CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Accurate Linearizations for Relaxed FIFO queues

Algorithms for Finding Linearizations of Minimal Relaxation Error

Master's thesis in Computer Science and Engineering

Ida Dahl
Hanna Schaff

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2025

Master's thesis 2025

Accurate Linearizations for Relaxed FIFO queues

Algorithms for Finding Linearizations of Minimal Relaxation Error

Ida Dahl
Hanna Schaff

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

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

Accurate Linearizations for Relaxed FIFO queues
Algorithms for Finding Linearizations of Minimal Relaxation Error
IDA DAHL
HANNA SCHAFF

Supervisor: Kåre von Geijer, Department of Computer Science and Engineering
Examiner: Philippas Tsigas, Department of Computer Science and Engineering

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

Typeset in LaTeX
Gothenburg, Sweden 2025

Accurate Linearizations for Relaxed FIFO queues
Algorithms for Finding Linearizations of Minimal Relaxation Error
IDA DAHL
HANNA SCHAFF
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

The increased use of multi-core processors in personal computers and servers has introduced the problem of time-consuming inter-thread communication during data exchanges. In response to this problem, data structures with a relaxed sequential specification are currently researched and developed. One type of relaxed data structure is out-of-order relaxed FIFO queues, which enable high throughput by containing multiple access points. This results in a FIFO queue which allows the order of operations to not be exactly first-in-first-out, resulting in a rank error for operations which are out-of-order. Rank error is an accuracy measurement for operations on relaxed queues and is often used when evaluating a newly developed relaxed queue. The current method of measuring rank error for operations on a queue is based on an approximation of when operations take effect. This may not exactly represent how out-of-order the operations on a relaxed queue is.

In this thesis, we investigate if it is possible to measure a lower rank error for operations on relaxed queues than what the current method measures. We present three algorithms aiming to achieve this (LP, OA and IC) and test them on operations on the relaxed 2Dd and d-CBO queues. Algorithm IC measures up to 18% less rank error than the current method measures, for a relaxed queue. We conclude that our findings motivate further research on this topic and suggest that the development of an extensive framework of algorithms may provide a useful tool for researchers in the field of relaxed data structures.

Keywords: Computer science, relaxed data structures, FIFO queues, concurrency.

Acknowledgements

We want to thank our supervisor Kåre von Geijer for introducing us to the research topic and for his constant encouragement, enthusiasm and patience. We also want to thank our co-supervisor Philippas Tsigas for providing invaluable feedback on our thesis and teaching us how to approach research. Finally, we want to thank our friends with whom we spent the semester side-by-side — thank you for making some unforgettable memories with us.

Ida Dahl and Hanna Schaff, Gothenburg, 2025-09-25

Contents

List of Figures

List of Tables

1

Introduction

Software that better utilizes parallelism in hardware has become an increasingly important research topic. Personal computers and servers typically run applications which are heavy in data exchanges. Multi-core processors have been utilized for these computers for fifteen years, introducing the problem of time-consuming inter-thread communication during data exchanges [1]. In response to this problem, new types of concurrent data structures which better utilize hardware parallelism are being researched and developed.

Concurrent data structures ensure predictable behavior in a multi-threaded environment by preventing *race conditions*, which are problems caused when multiple processes access the same unprotected resource. The prevention includes controlling the access points of concurrent data structures to ensure mutual exclusion. A lock-based concurrent data structure requires that a process obtains a mutex lock before executing a certain operation on the data structure. However, this may cause one process to block many other processes from execution as they wait for the mutex lock to become available, causing contention. Alternative lock-free implementations [2], [3] have been developed, which ensure mutual exclusion and allow more throughput, resulting in a more efficient concurrent data structures.

To further alleviate contention and increase parallelism, a data structure may be relaxed [1]. A *relaxed data structure* either adheres to a weaker correctness condition or has a relaxed sequential specification [4]. These methods aim to increase the number of access points to the data structure, which decreases contention and potential wait times. For example, a relaxed stack may allow retrieval of any of the k topmost elements, as opposed to a *strict* stack which only allows retrieval of the topmost element [5].

Linearizability is a correctness condition often used for concurrent data structures. A *history* states when operations were performed on a data structure during an execution [6]. Intuitively, if any concurrent history of operations on a data structure can be mapped to some valid sequential history, then the data structure is linearizable [7]. In practice, identifying the theoretical *linearization points* (where operations take effect, such as the atomic operation call which inputs a value to memory) is one way to prove that a data structure is linearizable.

A *linearization* maps operations to linearization points in an execution. If two mutually independent operations overlap (in time) in a concurrent execution, their

linearization points are ambiguous. In that case, either ordering of the two operations provide a valid linearization. Thus, there may be many valid linearizations for the same execution [7].

Linearizability is used as a correctness condition for data structures with relaxed sequential specification as well. In this case, a linearization can be compared to the strict sequential specification to determine the *relaxation error* of operations in a specific execution. Relaxation error is used as an accuracy measurement when developing relaxed data structures. A common way [8]–[11] to linearize an execution is to save a timestamp close to its theoretical linearization point. The timestamp can never record the exact theoretical linearization point, since the thread cannot simultaneously execute timestamping and call the atomic operations which manipulate memory. Additionally, timestamping may be further affected by scheduling. Due to this and the fact that there may be many valid linearizations, there may be more accurate linearizations (with lower relaxation error) than what can be achieved by this current method.

