Delay Errors in Relaxed FIFO and LIFO Queues.
Elis Kleen
Victor Olin
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

Relaxed data structures have been designed to leverage the parallelism of modern processors more effectively than current strict concurrent data structures can. This kind of data structure relaxes the semantics for performance at the cost of correctness. For instance, to avoid contention between threads, a relaxed FIFO queue may return an element close to the head rather than the head itself. This out-of-order operation can be measured through relaxation errors, most notably rank errors. A current problem within the field of relaxed data structures is that the process of computing rank errors is rarely discussed, and only final results are presented. Furthermore, the methods presented are inefficient and require computationally intensive calculations. Therefore, there is a need in the field for faster and more efficient algorithms to compute rank errors.

In this thesis, we present several approaches and algorithms for calculating the rank errors of FIFO (First-In-First-Out) queues and benchmark them against one another. Specifically, we introduce two algorithms that are hundreds of times faster than the current methods in the field. Additionally, the best-performing algorithms are adapted to calculate delay errors for FIFO queues and rank errors for LIFO (Last-In-First-Out) queues, demonstrating that their performance can be extended to other neighboring problems. Finally, we implement approximations of the calculations to enhance performance at the cost of correctness.

Keywords: relaxed data structures, FIFO queues, relaxation error, rank error

Acknowledgements

As a preface for this thesis, we would like to extend our gratitude toward our supervisor Kåre von Geijer and examiner Philippas Tsigas. We want to thank you for all support, incredibly valuable discussions, and great feedback during all parts of this thesis. This thesis would not have been as successful without it. We are grateful to have worked so closely with you throughout this project and for all that we've learned during it.

Elis Kleen & Victor Olin, Gothenburg, June 2025

Contents

List of Figures

List of Tables

1 Introduction

Concurrent data structures are specifically designed to leverage the capabilities of modern parallel computer architectures. A key feature of these structures is maintaining mutual exclusion to ensure data consistency. Typically, this is accomplished using locks [1]; however, this approach leads to a portion of the execution time being dedicated to synchronization, which tends to increase as the number of threads rises. Time spent synchronizing threads increases as the contention on the lock increases, resulting in more and more threads being idle, waiting for their turn.

To enhance performance and scalability, lock-free approaches to concurrent data structures were introduced [2]. Lock-free data structures maintain correctness by optimistic conflict control [1], meaning operations proceed without locking, assuming conflicts are rare, and are redone if a conflict is detected. Lock-free data structures improve several limitations of their lock-based counterparts, such as robustness, availability, and scalability. Lock-free data structures often outperform lock-based approaches, while still achieving correctness. However, they still experience contention and slow down as the number of threads increases [3], displaying their parallel limitations. The main factor hindering scalability is the correctness requirement; relaxing this requirement unlocks many new approaches [4].

Henzinger et al. [5] discuss the trade-off between correctness and performance, noting that relaxing the semantics of a concurrent data structure can lead to considerable performance enhancements. These data structures, known as relaxed data structures, represent a relatively new and promising area of study in computer science [5]–[11], with several implementations significantly outperforming non-relaxed equivalents, while still being viable in benchmarks such as shortest-path and branch-and-bound that resemble real applications [9].

A concurrent strict FIFO (First-In-First-Out) queue requires that elements are dequeued in the same order as they are enqueued. This leads to high memory contention at the head and tail of the queue [12], but a relaxed FIFO queue *relaxes* this requirement and may return an element near the head of the queue. These types of semantics benefit performance at the cost of correctness, and the trade-off can be measured using *relaxation errors* [5]. Relaxation errors are not errors per se, but merely indicators of how relaxed a data structure's semantics are compared to its non-relaxed equivalent. This thesis considers two main types of relaxation errors, *rank errors* and *delay errors*. For FIFO queues, rank errors refer to how many elements were skipped during a dequeue. For example, a relaxed FIFO queue may return the nth element from the head, which would yield a rank error of $n-1$ as it has skipped all elements preceding it. On the contrary, delay errors refer to how many times an element was skipped. In the previous example, this dequeue would infer one delay error on each of the $n-1$ elements closer to the head than the one dequeued.

Computing relaxation errors has not been standardized, despite being formalized in 2013 by Henzinger et al [5]. Among recent papers, there are two popular, but inherently flawed, approaches to computing these types of errors.

The simplest and most common way [5], [9], [10], [13], [14] is to use timestamps to approximate the linearization point of every mutating action. These timestamps can be used to rerun the execution sequentially to calculate the relaxation errors of every operation. This approach gives relatively low overhead and quite accurate results, but calculating said results can become computationally expensive [14].

The second way to approach the problem is by using a global mutex lock [6]–[8], [15] to capture and order all actions on the data structure. While delivering completely accurate relaxation results, all parallelization is destroyed in the monitored execution, leading to a significant alteration of the data structure's behavior, sometimes leading to a performance of 1% of the original data structure.

Although both methods come with their own sets of challenges, the difference in approach also presents a comparability problem. If different implementations of relaxed data structures measure their errors differently, the results across implementations are not necessarily comparable. This makes selecting the best-fitting data structure for a given problem nontrivial. Furthermore, this could lower the usage of relaxed data structures in real-world applications. With current incomparability and a need for a more efficient computation algorithm, the need for a new standard for calculating relaxation errors is clear.

The primary goal of this thesis is to design, construct, and analyze different algorithms and approaches to calculating rank errors for relaxed FIFO queues. The resulting algorithms should be based on using timestamps to retain data structure performance, while also presenting meaningful amounts of speedup over current algorithms in the field.

A secondary goal of this thesis is to evaluate whether the implementations can be extended toward other neighboring computational problems. More specifically, this would include computing delay errors for relaxed FIFO queues as well as rank errors for relaxed LIFO (Last-In-First-Out) queues. As delay errors model the inverse of rank errors, it should be fairly feasible to extend the presented algorithms to compute delays instead. Similarly, as the semantics of LIFO and FIFO queues are relatively similar, it should be possible to further extend algorithms to compute rank errors in LIFO queues as well. Both extensions will be benchmarked together with baseline implementations to evaluate whether the performance benefit propagates into the related computations.

2
Background

Relaxed data structures relax their semantics for performance. The semantics are often relaxed to a certain *degree of relaxation*, where a higher degree corresponds to dequeueing elements further from the head. [6], [7], [11].

One example of relaxing the semantics of a data structure is the k *out-of-order* paradigm, where k represents the degree of relaxation for *k-relaxed* FIFO queues. In a *k-relaxed* FIFO queue, the queue may dequeue any element from the first $k+1$ elements, not necessarily the first element in order. This paradigm is also not specific to FIFO queues and can be extended for LIFO queues, deques, and priority queues [7].

The degree of relaxation is usually determined offline, before using the data structure. However, there exist relaxed data structures that can tune this degree online, such as the SprayList [6]. These types of data structures fall in the subcategory called *elastic* data structures [7]. The tuning often depends on heuristics measured online, with the SprayList using contention to estimate the number of threads currently operating on it.

The idea behind relaxed data structures is that the need for safeguards is eliminated as threads no longer have to synchronize over a critical resource. This directly decreases the amount of synchronization overhead and directly improves performance. The amount of performance improvement depends on several factors, but it is reasonable that programs with fine-grained workloads that experience more synchronization should experience a greater speedup.

2.1 Computing rank errors with locks

Locks and mutexes that ensure mutual exclusion present an undeniable scalability problem for parallel algorithms and data structures [10]. Most notably, using these functionalities degrades performance as threads stall others. This can be seen in different ways among recent papers, as most implementations use locks in unique ways. In 2015, Rihani et al. presented MultiQueues, which are relaxed priority queues that use several smaller queues, reducing the overall memory contention [8]. Each of the smaller queues is guarded by a lock, which a thread attempts to acquire before enqueueing (dequeueing) an element to (from) the queue. For enqueues, a queue is randomly chosen, while for a dequeue, two queues are selected at random, and the dequeue is done from the queue with the higher priority item. When it comes to calculating rank errors in a MultiQueue, a separate thread is used to keep an ordered sequence of elements in queuing order, which is run alongside the relaxed data structure. When an element is dequeued, the element's position in the ordered sequence is its rank error [8]. How this sequence is kept synchronized and mutually excluded is not covered by the authors, but from the parallel nature of the MultiQueue, it should be safe to assume that this separate thread needs some ultimate form of mutual exclusion. Furthermore, this separate thread may also incur some additional execution time depending on whether the thread can keep up with the rate of queuing and dequeueing operations from the threads using the relaxed data structure.

A similar but simpler approach is to use locks to guard mutating operations, as presented by Rukondo et al. [15], where threads compete for a global lock whenever an element is enqueued or dequeued. In this specific instance, rank errors are calculated online using a linked list alongside the relaxed data structure similar to that of the MultiQueue. This list contains all elements currently in the queue, where rank errors are calculated as an element's distance from the head of the queue [15]. An issue with this approach is that, depending on the frequency of threads trying to acquire the lock, the time each thread spends idling may scale to undesirable levels. Additionally, keeping track of duplicate elements in a linked list incurs both memory and runtime overhead from the traversal of the list, similar to that of the MultiQueue [8].

As mentioned, these two previous papers show several similarities when it comes to calculating relaxation errors. Furthermore, this method of computing relaxation errors is a trend across additional papers [6], [7]. A brief conclusion to this method is that it requires additional memory by storing copies while also inferring some additional execution time by keeping sequential and ordered sets of these copies in one way or another. The degrees of overhead induced by this method are not covered by any of the aforementioned authors, but it is safe to assume that it is non-zero. As previously mentioned, a global lock also introduces behavior that deviates from execution without locks, which can lead to potentially misleading results. In the case of von Geijer et al. [7], the throughput can drop by a factor of 100 in settings with high relaxation and high thread counts.

2.2 Computing rank errors through timestamps

Timer-based approaches reduce the execution time of the program utilizing the relaxed data structure by deferring the computation of rank errors to after the program has finished. Kirsch et al. [14] presented an early rendition of this idea in 2012 where operations performed on a relaxed FIFO queue can be ordered into a sequential history H_S post-execution. The authors then use H_S to construct an ordered set of states that the relaxed FIFO queue assumes throughout its execution, defined as $\mathrm{POOL}(H_S)$. Each element in $\mathrm{POOL}(H_S)$ consists of a state (q_i, l_i), where q_i is a sequence of queue elements and l_i counts the number of dequeues since the last dequeue that removed the oldest element of q. The mean and max rank errors are then defined to be the mean and max of all l's in $\mathrm{POOL}(H_S)$, respectively. However, this method defines rank errors differently compared to the previously discussed lock-based methods. In this paper, rank errors are defined as the number of dequeues needed for the oldest elements to be removed, and not for each element separately [14].

A different approach to linearization is presented by Haas et al. in their benchmarking suite for relaxed data structures *Scal* [13]. In *Scal*, operations are logged with thread-local timestamps for, among other events, linearization points of the relaxed data structure. Although not covered in the paper presenting *Scal*, the authors refer back to a preceding paper by common authors where the history of linearization points is used to calculate the rank error [16]. An important note that Haas et al. present is that this method of computation infers negligible overhead on the program as every thread only timestamps their actions [13]. At program termination, every thread's independent data is merged into a single set of linearization points that can be used offline to calculate exact rank errors. The entire benchmarking suite is available online, although the rank error calculations are left out.

In 2021, Williams et al. published continued work on the MultiQueue and proposed a concrete and improved approach to calculating rank errors [9]. A problem with computing rank errors online is that every calculation needs to know exactly how many elements are ahead in the queue at the time of computing [9]. Furthermore, this implies severe synchronization if every thread were to calculate its errors. Williams et al. instead opt for an approach based on linearization points similar to that of Haas et. al. [13] and Kirsch et. al. [14]. The approach uses a B+ tree to efficiently replay the monitored execution and calculate rank errors for each dequeue operation.

To conclude, these approaches based on linearization points offer efficient alternatives to lock-based computation with minimal overhead on both execution time and memory usage at the cost of postprocessing [13]. An important remark about the timer-based calculations is that they are ultimately approximations, since the timestamps only approximate the linearization points. Leading to approximate relaxation error calculations.

2.3 Related work

As seen in the previous sections, there are plenty of existing methods to calculate rank errors. However, this broad selection of methods imposes several problems:

1) As seen with Haas et. al. [14] and Kirsch et. al. [16], different authors define rank errors differently, making results difficult to compare or even incomparable.
2) Although the idea of rank errors is discussed throughout several papers, very few describe in detail how the computation is performed in practice. Consequently, the overhead and additional processing needed to compute said errors is also left out [14], [15], [6], [8], [7], [16].
3) Although statistics on rank errors provide a meaningful insight into a relaxed data structure and how it operates, the complexity of computing these errors is important to cover to concretely argue for and against existing methods.