Many authors [4], [8]–[10], [12] of relaxed data structures provide correctness proofs, showing that their data structures are linearizable, or satisfy some other weaker correctness condition designed for relaxed data structures. The focus in papers introducing relaxed data structure designs is typically on proving a theoretical worst case bound on the relaxation error. The measured total relaxation error is presented, but there is no attempt to improve it as the papers aim at introducing a data structure not improving a linearization.

In this thesis, we investigate if there exists more accurate linearizations (ones with lower relaxation error across all dequeue operations) than the current timestamping method provides. We develop new methods that take as input a history of operations on a concurrent data structure. From such a history, the methods' aim to produce a linearization where the summed relaxation error over all dequeue operations is minimized. If we can obtain significantly lower total relaxation error than the timestamping method does, then results may indicate that current evaluations of relaxed data structures are underestimating the quality of the relaxations.

Though this area is relevant for all types of relaxed data structures, we focus on relaxed *first-in-first-out* (FIFO) queues. We present two principle methods which, from a history of operations on a relaxed FIFO queue, produce a linearization with the aim to minimize relaxation error. One of these methods is a linear programming-based solution which models the problem mathematically and utilizes existing optimization tools. The other is a novel algorithm which orders operations in the linearization. We also present an algorithm for improving an existing linearization by re-ordering its operations.

Outline. In Chapter 2, we provide necessary background on the area and prerequisites necessary to understand our developed methods. Chapter 3 accounts for current research in the field. In Chapter 4, the problem is further elaborated on and our algorithms are described as to be re-creatable. We declare our findings in Chapter 5 and conclude our work in Chapter 6.

2

Background

In this chapter, we provide essential background on concepts which are central to the project. In Section 2.1 we describe histories and linearizability. Section 2.3 and Section 2.4 account for traditional and relaxed queues, including examples of each type. The relation between these concepts is mainly accounted for in the latter section. Section 2.5 explains core concepts of linear programming, to provide background on existing linear programming solvers.

2.1 Linearizability

A history is a record of all operations' invocation (start) and response (end) events during an execution. In a *sequential history*, operations are ordered sequentially — an operation that has started will end before any other operation starts. A sequential history is a total order of operations, where for any pair of operations it is possible to determine in which order they were executed. In practice, a single-threaded execution results in a sequential history. Figure 2.1 depicts a sequential history of operations on a FIFO queue (see Section 2.3), where the vertical lines for each interval represents when the invocation and response events occur. In the example, item a is enqueued first, then item b is enqueued, followed by the dequeueing of item a and b respectively.

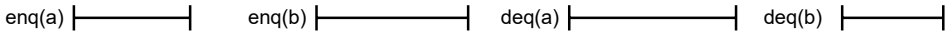

Figure 2.1: A sequential history of four operations on a FIFO queue.

In a *concurrent history*, the time intervals where operations are executed may overlap — an operation's start may be followed by another operation's start or end. This is a partial order of operations, where the only guarantee is that the order of operations executed by the same thread can be determined. Figure 2.2 depicts a concurrent history of operations on a FIFO queue. The vertical space between the operations indicate concurrent execution (on separate threads). A formal description of histories is provided by Herlihy and Wing [6].

As previously introduced, linearizability is a correctness condition often used for concurrent data structures. We provide an intuitive description of linearizability but refer to Herlihy and Wing [6] for the formal definition. The idea behind linearizability

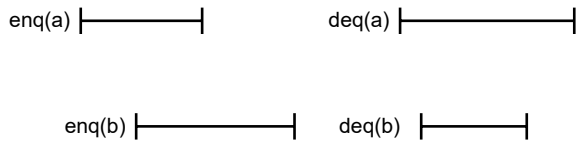

Figure 2.2: A concurrent history of four operations on a FIFO queue.

is that any concurrent history (of operations on a linearizable data structure) is equivalent to *some* sequential history [7]. Here, equivalence refers to the order in which operations take effect.

Linearizability corresponds to the idea that the partial order of any concurrent history (for a linearizable data structure) can be extended to a valid total order, and thus to a sequential history. The following two requirements must be fulfilled by the extension. Firstly, the order of non-overlapping operations must be preserved. Secondly, each operation must appear to take effect at some instant (called a *linearization point*), between its invocation and return. Special care must be taken to pending operations (operations which have not ended) and overlapping operations (for which the order is ambiguous) to ensure that the decided linearization points correspond to a valid final sequential history.

The concurrent example in Figure 2.2 can be interpreted as a sequential history by deciding the order of overlapping operations. Arbitrarily deciding that enq(b) takes effect before enq(a) implies that the first dequeue operation to take effect must be deq(b). Hence the corresponding sequential history would contain non-overlapping operations in the following order:

$$\text{enq}(b)\ \text{enq}(a)\ \text{deq}(b)\ \text{deq}(a).$$

If enq(a) would be decided as the first operation to take effect, then the corresponding sequential history would be in the order enq(a) enq(b) deq(a) deq(b). These sequential orders of operations are called *linearizations*. As this example shows, a concurrent history may correspond to several different sequential histories. Thus, there may be multiple valid linearizations to derive from a concurrent history of operations on a linearizable data structure [7].