As mentioned, most recent papers discussing rank errors for relaxed data structures do not explicitly cover algorithms used to compute said rank errors. One of the few papers that do is presented by Williams et al. for MultiQueue rank errors [9]. In this paper, the authors also utilize an augmented B+ tree to reduce the calculation complexity to $\mathcal{O}(n \log n)$. More specifically, the leaves of the tree are queued elements, and a path from the root to a leaf consists of augmented nodes that contain additional information and, most importantly, a *delay-counter*, d_v, for each leaf. These counters measure *delay* as the number of elements with lower priority that are dequeued before any given element. The tree is used to replay a sequential history of operations on a MultiQueue, such that only the leaves of the tree represent current elements of the queue. Although this thesis explores similar prospects based on trees similar to the one of Williams et al., the results will not necessarily be comparable. This is because of the aforementioned *delay-counters* that result in the tree needing more space than necessary while also dealing with priority queues as opposed to FIFO queues.

3
Theory

This chapter aims to equip the reader with the necessary knowledge to understand the remainder of the thesis. It begins by discussing relaxation errors, including how they are measured. Secondly, a brief overview of the data structures utilized in the experiments is presented.

3.1 Rank errors

Semantically, a relaxed data structure makes mistakes compared to its non-relaxed version. These mistakes are called *relaxation-errors* and were formalized by Henzinger et al [5]. The authors develop a language to describe transitions between states of a relaxed data structure associated with a cost called rank error. This rank error is a measurement of how wrong the transition was, more precisely, it describes how many missing actions there are to make the transition valid in the non-relaxed version of the data structure.

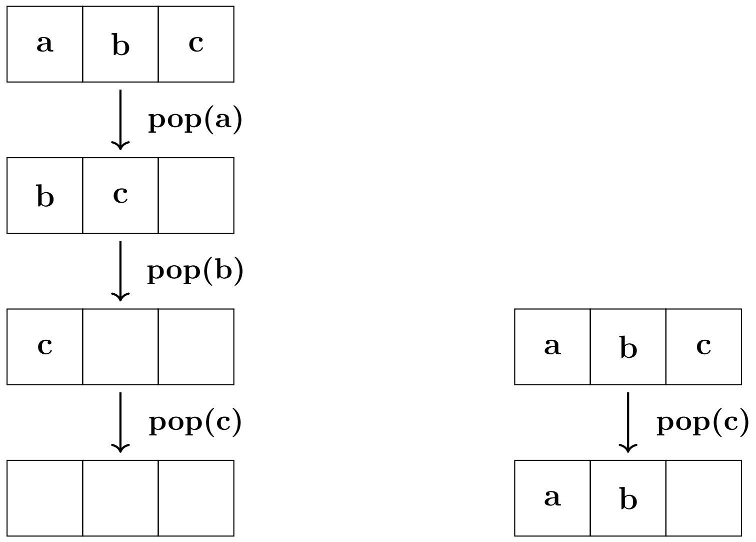
For example, consider a simple FIFO queue containing elements a, b, and c enqueued in that order. This means that a dequeue *should* return element a, but if the queue is relaxed, it *could* return, for example, c. This would result in a rank error of two since to legally dequeue item c, the queue first needs to dequeue a and b, i.e., two missing operations. Figure Figure 3.1b shows the states of the relaxed queue when dequeueing item c. Figure 3.1a shows all dequeues needed to dequeue item c legally, and one can see that when starting in the same states, the non-relaxed version has to do two more dequeues than the relaxed one, resulting in a rank error of two in the relaxed case.

3.1.1 Timestamp reordering

When calculating relaxation errors for FIFO queues using timestamps, the simplest approach is to replay all actions performed on the data structure in the order of their linearization points to find the rank of each dequeue.

An interesting observation is that for FIFO queues, one can reorder dequeues and enqueues operations performed on the queue, as long as two rules are followed: the internal ordering of enqueues and dequeues must be kept the same, and an element must be enqueued before it is dequeued. Internal ordering refers to the ordering of enqueues without respect to dequeues and vice versa.

The rank error for an item's dequeue is equivalent to how many other elements are enqueued before it and dequeued after it. This is the same as finding at what



(a) Non-relaxed dequeue of item c.

(b) Relaxed dequeue of item c.

Figure 3.1: Comparison of relaxed and non-relaxed dequeue of an item.

index, from the head, the dequeued item is. By this definition, there are two ways of altering the index of the item being dequeued:

1) Change the order it was enqueued; if one more item was enqueued before it, the index will be one higher, and so will the rank error.
2) Change the order it was dequeued; if it was dequeued earlier, there could be one more item in front of it at the time of the dequeue, increasing the rank error by one.

However, changing the order of an enqueue and a dequeue will never affect its index because enqueueing an item to the queue will never change the indices of items already in the queue. The second rule is a lot more trivial, dequeueing an item before it was enqueued to the queue makes no sense, but it makes a big difference in what reorders are legal. Now, one can be sure of the fact that we can reorder enqueues and dequeues, as long as the two rules are followed.

One of these legal reorders is moving all enqueues before all dequeues; this is the extreme case of using the rules, but after some reasoning, one can be convinced that this does not change the rank errors of any dequeues.

3.1.2 Rank errors as interval overlaps

For timer-based relaxation analysis, every element is associated with two timestamps representing the element's enqueue and dequeue times. These timestamps together form an interval representing an element's time spent in the queue. As aforementioned, an element's rank error in a FIFO queue is the number of other elements enqueued before but dequeued after a given element. In interval representation, given an element a in the set of all elements E, the rank error of a is defined as $|R(a)|$, using the definition in Equation 3.1.

$$\begin{aligned} R(a) &= \{\forall e \in E \mid C(e,a)\} \\ C(x,y) &= x.start < y.start \land y.end < x.end \end{aligned} \tag{3.1}$$

This perspective also opens up new possibilities for efficiently computing rank errors. This involves employing certain data structures, such as segment trees or prefix-sum trees, which can efficiently perform range queries over a set of intervals [17].

3.2 Rank delays

As can be seen in previous work, measuring rank errors is a popular way of visualizing the behavior of relaxed data structures [6], [16], [8]. However, Williams et al. [9] present another similar instrument related to their work on relaxed priority queues. Namely *delays*, which can be viewed as the opposite of rank errors. For relaxed queues, priority or not, rank errors represent the number of skipped elements when an element is dequeued. Delays, on the other hand, represent the number of elements that *skipped* another element [9].

Delays also hold a special relationship to the corresponding rank errors. When an element is dequeued with rank error e, it has skipped e elements, meaning that the skipped elements' delays will be incremented. Furthermore, this means that the sum of all delays is equal to the sum of all rank errors, while their distribution may be different [9]. This provides the quality of easily being able to verify calculation delays given a correct rank error calculation.

3.3 Relaxed FIFO queues

There are several different kinds of relaxed data structures, such as priority queues, FIFO queues, and LIFO queues, and for every kind, there are several different implementations. Throughout this thesis, two particular implementations of FIFO queues have been examined and used for benchmarking and analysis. These two are the 2Dd-Queue and the d-Choice Balanced Operations (d-CBO) queue.

3.3.1 2Dd-Queue

The 2Dd-Queue is a relaxed FIFO queue built with the 2D-framework proposed by Rukundo et al. [15]. The 2D-framework in general utilizes a disjoint set of lock-free concurrent FIFO queues called Michael-Scott queues [18]. Figure 3.2 shows an example of how the internal structure of a 2Dd-Queue might look like.

In Figure 3.2, the 2Dd-Queue has three underlying sub-queues named a, b, and c, which all contain some elements denoted as circles. The 2Dd-Queue uses two decoupled windows to orchestrate enqueues (red window) and dequeues (blue window). The windows span across the sub-queues (its width) and has a predetermined depth. When an item is enqueued, a thread will add the element to the first sub-queue it finds that has not already filled its slice of the window (c has filled its slice). Similarly, when dequeueing, a thread will remove an element from the first sub-queue

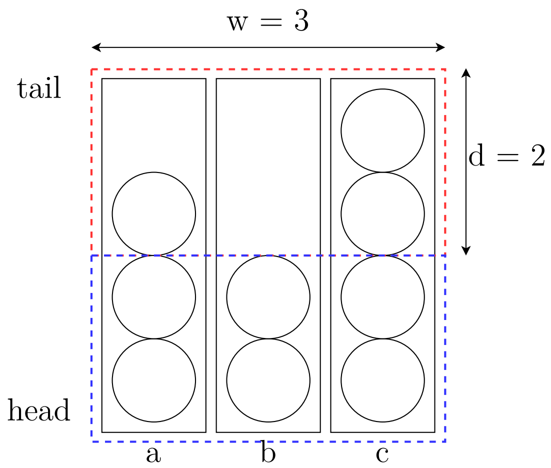


Figure 3.2: An example 2Dd-Queue instance with 3 sub-queues (a, b, and c) and a window depth of 2. Circles denote elements inside each sub-queue with the 2Dd-Queue growing from *head* to *tail*.

it finds that still has elements inside the current window. When the enqueue (dequeue) window is full (empty), it is shifted one window size further into the queue to accommodate new enqueues (dequeues) [15].

From Figure 3.2 it also becomes clear to see that the 2Dd-Queue is a *k-out-of-order* relaxed data structure. The rank error of a dequeued element is strictly bound by the size of the window. Since dequeued elements only originate from the dequeue window, the highest possible rank error r is bound by $r < w \cdot d$ where w and d is the window's width and depth, respectively.

3.3.2 d-CBO FAAArrayQueue

The d-Choice Balanced Operations Fetch And Add Array Queue (d-CBO-FAAAQ) is yet another relaxed FIFO queue [7], but it exhibits more random behavior compared to that of the 2Dd-Queue. The two main differences between the d-CBO-FAAAQ and the 2Dd-Queue are their choice of sub-queues and their load-balancing schemes.

The d-CBO-FAAAQ uses the performant FAAAQ [19] instead of the Michael-Scott Queue [11], [19]. This choice of sub-queue is not significant for this thesis, as it is the orchestration of the sub-queues that impacts the rank errors.

A more significant difference is that the d-CBO-FAAAQ uses a different load-balancing scheme, called d-Choice Balanced Operations (d-CBO). In both data structures, the sub-queues keep track of the number of elements they contain. The 2Dd-Queue uses this to determine when to shift the windows, but the d-CBO scheme uses this information to determine the optimal sub-queue in which the element should be enqueued to.

When an element is enqueued (dequeued) to (from) a d-CBO-queue, the queue samples d random sub-queues and chooses the sub-queue with the least (most) enqueues (dequeues) performed on it as the target. This has been shown to be more performant than the 2D-framework and generally produces lower rank errors [11].

Similarly, the maximum rank error r is also bound, but only probabilistically, by $\mathcal{O}(\frac{n \log \log n}{\log d})$.

3.4 Fenwick tree

A Fenwick tree is a *binary indexed tree* that stores its elements in a single array and traverses said array in a tree-like manner using bitwise manipulation [20]. The tree was initially proposed to aid data compression by storing cumulative frequencies of characters in an alphabet; one of its distinct traits is that the cumulative frequencies are stored as prefix sums in a logarithmic order [20].

Table 3.1 shows an example of which indices correspond to what ranges each element should contain the prefix sum for. The first element of the tree will never be used and stays zero, as this index only acts as an exit condition for the traversal logic, hence, the tree requires $n + 1$ entries for n elements. The prefix sums are orchestrated in a logarithmic order of base two, which creates the unique possibility of easily traversing the tree with bitwise manipulation. For instance, when an element is updated, its prefix sums are located by adding the index's LSSB (Least-Significant-Set-Bit) to itself. As an example, when the second element, at index one, is updated, the first prefix sum will be at index 2 since the LSSB of one is the first bit. Furthermore, the next prefix sum to update is at index four since two's LSSB is the second bit. This propagates upward until the next prefix sum index is outside the range of the tree, which means that all prefix sums have been correctly updated in $\mathcal{O}(\log n)$ time.

Table 3.1: An example Fenwick tree of 12 elements. The Index row represents the index into the backing array of the Fenwick tree, and the Prefix sum row represents the inclusive range for each prefix sum. The ranges are in notation $n..m$ where the prefix sum contains the sum of elements from n to m inclusive.

Index	0	1	2	3	4	5	6	7	8	9	10	11	12
Prefix sum	0	1	1..2	3	1..4	5	5..6	7	1..8	9	9..10	11	9..12

To query the tree for a sum at any given index, the aforementioned traversal is done oppositely. Instead of adding the LSSB of an index to itself, it is subtracted. For example, to query the prefix sum of the element at index 11, the tree will return the sum of elements at indices 11, 10, and 8. Essentially, the tree only needs to traverse as many elements as there are set bits in the index that the tree has been queried with. For instance, subtracting the LSSB of index 8, the new index will be 0, which means that all prefix sums have been added, and the tree may return the sum.

Trees that efficiently support these kinds of queries are not something unique. For instance, segment trees offer similar functionality and support efficient range querying as well [17], however, Fenwick trees offer two distinct advantages compared to segment trees. Fenwick trees exhibit a lower space complexity of $\mathcal{O}(n)$ compared to segment trees which require $\mathcal{O}(n \log n)$ storage, as well as a lower query complexity of $\mathcal{O}(\log n)$ compared to $\mathcal{O}(\log n + m)$ of segment trees [17]. The construction complexity for the two is equivalent, and both exhibit a complexity of $\mathcal{O}(n \log n)$, with the exception that Fenwick trees do not require any balancing during construction, which makes it faster in practice [17], [20].

3.5 Monte Carlo Methods

Monte Carlo methods are a class of algorithms that utilize randomness to improve or accelerate calculations [21]. These algorithms often use naturally or artificially generated objects to solve purely deterministic problems. In these cases, the Monte Carlo method involves random sampling from a probability distribution. The idea is to repeat the experiment many times to obtain a large number of points of interest to use the Law of Large Numbers [21]. One common application of these methods is estimation, where the goal is to approximate a numerical quantity related to a specific model. This technique often involves selecting a random subset of the model to gather insights about the entire system. Monte Carlo techniques are widely used to address numerous quantitative and deterministic problems across fields such as science, engineering, and finance [21].

4
Algorithms

This chapter presents a collection of algorithms developed as part of this work. Except for Section 4.6, which discusses LIFO rank and delay calculations, each section focuses on a distinct algorithm. The underlying intuition for each algorithm is described, followed by a complexity analysis that highlights performance constraints and computational bottlenecks. The explanations are structured to be comprehensive enough for readers to understand the corresponding source code implementations. To enhance consistency and clarity, Section Table 4.1 summarizes the notation used throughout the chapter.

Table 4.1: Table of symbols

Symbol	Meaning
n	Number of operations done on the queue
n_e	Number of enqueues
n_d	Number of dequeues
k	Mean rank error
m	Size of the queue during execution
b	Batch size
p	Number of available processors

4.1 List traversal replay

As previously stated, the simplest way of finding rank errors for a given trace is simply replaying the entire execution and finding the index in the queue of each element being dequeues. However, this will be severely slower than the monitored execution since the replay is done sequentially.

The extreme case reordering, where all enqueues are moved before any dequeue, was used by K. von Geijer[1] in his rank error calculations [11]. The list traversal algorithm is a C++ adaptation of that implementation. The intuition of this algorithm is that all elements are enqueued, then for each dequeue, the queue is traversed until the item to be dequeued is found, once found, it is removed from the queue. While traversing the queue, the index of the found item is tracked, as stated above, this index represents the rank error. When an item is found and its index is known, so is the rank error. This is done for each dequeue in order from first to last. Algorithm 1 shows a pseudo-code implementation of the proposed idea.

Algorithm 1 List Traversal Replay implementation

```
alias TimeStamp = int64
alias ItemId = int32
struct QOp = {time : TimeStamp, item : ItemId}
struct ListItem = {item : ItemId, next : ListItem}
procedure ListTraversalCalc(enqueues: [QOp], dequeues: [QOp])
    sumErr ← 0
    maxErr ← 0
    queue ← {{e.item, null} | e ∈ enqueues } : [ListItem]
    for i = 0 to |enqueues| − 1 do
        queue[i].next ← queue[i+1]
    head ← queue[0]
    for key in dequeues.items() do
        rank ← 0
        if head.item = key then
            head ← head.next
        else
            rank ← 1
            current ← head
            while current.next.item ≠ key do
                current ← current.next
                rank ← rank +1
            current.next ← current.next.next
        sumErr ← sumErr + rank
        maxErr ← MAX(maxErr, rank)
    return maxErr, sumErr
```

[1]The original implementation can be found at https://github.com/dcs-chalmers/semantic-relaxation-dcbo

When finding the time complexity of the algorithm, the mean rank error of all dequeues, k has to be taken into consideration. Preparing the queue can be done in $\mathcal{O}(n_e)$. Then for each dequeue, the algorithm traverses on average k steps into the queue, with n_d dequeues this results in a time complexity of $\mathcal{O}(n_e + n_d \cdot k)$ This clearly shows that this algorithm excels at low mean rank errors but grows linearly in respect to the mean error.

4.2 Batching dequeues

As with the previous algorithm, this is based on replaying the trace from the monitored execution while trying to perform fewer list traversals. This is done by batching together enqueues and dequeues. To create batches, all traces are replayed, performing all enqueues until there is a dequeue, then all dequeues until there is an enqueue will be called a batch.

Performing all enqueues until a dequeue is simple, each element is enqueued into the queue implementation. The difficult part is efficiently batching dequeues together to minimize list traversals.

The intuition used in this algorithm is that the queue can be traversed, from the front, until all items to be dequeued are found, and their rank errors can be calculated using the order in which they were found, together with the order they were dequeued. First, all items get a pop-order, the item dequeued first in the batch gets pop-order 0, incrementing with 1 for each subsequent item, resulting in the last item in the batch getting pop-order *b*-1, where *b* is the size of the batch. Then for each item, found at index *i* with pop-order *p*, the rank error for this item is *i*-*fp*, where *fp* is the number of items found with an index less than *i* and a pop-order less than *p*.

The implementation uses a map to keep track of the assigned pop-orders. A set is utilized for keeping track of pop-orders of found items during the traversal. Algorithm 2 provides pseudo code for dequeueing a batch while Algorithm 3 describes the full algorithm, utilizing the procedure defined in Algorithm 2

Algorithm 2 Implementation of dequeueing a batch

```
procedure DequeueBatch(queue, batch = {b_1, b_2, ..., b_n})
    sumErr ← 0
    maxErr ← 0
    popOrderMap ← Map from b_i to i
    foundPops ← ∅
    i ← 0
    while batch ≠ ∅ do
        curr ← queue[i]
        if curr ∈ batch then
            popOrder ← popOrderMap[curr]
            prevPops ← |{ p | ∀p ∈ foundPops,  p < popOrder }|
            rank ← i − prevPops
            sumErr ← sumErr + rank
            maxErr ← MAX(rank, maxErr)
            foundPops ← foundPops ∪ { popOrder }
            batch ← batch \ curr
            queue ← queue \ curr
        i ← i +1
    return queue, maxErr, sumErr
```

Algorithm 3 Implementation for creating batches of dequeues

```
procedure BatchingDequeues(enqueues: [QOp], dequeues: [QOp])
    nextEnqTime ← enqueues[0].time
    nextDeqTime ← dequeues[0].time
    deqIdx ← 0
    enqIdx ← 0
    sumErr ← 0
    maxErr ← 0
    queue ← ∅
    while all dequeues not processed do
        while enqIdx < |enqueues| ∧ nextEnqTime ≤ nextDeqTime do
            queue ← queue ∪ {enqueues[enqIdx].item}
            enqIdx ← enqIdx + 1
            nextEnqTime ← enqueues[enqIdx].time
        batch ← ∅
        while deqIdx < |dequeues| ∧ nextDeqTime < nextEnqTime do
            batch ← batch ∪ {dequeues[deqIdx].item}
            deqIdx ← deqIdx + 1
            nextDeqTime ← dequeues[deqIdx].time
        queue, err, max ← DequeueBatch(queue, batch)
        sumErr ← sumErr + err
        maxErr ← MAX(maxErr, max)
    return maxErr, sumErr
```

The algorithm can be hard to grasp, so an example is due. Assume one wants to dequeue items a and c from the queue below, in that order, creating a batch containing $\{a, c\}$, this would mean $batch = \{a, c\}$ and $queue = \{a, b, c, d\}$ at line 3.19.

a	b	c	d

When traversing the list, the first item encountered in the batch is $\boxed{\text{a}}$ at line 2.9. This item is at index 0, with pop-order 0. Hence, the rank error, calculated at line 2.12, of this dequeue is 0. Then at line 2.15, 0 is added to the set *foundPops*. The next item encountered is $\boxed{\text{c}}$. This item is at index 2 in the queue, with pop-order 1. When querying the set, at line 2.11, for the order of 1, it will return 1 0, since the set contains the 0, added when a was found. Then the rank, at line 2.12, is now calculated to $2 - 1 = 1$, resulting in a rank of 1 for dequeueing c.

Calculating the time complexity for this algorithm is non-trivial. When batching together b dequeues with average rank error k, the first item can be assumed to be at index k, and the following item to be dequeued is on average at index $k+1$, this is since it also has an average rank error of k. This means to find b directly following dequeues, one needs to look, on average, $k + b$ steps into the queue. For each found element in the batch, the pop-order needs to be found, using an hashmap, this can be achieved in $\mathcal{O}(1)$ time. However, finding *fp* from the set *foundPops* can only be done in $\mathcal{O}(\log b)$ time. Updates to the map are done in $\mathcal{O}(1)$ time, whereas insertion into the set is done in $\mathcal{O}(\log b)$ time. This results in the total time complexity for a single batch being $\mathcal{O}((b \cdot \log b) + k)$, from this the complexity becomes $\mathcal{O}(\sum_{i=0}^{N} \log b_i + \frac{k}{b_i})$, where N is the total number of batches, b_i is the size of batch i.

One interesting thing to note here is that this algorithm has no control over b, and it is changing with all batches. This makes its implementation slow and incredibly unpredictable, however, using the reordering strategy presented in Subsection 3.1.1, all enqueues can be done at once, offering the possibility to decide the batch size. Improving performance to make it not only more predictable but also significantly better. This is the idea explored in the next section.

4.3 Reordering for Batching

Combining the previous algorithms results in an implementation that utilizes both batching and reordering of operations. The previous algorithms are bounded by the number, and depth, of their list traversals. Specifically by $\mathcal{O}(n(\log b + \frac{k}{b}))$, then it is clear that the batch size that minimizes $\log b + \frac{k}{b}$ grows as k grows. It would be advantageous to choose the value of b freely instead of depending on the input to determine the sizes.

The algorithm divides the dequeues into $\frac{n_d}{b}$ batches, where n_d represents the number of dequeues and b signifies the batch size. For each batch, the procedure described in Algorithm 2 is used, the new algorithm is presented in Algorithm 4.

Algorithm 4 Implementation of reordering for batching

```
procedure BatchingDequeues(enqueues: [QOP], dequeues: [QOP], batchSize: int32)
    queue ← [e.item|e∈enqueues]
    sumErr ← 0
    maxErr ← 0
    deqIdx ← 0
    while not all dequeues handled do
        batch ← [dequeues[deqIdx + i].item | 0 ≤ i < batchSize]
        deqIdx ← deqIdx + batchSize
        queue, sum, max ← DequeueBatch(queue, batch)
        sumErr ← sumErr + sum
        maxErr ← MAX(maxErr, max)
    return maxErr, sumErr
```

As mentioned above, the implementation is very similar to the batching dequeues approach, with the main difference being the ability to set the batch size and thereby changing the complexity. The complexity then becomes $\mathcal{O}(\frac{n_d}{b}(b \cdot \log b + k))$ this can be simplified to $\mathcal{O}(n_d(\log b + \frac{k}{b}))$. Now one can see how the batch size affects the running time, however, at this stage, there is no magic number for the size of the batch to always minimize the complexity, since the complexity still depends on k, which can not be changed or known before the calculation is done.

4.4 Parallel batching

Since batches are computed independently, they can also be calculated in parallel. However, dividing a linear history into evenly-sized sub-problems is not trivial. In a linear history of a relaxed FIFO queue, elements enqueued in the period of one sub-problem may be dequeued in the span of another, meaning that all chunks in between also need to consider these elements. Figure 4.1 depicts an example history with intervals, where the intervals represent the time an element resides in the queue. Dividing this history into two sub-problems with the dashed line as the splitting point, it is clear that element 2 needs to be considered in both sub-problems, as it inflicts rank errors on elements in both sub-problems.

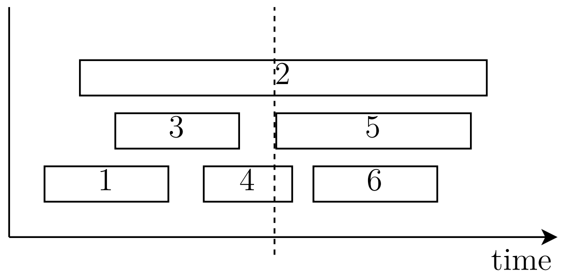


Figure 4.1: An example history visualized with intervals.