In practice, identifying the theoretical linearization points where operations take effect is one way to prove that a data structure is linearizable. For example, for a queue's dequeue operation, this may be where the head field is updated (an item has been removed), or when it is (atomically) determined to be empty [7].

2.2 Timestamp approximation (TA)

A common way [8]–[11] to obtain a linearization from operations on a data structure is to call a timestamping function in the code, close to the theoretical linearization point of each operation. A theoretical linearization point may, for example, be

at an atomic operation which puts a value in memory. Since a thread cannot simultaneously call atomic operations and execute the timestamping function, the obtained timestamps are only approximations of the theoretical linearization points. Additionally, timestamping can be further affected by scheduling. However, such approximated linearizations are commonly used to compute relaxation error (see Section 2.4) as a quality measurement newly developed relaxed queues (see Section 2.4). This method is used as a benchmark against which our methods are compared and evaluated.

2.3 Concurrent FIFO queues

FIFO stands for first-in-first-out, this means that a FIFO queue conceptually works as we imagine a queue for the grocery store checkout. Achieving this requires implementing two functions, *enqueue* and *dequeue*. The enqueue operation adds an item to the data structure, and the dequeue operation removes and returns the oldest item from the data structure.

A concurrent (thread safe) FIFO queue also implements features to limit process access when multiple processes try to access it simultaneously. In a non-thread safe queue, the following example may happen: One process invokes a dequeue operation and another process completes a dequeue operation before the first process actually retrieves the item. Then the first process will receive an unexpected value (perhaps null). This is known as a race condition — when multiple processes try to access the same unprotected resource, causing a problem.

To avoid this, there are different methods for ensuring mutual exclusion. Based on how they implement this, thread safe queues can be divided into two main categories, blocking and non-blocking queues. Blocking means that the queue implements thread safety by way of limiting other processes' access, for example by using a lock to ensure that only one thread can access the head and tail of a queue at a time. One example of this is the lock-based MS-queue [2]. Non-blocking queues instead utilize atomic operations like Fetch-And-Add (FAA) [3], or Compare-And-Swap (CAS) to ensure that each operation performed on a queue results in a valid state [2].

Performance of a concurrent FIFO queue can be measured in throughput, which is in essence the number of times a queue's critical section can be executed per unit of time (on average). In lock-free queues with no explicit critical section this is instead defined as the amount of time to execute a successful atomic operation on the queue. Since only one process is able to operate on each end of the queue at a time, more processes operating on the queue leads to more contention and wait time to successfully complete such an operation [2].

2.3.1 FAAArrayQueue

The FAAArrayQueue is a lock-free FIFO queue consisting of a linked list of arrays to utilize cache locality. For the enqueue and dequeue operations, the queue utilizes

atomic FAA operations to fetch an index from the queue of where the atomic CAS operation should put or remove an item from. This design means that there is no contention in the final CAS operation since only one process will try to dequeue or enqueue on a specific index. Hence the slow CAS operation does not have to be redone due to failing as long as the fast FAA returns a valid index. This makes it faster than most queues, since the difference in speed between FAA and CAS means contention in the FAA operation causes less overhead. According to benchmarks, this queue is at least as fast as other strict concurrent FIFO queues [3].

2.4 Relaxed queues

To alleviate the contention caused by the mutual exclusion on few access points, relaxed data structures may be used instead of traditional (or strict) concurrent data structures [1]. A relaxed data structure either adheres to a weaker correctness condition or has a relaxed sequential specification [4]. Both methods aim to increase the number of access points to a data structure. Weakening the correctness condition entails replacing linearizability with a weaker condition such as sequential [13] or quiescent consistency [14]. Relaxing the sequential specification of a data structure means that linearizability is maintained according to the relaxed specification. An example is out-of-order relaxation [4], which allows ordering of operations on the data structure that is invalid compared to the original sequential specification.

There is a cost to relaxing the semantics. Applications which rely on a data structure's strict semantics to provide correct output have no use for relaxed data structures. However, relaxed data structures have been found useful in applications where a tradeoff can be made between increased throughput and decreased quality to achieve faster computation. For example, the MultiQueue is a relaxed priority queue which performs well in a parallel implementation of Dijkstra's algorithm to solve the single-source shortest path problem [10] and is state-of-the-art for the same problem applied to sparse graphs [15].

In this thesis we consider only out-of-order relaxation. Specifically, the thesis utilizes two out-of-order relaxed FIFO queues (see Section 2.4.1 and 2.4.2) for benchmarking. Such a queue may dequeue an item which is not the oldest in the queue. Some out-of-order relaxed FIFO queues (designs vary) dequeue one of the k oldest items, where k is defined through the construction of the data structure. Connecting this to linearizability, we note that a linearization derived from any history of operations on such a queue may not adhere to the strict semantics (where the oldest item is always removed first). Instead, an out-of-order relaxed FIFO queue is considered linearizable if it is possible to derive a linearization which adheres to the relaxed semantics, for example where one of the k oldest items is removed first [16].