Two approaches to dividing the history into sub-problems are to either divide by time or by work. When dividing by time, the total time of the history, from the first to last operation, is divided into even periods. Then, elements are assigned to sub-problems matching their enqueueing timestamps with the periods of sub-problems. However, this approach can yield unevenly sized sub-problems as the linear history might not have a consistent frequency of operations.

The better alternative is to divide the history based on the amount of work, such that each sub-problem contains equal amounts of dequeue operations. It is favorable to divide based on the set of dequeues, since dequeues infer runtime complexity, as presented in Section 4.1. However, both approaches suffer from the problem of assigning overlapping elements to all sub-problems that need to consider them. This is cumbersome, as one has to iterate over all elements in all sub-problems, and then continuously assign them to later sub-problems that fully or partially overlap with the element's time in the queue. Furthermore, each sub-problem must have its list of intervals sorted on their end times (dequeue timestamps) for the batching process to work.

4.5 Replay Tree

One way to improve a replay algorithm's efficiency is to use a more efficient data structure with a more comprehensive preprocessing of the traces. To do this the definition of rank error has to be reworded. Instead of the rank error being the index in the queue of the dequeued item, it can be defined as follows: the rank error of an item being inserted in the queue is the number of items in the queue with a later dequeue time than the inserted item. An algorithm to answer the reworded question can be designed around events acting on the queue, where an event is either an *enqueue* or *dequeue* event, together with a red-black tree supporting order queries.

To calculate the errors of each item the algorithm iterates over a sorted list of events. When an *enqueue* event is handled the ordered tree is updated with the dequeue time of the enqueued item. The error is calculated using the order query to answer how many items in the queue have a later dequeue time than the inserted item. This number is exactly the rank error. Conversely, when a *dequeue* event is handled the dequeue time is removed from the ordered tree. Algorithm 5 presents one possible implementation using the idea of a replay tree.

Analyzing the complexity of the calculation part of the algorithm comes down to the complexity of the ordered tree. Insertion, deletion, and the query *order-of-key* are all done in $\mathcal{O}(\log t)$, where t is the size of the tree. Letting n be the number of events and m the maximum size of the queue results in a total complexity of $\mathcal{O}(n \log m)$. However, the total complexity of the algorithm, calculation together with preprocessing, results in a time complexity of $\mathcal{O}(n \log n)$, since this is the complexity of sorting the events, and one can assume that $n \geq m$ for any meaningful execution.

4.5.1 Using a Fenwick tree

Using a Fenwick tree reduces the preprocessing of the previous algorithm to preparing a list of dequeue-orders, ordered by their respective enqueue times. As seen in Algorithm 7, lines 7.1-7.5, this can be done without sorting, removing the limiting factor complexity factor of the Replay Tree.

Reversing the update and query traversals of the Fenwick tree (Algorithm 6) yields a tree now storing suffix sums. Replaying the dequeue-orders, updating the Fenwick tree at every step (line 7.18), results in a tree storing information about the number of elements in the queue (tree) dequeued after a given dequeue-order. This can be utilized to compute the rank error of an enqueued item (line 7.15).

Algorithm 7 presents an implementation using the Fenwick tree, while Algorithm 6 shows the reversed traversal patterns.

When analyzing the complexity of this algorithm, the biggest contributing factor is no longer the preprocessing, since the need for sorting is eliminated, and it can be done in linear time. Instead, the limiting factor is the tree updates and queries, done in $\mathcal{O}(\log n_d)$ time, one update and one query are performed for each dequeue. This results in a total complexity of $\mathcal{O}(n_e \log n_d)$, where n_e is the number of enqueues and n_d is the number of dequeues. Comparing this to the red-black tree solution it

Algorithm 5 Replay Tree Implementation

```
struct Event = {isEnq : bool, time : int64, deqTime : int64}
procedure CreateDeqMap(dequeues: [QOp])
    map ← empty HashMap
    for deq ∈ dequeues do
        map[deq.value] ← deq.time
    return map
procedure PrepareTreeInp(enqueues: [QOp], dequeues: [QOp])
    map ← CreateDeqMap(dequeues)
    enqEvents ← [{true, e.time, map[e.item] | e ∈ enqueues]
    deqEvents ← [{false, e.time, e.time} | e ∈ dequeues]
    events ← sort(enqEvents ∪ deqEvents)                ▷ Sorted on time
    return events
procedure ReplayTreeCalc(enqueues: [QOp], dequeues: [QOp])
    events ← PrepareTreeInp(enqueues, dequeues)
    tree ← empty Red-Black Tree
    sumErr ← 0
    maxErr ← 0
    constErr ← 0
    for e ∈ events do
        if e.deqTime = ∞ then                           ▷ If item is never dequeued
            constErr ← constErr +1
        else if e.isEnq then
            rank ← tree.indexOf(e.deTime) + constErr
            sumErr ← sumErr + rank
            maxErr ← MAX(maxErr, rank)
            tree.insert(e.deqTime)
        else
            tree.delete(e.deqTime)
    return maxErr, sumErr
```

Algorithm 6 Reversed Fenwick tree traversal

```
procedure LSSB(val : int64)
    lssb ← (val & −val)                                  ▷ Least-Significant-Set-Bit
    return lssb
procedure QueryRevFenwick(BIT : [int64], idx : int64)
    sum ← 0
    while idx < BIT.size() do
        sum ← sum + BIT[idx]
        idx ← idx + LSSB(idx)
    return sum
procedure UpdateRevFenwick(BIT: [int64], idx : int64, value int64)
    while idx > 0 do
        BIT[idx] ← BIT[idx] + value
        idx ← idx − LSSB(idx)
```

Algorithm 7 Fenwick tree based implementation

```
procedure PrepareInp(enqueues: [QOp], dequeues: [QOp])
    dequeues ← [{i+1, dequeues[i].item} | 0 ≤ i < dequeues.size()]
    deqMap ← CreateDeqMap(dequeues)
    deqOrders ← [deqMap[e.value] | e ∈ enqueues]
    return deqOrders
procedure FenwickRankCalc(enqueues: [QOp], dequeues: [QOp])
    deqOrders ← PrepareInp(enqueues, dequeues)
    sumErr ← 0
    maxErr ← 0
    constErr← 0
    for order ∈ deqOrders do
        if order = ∞ then                                ▷ If item is never dequeued
            constErr ← constErr + 1
            continue
        rank ← QueryRevFenwick(order) + constErr
        sumErr ← sumErr + rank
        maxErr ← MAX(rank, maxErr)
        UpdateRevFenwick(order, 1)
    return maxErr, sumErr
```

looks similar, but the main difference is that for the red-black tree solution, n is the number of events, which are roughly twice as many as the number of enqueues. From the aforementioned mitigated costs, the Fenwick-tree implementation significantly outperforms the red-black tree equivalent.

4.5.2 Parallelization

A good way of speeding up the calculation time would be dividing the input into independent sub-problems, where the errors could be calculated in parallel, using only enqueues in that sub-problem. A sub-problem can be defined as all enqueues happening inside a given time window. This sub-problem is independent if all elements enqueued before the time window starts are also dequeued before the time window starts. However, finding independent sub-problems for the base problem is not trivial and would negate any speedup from the parallelization.

Instead of giving each thread an independent sub-problem to solve, it can be assigned a time window to calculate all errors of enqueued items within it. The time windows are created by dividing all enqueues into chunks, where the first and last enqueues of the chunks define the time window. The thread can then iterate over all enqueues, keeping track of all enqueues with a dequeue time later than the start of the assigned time window. This makes sure that once the thread reaches its time window, it has access to all the information it needs to calculate the errors of the assigned time window.

When analyzing the complexity of this approach, the number of available threads, call it p, has to be considered. Furthermore, some threads will do significantly more work than others. More precisely, the thread assigned to the last time window has to look at all enqueues. This will be the limiting factor in the complexity. The number of time windows is p, and the number of enqueues in each window is then $\frac{n_e}{p}$. Then the complexity of calculating the last window is $\mathcal{O}(n_e - \frac{n_e}{p} + \frac{n_e}{p} \log \frac{n_e}{p})$. This complexity is slightly better than the sequential approach, and it scales well for the number of threads.

4.6 LIFO Rank and Delay calculations

Measuring the delays of elements in a relaxed queue is rather similar to measuring rank errors. Therefore, some previously discussed methods can be modified to compute delays instead of rank errors. The fact that rank errors in LIFO and FIFO queues are only defined by enqueue and dequeue times opens the possibility to modify FIFO rank calculations to fit LIFO calculations.

4.6.1 Delay Calculations

The ordered structure of the Fenwick implementation makes it straightforward to modify it to calculate delays instead of ranks. To achieve this, the tree should be adjusted to contain prefix sums rather than suffix sums. Additionally, reversing the order of the replay is necessary. This adjustment will effectively determine the delays for each element. Specifically, for each element, the tree will include only those enqueued after it, and a query will reveal how many of these elements were dequeued before the given element. This approach directly answers the question of delay for each element.

More specifically, for the Fenwick tree to contain prefix sums rather than suffix sums, the query and update traversal orders are swapped. Resulting in Algorithm 8. The only difference from the reversed traversal is lines 6.6, 6.8, 6.11, and 6.13. 6.6 and 6.11 swap places, and 6.8 and 6.13 also swap places. This change results in the tree containing prefix instead of suffix sums.

Algorithm 8 Fenwick tree traversal

```
procedure LSSB(val : int64)
    lssb ← (val & −val)                    ▷ Least-Significant-Set-Bit
    return lssb
procedure QueryFenwick(BIT : [int64], idx : int64)
    sum ← 0
    while idx > 0 do
        sum ← sum + BIT[idx]
        idx ← idx − LSSB(idx)
    return sum
procedure UpdateFenwick(BIT: [int64], idx : int64, value int64)
    while idx < BIT.size() do
        BIT[idx] ← BIT[idx] + value
        idx ← idx + LSSB(idx)
```

Three modifications have to be made in Algorithm 7 for it to calculate delays instead of rank errors. Reversing the looping order of line 7.11 makes sure the tree only contains elements enqueued after the currently handled element. The other two modifications are lines 7.16 and 7.19, where the Fenwick tree in Algorithm 8 has to be used instead of the reversed one used previously. These small modifications now allow for calculating the delays of a monitored execution, using the same traces used for calculating rank errors.

To verify and assess the performance of this implementation, a simple baseline algorithm was developed alongside it. This baseline draws inspiration from the Replay List Traversal implementation, described in Section 4.1. The key modification is that whenever an element is traversed that is not the currently dequeued item, its delay is increased. The delays are then summed up for each dequeued element. This means the *ListItem* struct in Algorithm 1 has to be modified to include a `delayCounter`; this counter would be increased for the head at line 1.18. Furthermore, the delay counter of `current.next` will be incremented before line 1.20, inside the while loop. Then, at line 1.23, the sum and max counters are updated in the same fashion as in the rank calculation case.

The sum, maximum, and average of the baseline implementation and the modified Fenwick implementation should all be identical. To further confirm the accuracy of these implementations, the rank error sum of each trace was also calculated. This value should match the sum of all delays.

4.6.2 LIFO Rank Errors

Rank errors for LIFO queues are slightly different from the ones for FIFO queues. As a reminder, the rank error for an element in a FIFO queue can be defined as the number of elements enqueued before it and dequeued after it. This means enqueue times only need to be compared with each other, and dequeue times with each other; this is what enables the possibility of reordering. For LIFO queues, however, the rank of an element is the number of items dequeued after it, and enqueued while it is in the queue. This requires comparisons between enqueue times and dequeue times, which effectively hinders any reordering of operations.

The baseline algorithm was developed based on the list traversal implementation, with a significant change being that all enqueue operations could not be performed during the preparation stage. As a result, this implementation functions as a replay implementation, where each operation is replayed and executed using a LIFO queue. The LIFO queue is implemented as a linked list, where each item is inserted at the head during an enqueue operation, and for a dequeue operation, the list is traversed until the desired item is found.

The baseline uses events to replay the traces, where an event is either an enqueue or dequeue, together with an ID and a timestamp. Then all events are sorted based on time and replayed in that order. When an enqueue event is replayed, the item is added to the linked list LIFO queue implementation. For a dequeue, the linked list is traversed until the item is found, and its index is the rank error. Algorithm 9 shows pseudo code implementing this idea.

The Replay Tree implementation is modified to keep track of all enqueue times of the items in the queue. This means that when a dequeue event is replayed, its rank can be calculated by querying the tree on the number of items in the queue with a later enqueue time. This number is exactly the rank error of the current dequeue. In Algorithm 5, this would mean events contain enqueue times instead of dequeue times. Trivially, the tree modifications (lines 5.26 and 5.28) and query (line 5.23) are made with the enqueue time of an event instead of the dequeue time. The if statement at line 5.20 can safely be removed, since the notion of constant error is

Algorithm 9 LIFO Rank Baseline implementation

```
struct LIFOEvent = {isEnq : bool, time : int64, item : int64}
procedure PrepareLIFOEvents(enqueues: [QOp], dequeues: [QOp])
    enqEvents ← [{true, e.time, e.item} | e ∈ enqueues]
    deqEvents ← [{true, e.time, e.item} | e ∈ dequeues]
    events ← sort(enqEvents ⋃ deqEvents)                    ▷ Sort on time
    return events
procedure LIFOBaseline(enqueues: [QOp], dequeues: [QOp])
    sumErr ← 0
    maxErr ← 0
    events ← PrepareLIFOEvents(enqueues, dequeues)
    queue ← [] : [ListItem]
    head ← {} : ListItem
    for event ∈ events do
        if event.isEnq then                          ▷ Add item at head of queue
            newHead ← {event.item, head}
            head ← newHead
        else                                               ▷ Find rank of dequeue
            key ← event.item
            rank ← 0
            if head.item = key then
                head ← head.next
            else
                rank ← 1
                current ← head
                while current.next.item ≠ key do
                    currents ← current.next
                    rank ← rank +1
                current.next ← current.next.next
            sumErr ← sumErr + rank
            maxErr ← MAX(maxErr, rank)
    return maxErr, sumErr
```

not present in the LIFO calculations.

A significant modification is necessary to implement the Fenwick solution for LIFO queues. The standard Fenwick implementation does not require comparing enqueue and dequeue times, so the approach needs to be adjusted. Instead of only iterating over the enqueue operations, all operations will be replayed in order.

For each enqueue operation, the enqueueing order of the enqueued item will be added to the tree (line 10.12). When a dequeue operation is performed, the item is first removed from the queue (line 10.14). Next, the tree is queried for the suffix sum of the enqueue order of the item (line 10.15). This query will return the number of items currently in the queue that were enqueued after the item that is being dequeued. The result will indicate the rank error associated with the dequeue operation.

Algorithm 10 Fenwick LIFO Calculation

```
10.1:  struct FenwickLIFOEvent = {isEnq : bool, enqOrder : int64, time : int64}
10.2:  procedure PrepareLIFOInp(enqueues: [QOp], dequeues: [QOp])
10.3:      dequeues ← [{false, enqueues.indexOf(e.item) + 1, e.time}|e ∈ dequeues]
10.4:      enqueues ← [{false, i+1, enqueues[i].time}|0 ≤ i < enqueues.size()]
10.5:      events ← sort(enqEvents ⋃ deqEvents)                ▷ Sort on time
10.6:      return events
10.7:  procedure FenwickRankCalc(enqueues: [QOp], dequeues: [QOp])
10.8:      events ← PrepareLIFOInp(enqueues, dequeues)
10.9:      sumErr ← 0
10.10:     maxErr ← 0
10.11:     for event ∈ events do
10.12:         if event.isEnq then UpdateRevFenwick(elem.enqOrder, 1)
10.13:         else
10.14:             UpdateRevFenwick(elem.enqOrder, -1)
10.15:             rank ← QueryRevFenwick(order)
10.16:             sumErr ← sumErr + rank
10.17:             maxErr ← max(rank, maxErr)
10.18:     return maxErr, sumErr
```

5

Approximation Algorithms

A problem with computing relaxation errors is that it may be computationally expensive or time-consuming for large data sets. Rank errors are useful instruments to understand how a relaxed data structure behaves, and may be important to consider when designing a new data structure. However, with the computation being time-consuming, this can lead to tests being left out and solely tested under smaller loads. In that case, it may be worth approximating the rank errors instead of computing them accurately.

Another aspect of approximations is that it may be useful for elastic data structures. For example, the elastic data structure SprayList uses contention between threads as a heuristic when deciding to adjust its degree of relaxation [6]. As contention only models the load degree, it does not necessarily represent the work done on the data structure. For this reason, it may prove beneficial to instead adjust depending on the currently approximated rank error. Implementing this in elastic data structures is out of scope for this thesis, but this section provides some example approaches to approximating rank errors for elastic FIFO queues.

5.1 First k elements

A starting point for approximating the rank errors is to accurately calculate errors for the first k elements. This may prove beneficial for data structures that exhibit a uniform behavior, as the approximated mean error should stay the same across the rest of the elements. Since the first k errors are calculated accurately, any of the algorithms previously mentioned may be picked depending on the expected behavior of the data structure. For example, data structures with a smaller expected mean error may benefit from a naive or replay tree-based implementation. However, for non-uniform data structures, it is trivial that the mean error of the first k elements may not be very representative of the execution. These types of data structures may need another type of approximation that incorporates the unpredictability into the calculation.

5.2 Monte Carlo Estimation

One method for estimating relaxation errors is through Monte Carlo Estimation. This approach has the advantage of including the entire execution process in the calculations by randomly selecting k items. These selected items are then used to approximate the rank errors of all items. As a result, this method allows for a more accurate estimation of rank errors, particularly in non-uniform data structures.

The idea behind this approximation is to utilize these k items and apply any of the developed algorithms to them as if they were a complete input. Then, for each dequeue, the rank obtained by the algorithm is scaled by a factor of $\frac{n}{k}$ to account for all the missing items. This factor is used to show the weight of each item, basically saying that for each item inducing a rank error on the dequeue, there are $\frac{n}{k}$ more items in the entire dataset. In other words, each item in the queue represents $\frac{n}{k}$ items.

This approach is demonstrated in Algorithm 11, where k elements are selected on lines 11.9 and 11.10. The selections have to be sampled from the set of enqueues specifically. Sampling from the dequeue set will skew the approximation downward, as elements that are enqueued and not dequeued (counting toward constant rank errors) can never be selected.

In this example, the Fenwick implementation, described in Subsection 4.5.1, is used to calculate the errors of the selected items. Then at lines 11.14 and 11.15 the values are scaled by $\frac{n}{k}$. Scaling the max and sum is the same as scaling all rank errors the algorithm found, since multiplication is commutative under both summation and the *max* operation. Two implementations utilizing this approach have been developed, using the Fenwick and Replay Tree approach, respectively.

Algorithm 11 Pseudo code for Monte Carlo approximation using Fenwick Tree

```
11.1:  procedure SelectRandom(set: [QOp], k: int64)
11.2:      returnSet ← ∅
11.3:      for i = 0 ; i < k ; ++i do
11.4:          r ← random(0, set.size() - 1)
11.5:          returnSet[i] ← set[r]
11.6:      return returnSet
11.7:  procedure MonteApprox(enqueues: [QOp], dequeues: [QOp], k : int64)
11.8:      n ← dequeues.size()
11.9:      randEnqs ← SelectRandom(enqueueus, k)
11.10:     randDeqs ← [ d | d ∈ dequeues if d.value ∈ randEnqs.values()]
11.11:     sort(randEnqs)                                    ▷ Sorted on time
11.12:     sort(randDeqs)
11.13:     maxErr, sumErr ← FenwickRankCalc(randEnqs, randDeqs)
11.14:     maxErr ← maxErr · n/k
11.15:     sumErr ← sumErr · n/k
11.16:     return maxErr, sumErr
```

The benefit of this algorithm is the lowered calculation time, since n is scaled down to k. However, picking k items from the entire dataset requires iterating over all enqueues and dequeues in order to find the pairs of timestamps connected to the chosen items. This is the bottleneck of any applied algorithm, no matter the complexity of calculation versus preparation.

5.2.1 Tailoring an approximation for the 2Dd-Queue

For any good approximation computation, looking for trends or patterns in a dataset can be beneficial. As previously discussed, the 2Dd-Queue exhibits uniform behavior, resulting in a pattern in its generated histories. Figure 5.1 shows an example history where the windows in the 2Dd-Queue result in a pattern of boxes of grouped elements. For reference, rank errors come from elements enqueued prior to and dequeued after a given element. This means that in Figure 5.1 an element's rank error is the number of elements lying above and left of itself. Furthermore, it also implies that the rank error of elements grows down to the right in a box, and most importantly, it makes every box independent of other boxes.

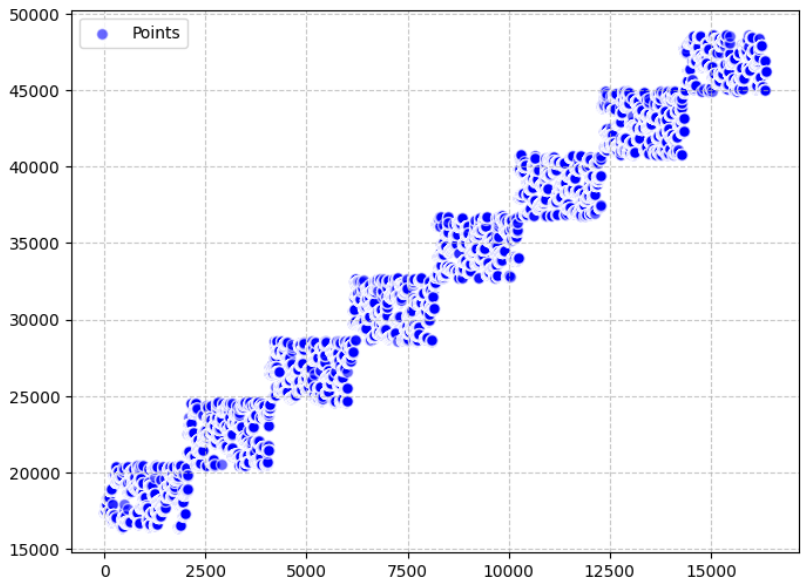


Figure 5.1: A typical 2Dd-Queue history generated during a stress test. Every point is an element with the horizontal and vertical axes representing the elements' enqueue and dequeue times, respectively.

This tailored approximation extends the k-first method by rounding k up to the nearest multiple m of the window size w, and chooses the $\frac{m}{w}$ middlemost boxes from the history. Choosing the centered boxes is desired in order to avoid any irregular leading or trailing behavior from a a stress test, as it may skew the results and not necessarily model the actual trends in the data structure. Since each box is independent of the other boxes, it suffices to copy a section of the original history as long as its span is aligned with multiples of the box size.

Algorithm 12 lines 1-9 show the process of retrieving the $\frac{m}{w}$ middlemost boxes. The mean and max rank errors can then be calculated by any accurate algorithm presented in Chapter 4. As the 2Dd-Queue exhibits uniform behavior, the distribution of elements inside the boxes tends to be the same across the total history. Therefore, all boxes exhibit rather similar mean and max errors, meaning that a single box might suffice as an accurate approximation of a whole history.

Algorithm 12 Pseudo code for tailored approximation

```
procedure GetBoxes(enqueues: [QOp], dequeues: [QOp], k: int64)
    w ← windowSize                          ▷ Window size is known at runtime
    kp ← ⌈k/w⌉ · w
    n ← (dequeues.size() - kp) / 2
    start ← n - (n mod w)
    end ← start + kp
    enq ← enqueues[start : end]
    deq ← dequeues[start : end]
    return enq, deq
procedure Tailored(enqueues: [QOp], dequeues: [QOp], k : int64)
    enq, deq ← GetBoxes(enqueues, dequeues, k)
    maxErr, sumErr ← FenwickRankCalc(enq, deq)
    return maxErr, sumErr
```

6

Experimental evaluation

This chapter aims to present the benchmarks and measurements of the algorithms previously mentioned. Before diving into the results, it is important to discuss the applied benchmarking methodology in this thesis and the consequences that follow. The benchmarking suite, together with all implemented algorithms can be found online [1].

6.1 Benchmarking methodology

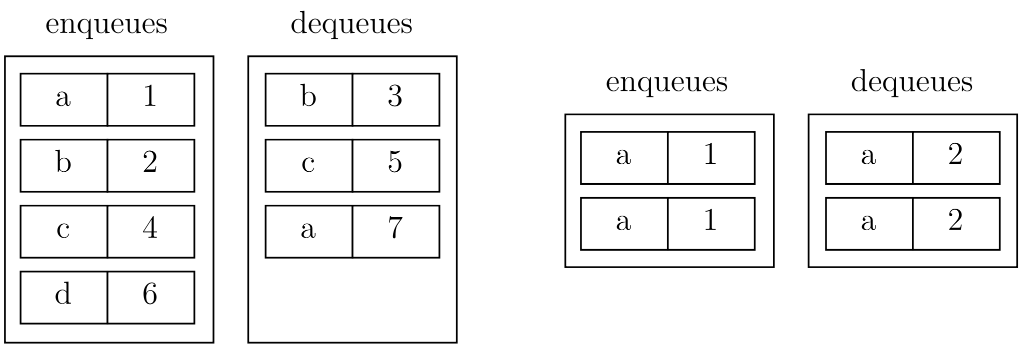
The histories to consume are given to each algorithm as a set of two lists, one for enqueues and one for dequeues. Every element of these lists contains a value and a timestamp, and a primitive example is shown by Figure 6.1a. This form of input was chosen by the fact that it is the expected input data by the baseline algorithm. In this thesis, histories are generated by different relaxed data structures subjected to stress tests in the same fashion as presented by von Geijer et al. [7], using 64 threads.