Rank Error. Relaxation error, in general, measures how much a relaxed data structure violates the corresponding strict semantics. Rank error is a type of relaxation error and is used to measure how out-of-order an operation is on an out-of-order relaxed data structure. There are other measurements which are also used in this

context. To limit the objective of our algorithms, this thesis only uses rank error to measure relaxation error. For queues, rank error is defined for each dequeue operation in the history. The rank error of the dequeue operation on item x is the number of items which were enqueued before but dequeued after x, thus violating the semantics of a strict FIFO queue. Alternatively, it can be described as the number of items older than x in the queue at the time when x is dequeued [9].

2.4.1 2Dd-queue

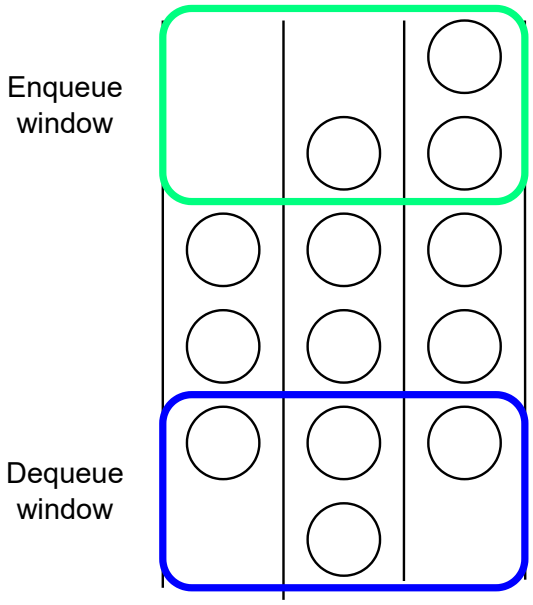

Figure 2.3: Illustration of a 2Dd-queue with enqueue window marked in green and dequeue window marked in blue. The *depth* of the windows is two and the *width* is three.

The 2Dd-queue is a relaxed FIFO queue consisting of several sub-queues. The relaxation of the 2Dd-queue is achieved by two two-dimensional windows that determine what portion of each sub-queue an enqueue or dequeue operation can be executed on. The first dimension of the window is the *width* and is determined by the number of sub-queues, the second dimension is the *depth* of the windows.

When a process dequeues an item from the queue, it first tries to find a valid sub-queue, that is, one with items in the window range, that is not currently operated on by another process. Once a valid sub-queue is found the oldest item in that queue is dequeued. If there is no valid sub-queue the window will instead shift up by *depth* indices until it is filled with items. The procedure for enqueueing is similar, the process tries to find a valid sub-queue with space within the window to enqueue an item. If there is no space in the window it will shift up *depth* indices, creating more space so that processes can continue to enqueue. By the definition of a valid sub-queue, optimally up to *width* processes could operate on the queue in parallel. By design there is a strict bound on the maximum rank error items can achieve, determined by the size of the window. That bound is $depth \times (width - 1)$ [5].

2.4.2 d-Choice Balanced Operations queue

The d-Choice Balanced Operations (d-CBO) queue is a relaxed queue consisting of n sub-queues. The algorithm for enqueue and dequeue entails randomly choosing d sub-queues and picking the one with the fewest of the respective operation, that is not currently being operated upon by some other thread, to execute the new operation on. This way of balancing operation allocation results in a higher throughput and lower rank error than similar queues. The rank error is proved to be bounded by $O(\frac{n\ log\ log\ n}{log\ d})$ with high probability. This design is one of the latest developments in the area of relaxed queues, and it outperforms previous designs [9].

2.5 Linear Programming

Linear programming is a method to determine a vector of variables such that it maximizes (or minimizes) a linear objective function dependent on those variables, while satisfying some set of constraints on the variables. The technique was developed around the time of World War II and had applications in scheduling and logistics (among other areas). Today, it is a relevant method of optimization within economy, computer science and other fields [17].

In Chapter 4, we present two methods which utilize linear programming by defining mathematical optimization problems aiming to obtain a linearization with minimized rank error across dequeue operations. Our methods apply existing solvers, and this section aims to provide basic understanding of these solvers, as well as resources for further reading.

2.5.1 A Linear Program

The general linear programming problem, or *linear program* (LP), can be formally defined as follows [17]:

$$\text{maximize } \mathbf{c}^T\mathbf{x} \text{ subject to } A\mathbf{x} \leq \mathbf{b}. \tag{2.1}$$

Here, $\mathbf{x} \in \mathbb{R}^n$ is the vector of variables to be determined. The given vector $\mathbf{c} \in \mathbb{R}^n$ decides the factor of each variable in the objective function through their dot product, since $\mathbf{c}^T\mathbf{x} = c_1x_1 + ... + c_nx_n$. $A\mathbf{x} \leq \mathbf{b}$ defines linear constraints. A is an $m \times n$ real matrix, where m is the number of constraints and n is the number of variables. The vector $\mathbf{b} \in \mathbb{R}^m$ is given. The constraints' relation $\leq$ is applied component-wise to the operator vectors of equal length.