This thesis also imposes the necessary constraint of unique values and timestamps. Unique timestamps are needed to enforce a total ordering, since some algorithms utilize sorting, and the order of equal timestamps is arbitrary, leading to diverging results. However, unique values are necessary for correct results. Consider the history depicted in Figure 6.1b, where two duplicate values are enqueued by two different producers concurrently. When iterating over the dequeues, the order in which they are dequeued is uncertain. This history has both duplicate values and timestamps, but the same consequences follows for when only one of the fields has duplicates. As this problem does not directly affect the computation complexity, it is out of scope for this thesis and left out.

Each algorithm presented in these benchmarks consists of two stages, referred to as the *preparation* and *calculation* stages. The former involves restructuring the input data as well as building any auxiliary data structures, while the latter is solely for computing the errors. As the input data form directly suits the baseline algorithm, all algorithms presented in this thesis are immediately put at a disadvantage as they require the input data to be restructured before computation. Therefore, there is a point to emphasize the results of the calculation stages, as the relaxed data structures could produce other data forms that suit other algorithms better.

Each algorithm is benchmarked over three runs, measuring the execution times for both the preparation and calculation stages separately. The result is then taken

[1] https://github.com/relaxed-master-thesis/semantic-relaxation



(a) A typical history of enqueues and dequeues.

(b) An uncertain history with arbitrary order.

Figure 6.1: A set of two histories, with Figure 6.1a representing an example history used for benchmarking, while Figure 6.1b shows the importance of data constraints.

as the average execution time of each step, with the total time being the sum of the two averages. An important note is that each algorithm also undergoes a *cold run* before any measurements are made, to warm up the processor caches and minimize outlier runs.

6.2 Performance over increasing errors

As derived in Section 4.1, the limiting factor of the baseline algorithm's running time complexity is the mean rank error in the monitored execution. To contrast the presented algorithms in this thesis against the baseline, the benchmarks presented compare how each algorithm's throughput scales with the mean rank error. Throughputs are measured in dequeues per second, with the total set of dequeues being limited to 10^6 for each level of mean rank error.

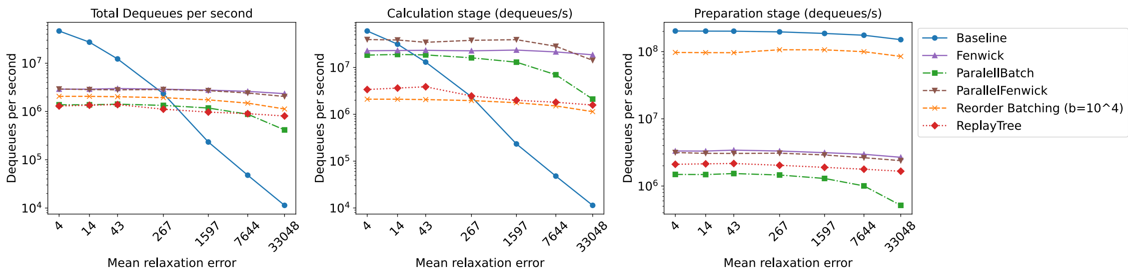
To create traces with increasing errors, the behaviors of the relaxed data structures are adjusted. As mentioned in Subsection 3.3.1, the relaxation errors for the 2Dd-Queue are bound by $(w-1)\cdot d$ with a mean generally around $\frac{w\cdot d}{4}$, which gives the possibility to create traces with increasing mean errors. The same goes for the d-CBO Queue, as stated in Subsection 3.3.2 it is probabilistically bound by $\mathcal{O}(\frac{n \log\log n}{\log d})$, and often generate mean errors of between $\frac{n}{2}$ and n.

As a reminder, Table 6.1 presents the complexities of the discussed accurate algorithms. Furthermore, Figure 6.2 shows two benchmarks comparing the throughput-to-mean-error relationship. When comparing this Table 6.1 to Figure 6.2, it is clear that only the Baseline and Reordering for Batching slow down as the mean rank error increases. However, the steps taken to limit this factor in the Reordering for Batching are very noticeable.

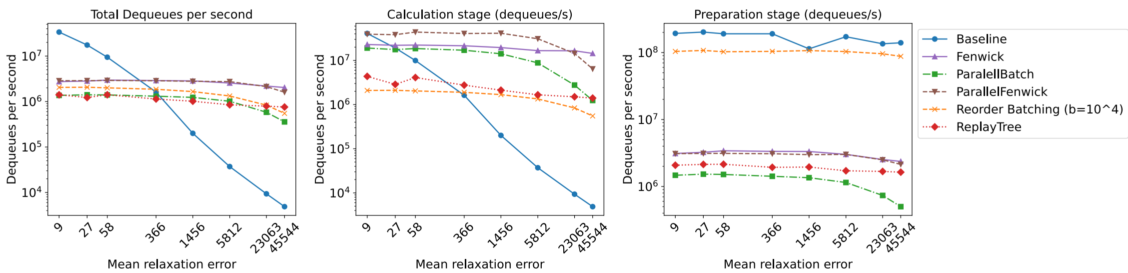
Algorithm	Complexity	Statement
Baseline (List Traversal)	$\mathcal{O}(n_e + n_d \cdot k)$	Section 4.1
Reordering for Batching	$\mathcal{O}(n_e + n_d(\log b + \frac{k}{b}))$	Section 4.3
Replay Tree	$\mathcal{O}(n \log n)$	Section 4.5
Fenwick	$\mathcal{O}(n_e \log n_d)$	Subsection 4.5.1
Parallel Fenwick	$\mathcal{O}(n_e - \frac{n_e}{p} + \frac{n_e}{p} \log \frac{n_e}{p})$	Subsection 4.5.2

Table 6.1: Runtime complexities of algorithms covered in Chapter 4.

Figure 6.2a is run using output produced by the 2Dd-Queue, while Figure 6.2b uses the d-CBO-FAAAQ queue. As postulated, the baseline algorithm experiences a linear decline as the mean error grows, while the remaining algorithms show a steady throughput. The smaller decline is most notable in *ParallelBatch*, which we attribute to the fact that as the mean error grows, there will be more overlap across subproblems. Furthermore, this means that as the overlap grows, each thread will do more and more wasted work when processing its subproblem. This is also a problem during the preparation stage, as it becomes more computationally expensive to split and assign subproblems.



(a) Throughput benchmark on data produced by the 2Dd-Queue.

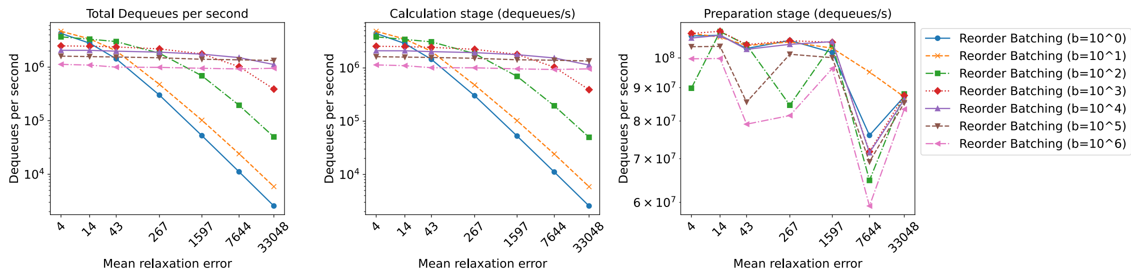


(b) Throughput benchmark on data produced by the d-CBO Queue.

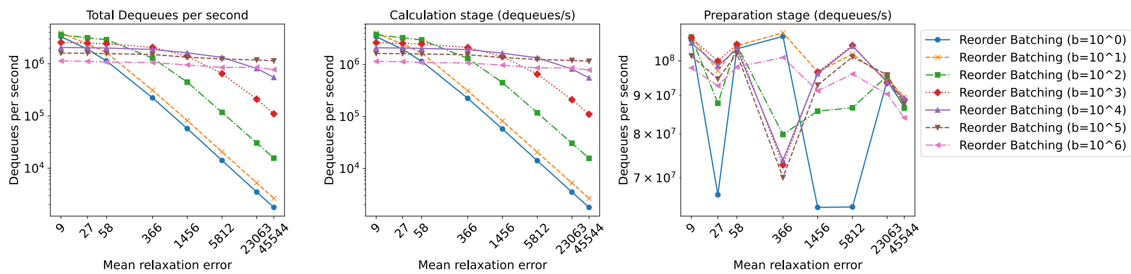
Figure 6.2: Throughput benchmark comparing all developed accurate algorithms.

In Figure 6.2b, the impact of mean rank error on the throughput of *Reorder Batching* is evident. This effect can be attributed to the fact that complexity increases linearly with the rank error, albeit at a much slower rate. Specifically, it grows b times slower, where $b = 10^4$. As a result, the slowdown is less noticeable in this benchmark. For clarification, batching implementations are named *Reorder Batching (b)*, where b is the batch size. The batch size used in the benchmarks is $b = 10^4$ since that gave the best performance over all tested rank errors. Figure 6.3

shows the throughput to mean rank relationship for several different batch sizes; these plots were used to select the batch size used in all presented benchmarks.



(a) Throughput benchmark on data produced by the 2Dd-Queue.



(b) Throughput benchmark on data produced by the d-CBO Queue.

Figure 6.3: Throughput benchmark comparing different batch sizes.

Figure 6.2 clearly shows the point where the parallel Fenwick implementation surpasses the baseline: for the total calculation, this happens around a mean rank error of 200. An interesting observation is that if one disregards the preparation stage, this number shrinks to around 8, making the parallel Fenwick implementations a viable replacement, no matter the mean rank error. The baseline could be said to have an unfair advantage when it comes to the preparation stage, because the output from the queues was made to work with the baseline implementation. Making it, disregarding the preparation time, a viable claim.

Furthermore, runs with lower rank errors perform significantly fewer operations in the same time window than runs with greater errors. This means that if an implementation is slow at low errors, this will be less noticeable, since the total number of operations is smaller overall. However, for high rank errors, a stress test of one second could generate 50 million operations: this, together with the lowered throughput of the baseline, makes the Fenwick implementations the clear winners across a bigger range of rank errors.

As shown in Figure 6.2, the parallel Fenwick implementation generally exhibits a higher throughput in the calculation stage, with the preparation stage being slightly more computationally heavy. However, as the ratio between preparation and calculation is roughly 10:1, this higher throughput is mitigated in the overall performance. Therefore, the non-parallel Fenwick implementation might be more attractive, considering it needs fewer resources to perform the same task at a similar pace.

The throughput of the Replay Tree decreases as the mean rank error increases. However, this decline is not due to an actual rise in rank error; rather, it is a result of the queue size adjustments made. As the relaxation error grows, the starting size of the queue is increased to ensure that the behavior of the queue remains unaffected. For the 2Dd-Queue, the initial size is set to $w \cdot h \cdot 2$, while for the d-CBO queue, it is set to $w \cdot 4$. Since the benchmarks are conducted as stress tests, the queue size remains relatively consistent throughout the run. Figure 6.4 shows how the throughput of the Replay Tree and Fenwick scales with queue size. This benchmark uses data from the 2Dd-Queue with $w = 16$, $l = 8$, where the initial queue size is displayed on the x-axis of the plot. Here, the behavior of the Replay Tree is the same as in Figure 6.2, even though the mean relaxation error is the same, conforming to the proposed complexity in Table 6.1.

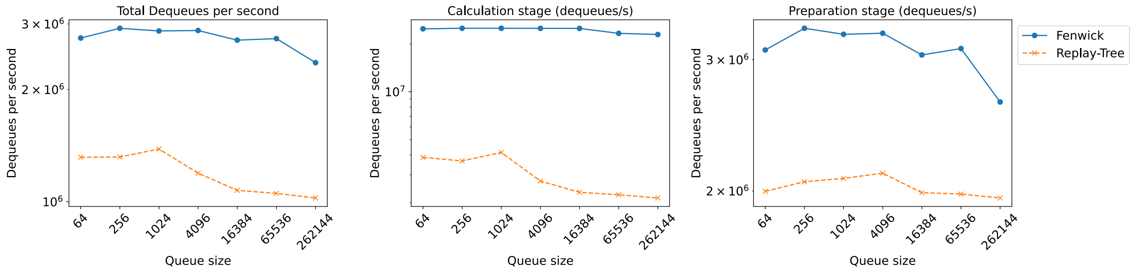


Figure 6.4: Benchmark comparing throughput to queue size for 2Dd-Queue

6.3 Variable workloads

According to the complexity analysis done on the different implementations, their behavior should stay relatively the same even if the type of workload is changed. An interesting workload to examine would be one where the amount of work changed during the execution. One such workload is a BFS (Breadth-First-Search) using the 2Dd-Queue. von Geijer et al. [7] provides this workload, producing a variable and realistic scenario. Figure 6.5 shows the throughput of the implementations when run on this data.