A vector $\mathbf{x} \in \mathbb{R}^n$ that satisfies the constraints is a *feasible solution*. Any solution that results in the maximum value of $\mathbf{c}^T\mathbf{x}$ is an *optimal solution* (or *optimum*). Geometrically, the set of vectors satisfying all constraints form a *feasible region*, which is a polytope. If the linear program has an optimum, that solution is on the border of the polytope (global maximum in the direction of the objective function). There is no optimum in the case of an *unbounded* or *infeasible* linear program. There

are infinitely many optimal solutions if points on a side of the polytope are global maxima [17].

We proceed with a trivial example of a linear program:

$$\begin{array}{ll} \text{maximize} & 2 \cdot x_1 + x_2 + 2 \cdot x_3 \\ \text{subject to} & x_1 + x_2 \leq 1 \\ & x_2 + x_3 \leq 1 \\ & x_1, x_2, x_3 \geq 0. \end{array} \tag{2.2}$$

Here, $\mathbf{x} = (0, 1, 0)$ is a feasible solution, $\mathbf{x} = (1, 0, 1)$ is the optimum and $\mathbf{x} = (1, 1, 1)$ is an infeasible solution. This linear program can written be in the vector-and-matrix format by first defining: $\mathbf{c}^T = \begin{bmatrix} 2 \\ 1 \\ 2 \end{bmatrix}$ as a vector of the coefficients of the objective function, $A = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix}$ representing the variables' presence in each constraint and and $\mathbf{b} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$ defining the bound in each constraint. The program can be defined as:

$$max \begin{bmatrix} 2 \\ 1 \\ 2 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad \text{subject to} \quad \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \leq \begin{bmatrix} 1 \\ 1 \end{bmatrix}.$$

Any program can be defined in such way. Every equality constraint can be defined as two inequality constraints (for example, $x_1 + x_2 = 2$ is equivalent to the two inequality constraints $x_1 + x_2 \leq 2$ and $x_1 + x_2 \geq 2$). A function $min\ \mathbf{c}^T\mathbf{x}$ is equivalent to $max\ -\mathbf{c}^T\mathbf{x}$ [17].

2.5.2 Dual and Primal Linear Programs

The *dual* LP is a linear program which guards a regular linear program (in this context named *primal* LP) such that a feasible solution of the dual LP provides an upper bound on the maximum of the primal LP objective function [17].

To obtain an upper bound on the primal objective function, nonnegative coefficients are used to represent the primal constraints, which in turn can be formulated as a (dual) LP. We show this using the previous example (2.2). The desired form of inequality is

$$d_1x_1 + d_2x_2 + d_3x_3 \leq h$$

where d_1, d_2 and d_3 correspond to the coefficients in the primal objective function and must be at least as large (here 2, 1 and 2 respectively) and h is an unknown expression which should be minimized (to obtain a *good* upper bound). An inequality of this form can be achieved by first combining the inequalities of the primal constraints. Each variable's coefficient in each constraint is grouped together with new variable coefficients y_i. And each right hand side of the inequalities become a factor to a variable coefficient y_i. Applying this from to the example, we get:

$$(1 \cdot y_1) \cdot x_1 + (1 \cdot y_1 + 1 \cdot y_2) \cdot x_2 + (1 \cdot y_2) \cdot x_3 \leq 1 \cdot y_1 + 1 \cdot y_2 \tag{2.3}$$

We derive $d_1 = y_1$, $d_2 = y_1 + y_2$ and $d_3 = y_2$ and $h = y_1 + y_2$. Given the primal objective function in (2.2) and the inequality (2.3), the following dual constraints can be derived: $y_1 \geq 2$, $y_1 + y_2 \geq 1$ and $y_2 \geq 2$. Since $h = y_1 + y_2$ should be minimized, this can be formulated as a dual linear program for which any feasible solution provides an upper bound on the maximum of the objective function in (2.2).

The dual LP example:

$$\begin{array}{ll} \text{minimize} & y_1 + y_2 \\ \text{subject to} & y_1 \geq 2 \\ & y_1 + y_2 \geq 1 \\ & y_2 \geq 2 \end{array} \tag{2.4}$$

Given a general primal LP formulated as (2.5) the general dual LP is formulated as (2.6) [17].

$$\text{maximize } \mathbf{c}^T\mathbf{x} \text{ subject to } A\mathbf{x} \leq \mathbf{b} \text{ and } \mathbf{x} \geq \mathbf{0} \tag{2.5}$$

$$\text{minimize } \mathbf{b}^T\mathbf{y} \text{ subject to } A^T\mathbf{y} \geq \mathbf{c} \text{ and } \mathbf{y} \geq \mathbf{0} \tag{2.6}$$

2.5.3 Integer Linear Programming

Integer linear programming is similar to linear programming but requires that solutions are integral. An *integer linear program* (ILP), or simply *integer program* (IP), is in general a much more difficult problem than a linear program. The general integer program is defined as below, similar to the linear program but with the added constraint that each variable must be an integer value [17].

$$\text{maximize } \mathbf{c}^T\mathbf{x} \text{ subject to } A\mathbf{x} \leq \mathbf{b}, \mathbf{x} \in \mathbb{Z}^n \tag{2.7}$$

Integer programs are $\mathcal{NP}$-hard. This can be shown by a reduction from, e.g. the minimum vertex cover problem [17, p.37]. Minimum vertex cover is a well-known $\mathcal{NP}$-hard graph problem which consists of deciding the smallest set of vertices such that it includes at least one endpoint of every edge in the graph. In the reduction, the minimization of the sum of decided vertices is turned into an objective function. The definition that at least one vertex per edge must be chosen is modeled as constraints on variables, such that they must be nonnegative integers and any two variables representing ends of an edge must sum to at least one. We do not provide a full proof here.

MIPs. The class of problems which enforce integer constraints for *some* variables are called *mixed-integer programs* (MIPs). Naturally, techniques for solving ILPs can be used to solve MIPs as well.

LP relaxation. An *LP relaxation* of an ILP is constructed by allowing variables to take values between the integer values required by the ILP constraints. The LP relaxation can be used to find an approximate solution to the ILP, by using solving techniques for LPs on the LP relaxation and then rounding the solution to integers [17, p.33]. Note that relaxation here has separate meaning than the relaxation of data structures.

2.5.4 The Simplex Method

The simplex method is a method for solving LPs. It utilizes a *basic feasible solution* and a *simplex tableau* to iteratively increase the variables of the objective function until a maximum has been reached. This corresponds to iterating over the corners of a polytope defined by the constraints. We proceed with an overview of its mechanics based on the in-depth explanation provided in [17, p.57-79].

The tableau uses an equational form of LP where the inequalities of the LP constraints are turned into equalities by using *slack variables* (any inequality $x \leq a$ can be rewritten as $x + \sigma = a$, or $\sigma = a - x$, where σ is a slack variable). The tableau lists constraints in the $\sigma = a - x$ format and includes the objective function, there assigned to z. Initially, a basic feasible solution is derived from the existing constraints. In each iteration, an *entering variable* is chosen based on its positive contribution to the value of z. A *leaving variable* is chosen with the criteria that the entering variable should be able to be increased as much as possible. The entering variable is increased as much as constraints on the other variables allow, and the entering variable switches status with the leaving variable in all listed constraints and in z. Each iteration yields a feasible solution. Eventually, variables in z will all have negative coefficients, indicating that there is no variable that can be increased to increase the value of z. Then the solution is easily interpreted from inserting the variable values in the final feasible solution into z.

We provide a short example of this procedure using the earlier example (2.2). Slack variables $x_4 = 1 - x_1 - x_2$ and $x_5 = 1 - x_2 - x_3$ and the constraint $x_1, x_2, x_3 \geq 0$ provide a natural basic feasible solution $(0, 0, 0, 1, 1)$ which results in $z = 0$, the lowest possible value of z. The initial simplex tableau (2.8) reflects this.

$$\begin{array}{rrrrr} x_4 = & 1 & -x_1 & -x_2 & \\ x_5 = & 1 & & -x_2 & -x_3 \\ \hline z = & & 2x_1 & +x_2 & +2x_3 \end{array} \tag{2.8}$$

$$\begin{array}{rrrrr} x_1 = & 1 & -x_2 & & -x_4 \\ x_5 = & 1 & -x_2 & -x_3 & \\ \hline z = & 2 & -x_2 & +2x_3 & -2x_4 \end{array} \tag{2.9}$$

$$\begin{array}{rrrrr} x_1 = & 1 & -x_2 & -x_4 & \\ x_3 = & 1 & -x_2 & & -x_5 \\ \hline z = & 4 & -3x_2 & -2x_4 & -2x_5 \end{array} \tag{2.10}$$

In the first pivot step, x_1 and x_3 are both entering variable candidates, and x_1 is chosen arbitrarily. Consequently, x_4 is chosen as a leaving variable since it is the only

expression containing x_1. The expression $x_1 = 1 - x_2 - x_4$ replaces $x_4 = 1 - x_1 - x_2$ and in z, x_1 is replaced with $1 - x_2 - x_4$, resulting in the equivalent tableau (2.9). In the second pivot step, x_3 and x_5 are selected on the same grounds as x_1 and x_4 respectively. The exchange results in the third tableau (2.10), where we can easily interpret from the nonnegativity of x_2, x_4 and x_5 that the maximum value of z is 4 and the feasible solution of the simplex table is $(1, 0, 1, 0, 0)$. This aligns with our previous statement that $\mathbf{x} = (1, 0, 1)$ is the optimum to (2.2).

The simplex method can be illustrated geometrically. Each feasible solution determined in a pivot step represents a point on the border of the feasible region. Hence, in an illustration of the polytope, the feasible solutions determined by the simplex method can be plotted. Thus, its progress along the border, past corners and towards an optimum can be depicted. An example of such an illustration is provided in [17, p.60].