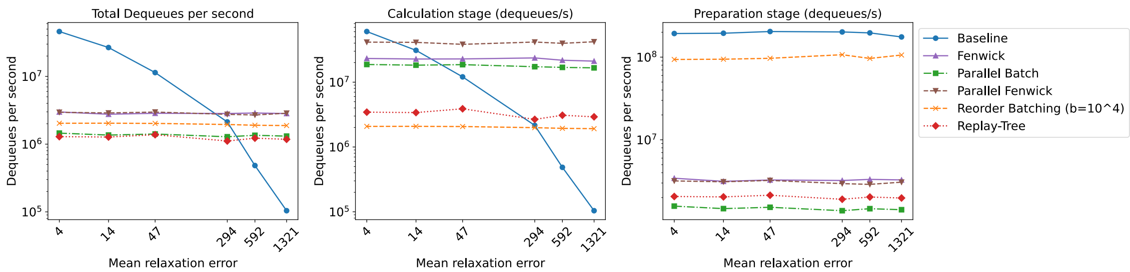


Figure 6.5: Throughput benchmark on data from BFS using the 2Dd-Queue

When analyzing the benchmark, it is important to note that the relaxation errors do not grow as large as the ones in the pure producer-consumer benchmark. We attribute this to the fact that the rank errors are, in this benchmark, limited by the queue size. However, none of the algorithms behave drastically differently, apart from the baseline. The baseline has about half the throughput compared to the stress test benchmark. Furthermore, in this benchmark, even though some runs have roughly the same mean relaxation errors as the stress test, the maximum error in the variable workload benchmark is significantly smaller. This means that the variance of the relaxation errors is smaller. This might be the reason the baseline performs worse, unfortunately, an extensive analysis of this behavior is out of scope for this thesis.

Besides the baseline, all other implementations have the same behavior patterns. The biggest difference is the decline in performance at the highest rank errors, as these might not grow as large in real-world applications as in the stress test benchmark. On the other hand, both the Replay Tree and Parallel Batch might experience slowdowns at these rank errors. If the application utilizes big queues, the Replay Tree will be significantly slower; furthermore, if there are large amounts of overlapping intervals, Parallel Batching would experience a higher percentage of wasted work.

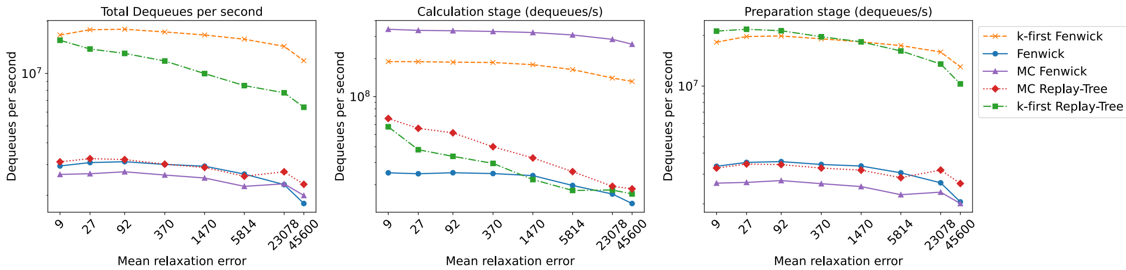
6.4 Approximation accuracy

When benchmarking the approximation algorithms, they all use 10% of the total input size. If the total number of dequeues is 10^6, the approximation algorithms will only calculate 10^5 rank errors. Figure 6.6 and Figure 6.7 show benchmark results from the developed approximation implementations, Figure 6.6a and Figure 6.7a depict the throughput over increasing relaxation errors for the implementations. Figure 6.6b and Figure 6.7b shows the error percentage over mean relaxation error.

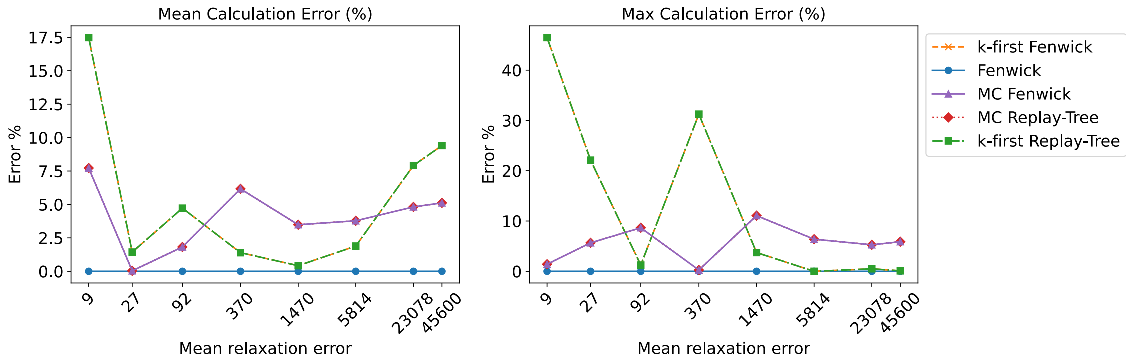
These implementations have throughput values between those of the Fenwick implementation, up to six times faster than the Fenwick as seen in Figure 6.6a and Figure 6.7a. However, this comes with one major drawback, namely differing results. As seen in Figure 6.6b and Figure 6.7b, the error percentages decrease as the errors increase. This is due to it being represented as a percentage and not a raw distance.

Looking closer at Figure 6.6, the pros and cons of a Monte-Carlo (MC) style approximation. In this case, the queue behaves randomly, which makes the results of a random approximation algorithm better, but it is also clear that it is the preparation that is the main obstacle for these types of approximations. The preparation throughput of the MC Fenwick and MC Replay Tree is as bad as the accurate Fenwick implementation. However, the calculation step is a lot faster than the accurate Fenwick. Furthermore in the calculation step it looks like the MC Fenwick is a lot faster than the k first Fenwick, however this is because at this input size, the calculation step for the k first approach takes about $4ms$, and the the MC Fenwick takes between 2 and $3ms$, meaning in practice there is not a huge difference.

When it comes to approximation accuracy, depicted in Figure 6.6b, the interesting observation is that the k-first approaches maximum error percentage is a lot higher than the MC approaches. We believe that this is due to the random nature of the d-CBO Queue and that the MC approaches are better at estimating its behavior through the calculations. However, as the error grows, the k first approaches are much better at approximating the maximum rank error.



(a) Throughput benchmark for approximation algorithms.

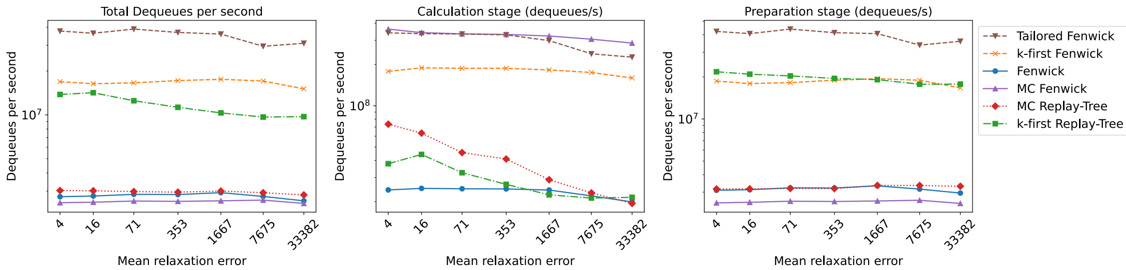


(b) Error percentage produced by approximation algorithms.

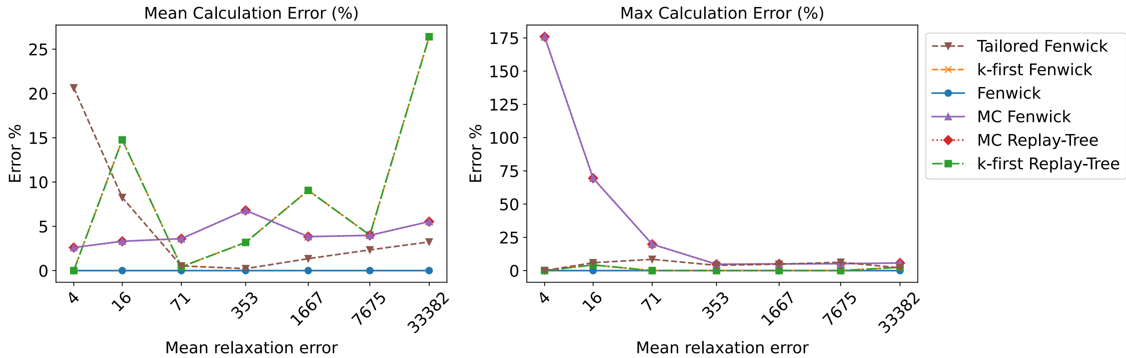
Figure 6.6: Throughput and Error percentage of developed approximation algorithms on the d-CBO-FAAAQ Queue

Figure 6.7 shows throughput and error percentages produced by approximation algorithms for a 2Dd-Queue history. The most interesting observation in this figure is how fast and accurate the tailored approximation is. In this case, the speedy preparation step for the tailored approximation makes it come out on top. The speedup in the preparation stage is attributed to the fact that the tailored approximation can eliminate unnecessary operations while having a more predictable execution path. For all other algorithms, the chosen set of dequeues to compute has to be matched with their corresponding enqueues. Since the enqueues can be in any order, the algorithms will, with high probability, iterate over more enqueues than there are dequeues. On the contrary, the tailored approximation utilizes the principle that all matching enqueues will be within a certain window in the list of enqueues, which can easily be singled out. Therefore, every enqueue iterated over will match a selected dequeue, which, in the implementations, means fewer branch-mispredictions and faster execution.

Figure 6.7b shows interesting results, especially the Mean Calculation Error (%) plot. Firstly, it is visible that as the window size grows, the Tailored approximation performs very well. Due to the very uneven behavior of the data structure, the k first approaches sometimes perform very well, sometimes not. We believe this happens when k is far from being a multiple of the window size. If a window is cut, the computed mean error will not match the actual mean, as discussed in Subsection 5.2.1. We believe this is more pronounced for larger means, as a larger mean error implies a larger window size, meaning a higher likelihood of splitting a window. Furthermore, it is clear that the mean calculated by the MC approaches is not perfect, but they are also never as bad as the k first approaches. For the Max Calculation (%) plot, it is clear that the MC approaches give absurdly bad errors for small window sizes.



(a) Throughput benchmark for approximating algorithms.



(b) Error percentage produced by approximation algorithms.

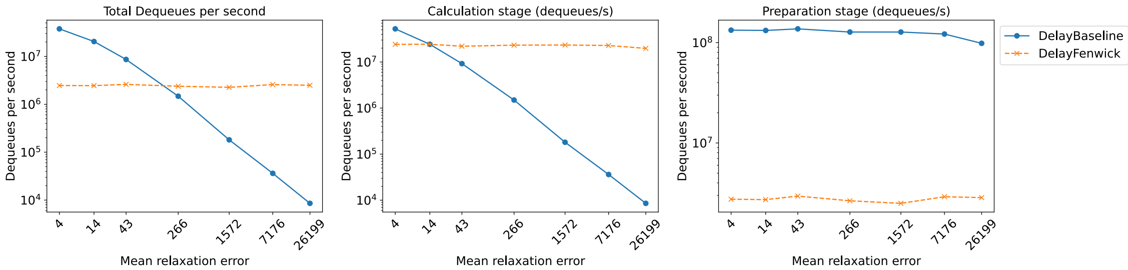
Figure 6.7: Throughput and Error percentage of developed approximation algorithms on the 2Dd-Queue

Interestingly, there is no clear winner among the approximation algorithms. On one hand, the MC approximations often give quite accurate results, with a fast calculation step, but the total time is the same as the accurate Fenwick implementation. While the k first approach is the fastest approximation paradigm, it can give inaccurate results, either due to heavily structured input, or because the queue is elastic, and behaves differently through the run. We also see that it could be viable to develop a tailored approximation to compute more accurate results even faster. This is not always possible, but using knowledge about the data structure to enhance approximations could sometimes be viable.