There are situations which require special care when using the simplex method. These include solving unbounded linear programs, handling degeneracy and handling infeasibility. Additionally, there is a risk that the simplex method will cycle, thus failing to find a solution. Detection and handling of these situations is accounted for in [17, p.61-65]

The dual simplex method. The *dual simplex method* solves the dual LP instead of the primal LP and it may be faster than the *primal simplex method* in some cases [17]. Implementation details differ from the primal simplex method. The primal simplex method preserves primal feasibility during the procedure and stops when dual feasibility has been established. Analogously, the dual simplex method maintains dual feasibility and stops when primal feasibility has been established. The dual simplex method is especially useful when solving MIPs and sparse LP problems [18].

2.5.5 Branch-and-cut

The *cutting plane algorithm* and the *branch-and-bound algorithm* are two methods which trim a feasible region, to make the the optimum easier to find. The former defines a tighter search area still containing all feasible solutions, while the latter removes non-optimal feasible solutions from its tree structure of all feasible solutions. The *branch-and-cut* technique is a combination of the two and is typically used to find the solution to an ILP or a MIP.

Branch-and-bound. The branch-and-bound algorithm removes solutions which are not the optimum, thus refining the feasible region. This algorithm utilizes a lower bound on the solution quality to remove bad solutions as soon as they are recognized as such, thus avoiding an exhaustive search of all solutions. *Branching* refers to the algorithm's traversal of a tree-like solution space, where each possible solution represents a branch. It maintains a lower bound on the solution quality (value of the objective function) and as it traverses each branch, it either updates

the bound with a new-found lowest value, or moves on from a branch as soon as its solution proves worse than the current bound (effectively "pruning" it) [19].

Cutting planes. The cutting plane algorithm is an iterative method for solving ILPs which is typically combined with other techniques. An intuitive explanation of this method relies on the geometric representation of the ILP. Similarly to the LP, the ILP has a geometric search area (a polytope) enclosed by the problem's constraints. Unlike the feasible region of the LP of which all points are feasible solutions, the ILP has only feasible integer solutions. These are found within the search area. The cutting plane algorithm iteratively adds valid constraints to the problem, thus removing (cutting) parts of the search area. The cutting is done such that all feasible solutions are maintained within or on the border of the search area. The goal is that the optimum will (after some iteration) become an extreme point in the resulting polytope, to enable finding it using the simplex method on the LP relaxation, which will be equivalent to the ILP in its solution [19].

Implementation of branch-and-cut. A general implementation of branch-and-cut [20] employs the branch-and-bound algorithm which solves the LP relaxation, prunes if necessary, partitions the problem into branches and iterates over each branch. As an intermediate step before pruning, a cutting plane algorithm can be employed to find cutting planes which violate the LP relaxation solution. If such a constraint is found, it is added to the LP relaxation which is then re-solved, thus avoiding unnecessary pruning and partitioning.

2.5.6 HiGHS and SCIP

HiGHS [21] and *SCIP* [22] are two computational solvers that can be used to solve linear programs and mixed-integer programs.

HiGHS. This LP solver implements two dual simplex solvers with different parallel strategies to achieve computational optimization. It is one of the first parallel implementations of the dual simplex method, as opposed to previous work which mostly focused on optimizing primal simplex solvers. The use of the dual simplex method makes it especially useful for solving large-scale, sparse LPs [21].

SCIP. This MIP solver utilizes cutting planes techniques, in addition to preprocessing algorithms and branching methods, to solve MIPs [22]. It is comparable to state-of-the-art solvers and is currently the fastest non-commercial solver for mixed integer programming [23].

3

Related works

Linearizability in and of itself is a well researched subject. There are many research studies on the topic of proving that certain data structures are linearizable.

In [24], Lowe creates a system for testing a history for a valid linearization rather than doing the heavier work of proving the linearizability of the data structure. On the formal proof side, Emmi and Ennea [25] use horn logic to create a specification of the data structure being tested and through satisfiability prove that a data structure is linearizable.

Research on linearizability of relaxed data structure is more focused on finding versions of linearizability that satisfy the relaxed semantics of the datastructure. For example [26] defines the concept of distributional linearizability. Similarly, the notion of quasi-linearizability exists to describe a quantitative relaxation of the normal linearization [27]. K-linearizability and regular relaxed linearizability are two other suggested correctness conditions for relaxed data structures [28].

As for techniques for finding a linearization to base relaxation error computations on, the method of taking a timestamp at the approximate linearization point in the code (see Section 2.2) is most commonly used [8]–[11] . There is also another method which entails encapsulating linearization points with global mutex locks and ordering the operations in the order of lock acquisitions. When comparing the order to the sequential execution order, completely accurate relaxation errors are retrieved. However, since the lock prohibits parallelism in the monitored execution, this is mainly useful when testing whether an implementation satisfies a worst-case bound [12].

4

Algorithms

This chapter includes formal descriptions of our proposed algorithms, relevant implementation details for re-creation, and complexity analyses. We present several methods for deciding a linearization from start and end timestamps of operations, including naive algorithms, two linear programming solutions and a novel ordering-based solution. We also present an algorithm for improving an existing linearization by re-ordering its operations. Our implementations of these algorithms are provided online [29].