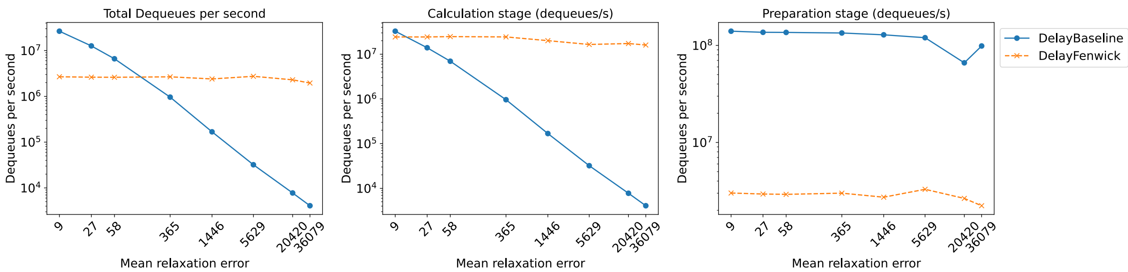
6.5 Performance in extended applications

Figure 6.8 shows the result of delay calculation benchmarks for the implementations discussed in Subsection 4.6.1, Figure 6.8a and Figure 6.8b use output from the 2Dd-Queue and d-CBO-FAAQ, respectively. The same trends visualized in Figure 6.2 can also be seen in Figure 6.8. In Figure 6.8, *DelayBaseline* is the mock baseline mimicking the behavior and complexity of the baseline algorithm described in Subsection 4.6.1. Hence, it also follows the same downward trend as the mean rank error increases. Similarly, *DelayFenwick* is an algorithm based on the Fenwick tree method presented in Subsection 4.5.1, but modified to instead compute the mean and max rank delays as described in Subsection 4.6.1. Due to both algorithms' similarities to their originating counterpart, the relationship between them in throughput also follows the same trend as in Figure 6.2.

The important takeaway from Figure 6.8 is not the increased throughput over the arbitrary baseline *DelayBaseline*. It is rather that a Fenwick tree-based algorithm can be modified to solve additional problems while still yielding similar performance gains. Furthermore, this also holds for examples modified to compute rank errors for relaxed LIFO queues (i.e., stacks) as shown in Figure 6.9.



(a) Throughput benchmark for delay calculating algorithms on input from the 2Dd-Queue.



(b) Throughput benchmark for delay calculating algorithms for the d-CBO-FAAAQ queue.

Figure 6.8: Benchmarks for delay calculating algorithms

Compared to Figure 6.8, Figure 6.9 instead quantifies the results of computing rank errors in histories produced by a relaxed LIFO queue. The *LIFOBaseline* algorithm is yet another arbitrary baseline, modified for LIFO rank errors in the same way as *DelayBaseline*. The *LIFOFenwick* implementation is the one described in Subsection 4.6.2. This benchmark also includes another algorithm, namely the *LIFOReplayTree*, which is also described in Subsection 4.6.2. The reason for including *LIFOReplayTree* is to show that the Fenwick tree's advantages over a red-black tree in this case still hold for other problem sets, which is not surprising. To conclude, Figure 6.9 describes the same relationship as Figure 6.2 and Figure 6.8, showing increased throughput and tolerance against trends in the mean rank error.

To further verify the behavior of the Fenwick and Replay Tree implementations,

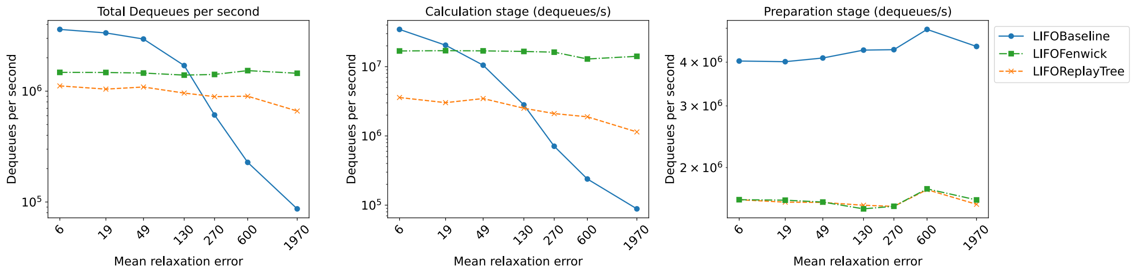


Figure 6.9: Benchmark computing rank errors of a relaxed LIFO queue.

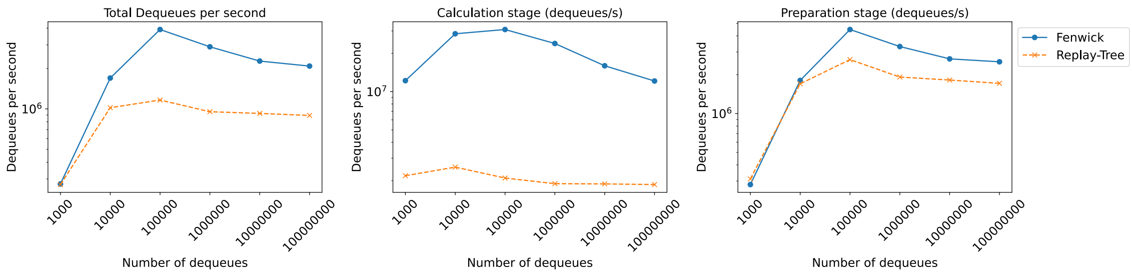


Figure 6.10: Benchmark comparing throughput to number of calculations.

one final benchmark is illustrated in Figure 6.10. This figure demonstrates how the throughput of these implementations is affected by the pure input size, defined as the number of rank errors calculated. For this benchmark, we utilized the 2Dd-Queue with parameters $w = 256$ and $l = 128$, ensuring that the mean error remains consistent. The purpose of this benchmark is to show that neither the Replay Tree nor the Fenwick implementation experiences any unexpected performance degradation as the input size increases. They follow their expected behavior, the Fenwick implementation slightly slowing down, in a logarithmic fashion due to the underlying complexity of the tree traversal, and the Replay Tree maintains a fairly steady throughput since the queue size is roughly the same across all runs. We attribute the low throughput of the smaller input sizes to the constant overhead of running the algorithms, taking up a greater portion of the total running time, effectively skewing the results.

7
Conclusion

The primary goal of this thesis was to design, construct, and analyze faster algorithms for calculating relaxation errors. As discussed in Chapter 6, the presented algorithms contribute to meaningful speedups against existing ones. In some cases, like the one depicted in Figure 6.2b, one developed algorithm is $3 \cdot 10^3$ times faster in the calculation step, and over $4 \cdot 10^2$ times faster in total. In practice, when calculating the errors for one trace, produced by the d-CBO queue when run for one second with the largest mean error, the baseline would run for about one hour; in contrast, the best-performing algorithm, in this case the Fenwick implementation, would take about eight seconds to finish.

It is also clear that the developed algorithms struggle to keep up with the baseline at the lowest errors. We attribute this to two main reasons, the first being the design of the input to the algorithms. The baseline was designed with the framework producing it, which led to some of the preparation being performed by the framework. However, by changing the design of the input, less preparation has to be done by the Fenwick implementation. While the framework does not have to do more work, it simply produces a different output. Secondly, the baseline is designed for low mean error calculations, while the developed algorithms are designed to perform well across all mean errors. Using the baseline could lead to higher mean errors not being explored by the researchers, since so much time has to be spent waiting for the calculations to finish.

This thesis also shows how the use of approximations can further enhance the performance of rank error calculations by, in the spirit of the relaxed data structure field, trading correctness for performance. While it is unclear to us whether it is viable in practice to approximate the behavior of relaxed data structures, it is still an interesting problem to tackle.

As mentioned, the secondary goal of this thesis was to explore how the presented algorithms would be extended and applied to other problems. More specifically, the problems mentioned were to calculate delay errors and rank errors for other relaxed data structures, such as LIFO queues. As shown in Section 4.6 and Section 6.5, the best performing implementations, the Fenwick and Replay tree implementations, can easily be extended while maintaining the performance increase over the baseline. These extendable implementations show promise for efficient algorithms for other relaxed data structures.

7.1 Future work

As discussed throughout this thesis, it is difficult to compute rank errors efficiently in parallel. However, the algorithms this thesis presents have shown other approaches to solving the same problem faster. This might create new possibilities and inspire future work on the presented parallel algorithms for further optimization and greater performance.

As previously described, Williams et al. [9] employ a B+ tree for computing rank and delay errors for priority queues. From the results of this thesis, it might be beneficial for performance to exchange the B+ tree for a modified Fenwick tree. If the Fenwick tree were modified to change its size dynamically, it could replace the B+ tree, making it suitable for rank error calculations for priority queues.

Lastly, this thesis hopes to enlighten others about the Fenwick tree data structure. As interpreted from the results presented in Chapter 6, a Fenwick tree significantly outperforms a red-black tree when used similarly. Hence, there might exist other algorithms and problems that could benefit from the Fenwick tree data structure, and it would be interesting to see how a Fenwick tree would impact performance in such applications.

Bibliography

[1] P. Tsigas, “Lock-free concurrent data structures and how to model their performance,” in *2019 19th International Conference on Application of Concurrency to System Design (ACSD)*, 2019, pp. 1–2. DOI: 10.1109/ACSD.2019.00004.

[2] M. Herlihy, N. Shavit, V. Luchangco, and M. Spear, *The art of multiprocessor programming*. Newnes, 2020.

[3] A. Atalar, P. Renaud-Goud, and P. Tsigas, “Analyzing the performance of lock-free data structures: A conflict-based model,” in *Distributed Computing: 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9, 2015, Proceedings 29*, Springer, 2015, pp. 341–355.

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

[5] T. A. Henzinger, C. M. Kirsch, H. Payer, A. Sezgin, and A. Sokolova, “Quantitative relaxation of concurrent data structures,” in *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2013, pp. 317–328.

[6] D. Alistarh, J. Kopinsky, J. Li, and N. Shavit, “The spraylist: A scalable relaxed priority queue,” in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2015, pp. 11–20.

[7] K. von Geijer and P. Tsigas, “How to relax instantly: Elastic relaxation of concurrent data structures,” in *European Conference on Parallel Processing*, Springer, 2024, pp. 119–133.

[8] 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.

[9] M. Williams, P. Sanders, and R. Dementiev, “Engineering multiqueues: Fast relaxed concurrent priority queues,” *arXiv preprint arXiv:2107.01350*, 2021.

[10] 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.

[11] K. von Geijer, P. Tsigas, E. Johansson, and S. Hermansson, “Balanced allocations over efficient queues: A fast relaxed fifo queue,” in *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, 2025, pp. 382–395.

[12] A. Morrison, “Scaling synchronization in multicore programs,” *Communications of the ACM*, vol. 59, no. 11, pp. 44–51, 2016.

[13] A. Haas, T. Hütter, C. M. Kirsch, M. Lippautz, M. Preishuber, and A. Sokolova, “Scal: A benchmarking suite for concurrent data structures,” in *Networked*

Systems: Third International Conference, NETYS 2015, Agadir, Morocco, May 13-15, 2015, Revised Selected Papers 3, Springer, 2015, pp. 1–14.

[14] C. M. Kirsch, H. Payer, H. Röck, and A. Sokolova, "Performance, scalability, and semantics of concurrent fifo queues," in *International Conference on Algorithms and Architectures for Parallel Processing*, Springer, 2012, pp. 273–287.

[15] A. Rukundo, A. Atalar, and P. Tsigas, "Relaxing concurrent data-structure semantics for increasing performance: A multi-structure 2d design framework," *arXiv preprint arXiv:1906.07105*, 2019.

[16] A. Haas, C. M. Kirsch, M. Lippautz, and H. Payer, "How fifo is your concurrent fifo queue?" In *Proceedings of the 2012 ACM workshop on Relaxing synchronization for multicore and manycore scalability*, 2012, pp. 1–8.

[17] M. De Berg, *Computational geometry: algorithms and applications*. Springer Science & Business Media, 2000.

[18] 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.

[19] P. Ramalhete and A. Correia, *Faaarrayqueue-mpmc lock-free queue*, 2016.

[20] P. M. Fenwick, "A new data structure for cumulative frequency tables," *Software: Practice and experience*, vol. 24, no. 3, pp. 327–336, 1994.

[21] D. P. Kroese, T. Brereton, T. Taimre, and Z. I. Botev, "Why the monte carlo method is so important today," *Wiley Interdisciplinary Reviews: Computational Statistics*, vol. 6, no. 6, pp. 386–392, 2014.

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

CHALMERS
UNIVERSITY OF TECHNOLOGY