We utilize the queue implementations and testing system (which generates a history) from [30], where the timestamp approximation method (see Section 2.2) is used. We have adapted the timestamping to use monotonic time, to remain robust against clock adjustments and to ensure that consecutive operation calls are always registered with increasing timestamps. Additionally, unique item identifiers were implemented for the items enqueued and dequeued in the history. Our implementation prints such timestamps (approximate linearization points) from an execution to a file, which is structured as the example in Figure 4.1. A linearization obtained in this way may be utilized by the algorithm presented in Section 4.5.

We limited our project scope to solutions which use start and end timestamps of each operation to determine some linearization point in between. These start and end timestamps are obtained as described above, but at a point close to each operation's start and end, rather than its theoretical linearization point. They are printed to a file which is structured as the example in Figure 4.2. All algorithms presented here take such a file as input. These utilize the item ID, operation type and start and end timestamps, whereas which thread executed an operation can be used for debugging purposes.

Performance metric. Our algorithms are designed to minimize mean rank error and, in some cases, maximum rank error. These are used as performance metrics in Chapter 5. The mean rank error is the average rank error over all dequeue operations in the resulting linearization. The maximum rank error is the highest rank error of any dequeue operation in the resulting linearization. The expected output of each algorithm is a linearization, for which mean and maximum rank error over all dequeue operations are computed.

Thread	Item ID	Operation	Timestamp
0	390158	GET	575813235130309
0	391696	GET	575813235134208
0	399110	GET	575813235140663
0	399366	GET	575813235141264
0	399622	GET	575813235141685
1	257	PUT	575813228402710
1	513	PUT	575813228403743
1	769	PUT	575813228404224
1	1025	PUT	575813228404655
1	1281	PUT	575813228405076

Figure 4.1: File structure for timestamps which approximate linearization points. This file is the output from our implementation of the timestamp approximation method (see Section 2.2). Note that this example contains ten selected operations from a much longer execution and does not represent a valid queue linearization.

Thread	Item ID	Operation	Start timestamp	End timestamp
0	390158	GET	575813235122681	575813235130690
0	391696	GET	575813235131712	575813235134549
0	399110	GET	575813235136573	575813235140994
0	399366	GET	575813235141134	575813235141425
0	399622	GET	575813235141555	575813235141846
1	257	PUT	575813228400746	575813228403282
1	513	PUT	575813228403522	575813228403923
1	769	PUT	575813228404084	575813228404394
1	1025	PUT	575813228404515	575813228404815
1	1281	PUT	575813228404936	575813228405246

Figure 4.2: File structure for timestamps obtained at the start and end of operations. Note that this example contains ten selected operations from a much longer execution.

Missing dequeue operations. An item may be enqueued without being dequeued in a valid history. The designs of Linear Program (Section 4.2), Integer Program (Section 4.3) and Interchange (Section 4.5) assume that all enqueued items have been dequeued in the history. We temporarily extend each input history with missing dequeue operations. In a pre-processing step, we add non-existing dequeue operations for which corresponding enqueue operations exist. Each of the dequeue operations have a single, distinct timestamp for both start and end, which is after all other operations' end timestamps. This simulates that they occur after the recorded history. The order of these operations is arbitrary. A post-processing step is implemented to remove the dequeue operations from the output linearization, ensuring that they do not contribute to the total rank error computation. In the Ordering Algorithm (see Section 4.4) missing dequeues are instead handled as a Null value and enqueues without a corresponding dequeue operation are handled as a separate case in the algorithm.

4.1 Multiple Probing (MP)

To start we created a naive algorithm that selects linearization points in linear time. The idea is to select a timestamp at the same relative distance between the start and end of each operation and compute the rank error. The algorithm runs for 25 iterations trying different distances and then returns mean, max and total rank error for the iteration with the best result in the selected parameter. For a more detailed explanation see Algorithm 1 (where either mean or maximum rank error is computed as described in Chapter 5).

The number of timestamps between start and end to try was determined empirically. The algorithm was ran with 100, 50, 25, and 20 iterations. The first three returned the same rank error while running only 20 iterations returned a worse rank error so for the rest of the tests 25 iterations were ran to decrease the amount of computation.

Our implementation includes functionality to plot all of the intermediate results, providing a visualization on how a later or earlier linearization point on average effects the rank error of the history.

4.1.1 Complexity and performance

The complexity of the algorithm is easily computed. For each iteration, we find a linearization in linear time and then run the quadratic time rank error computation on it. Hence, the complexity is $O(n + n^2)$ which simplifies to $O(n^2)$, where n is the number of items enqueued or dequeued in the input history.

The performance of the implementation is greatly dominated by the rather slow compute rank error function.

4.2 Linear Programming model (LPM)

For the first of our advanced methods, we approached the problem[1] as a mathematical minimization problem to enable modeling it as a linear program. Each variable corresponds to an operation's linearization point (as a timestamp). Constraints are naturally obtained from the given start and end timestamps of each operation, and the fact that an item must be enqueued before being dequeued. A linear objective function attempts to minimize the rank error. Our implementation utilizes an existing LP solver to obtain a solution. Due to the size of the input, our implementation processes and combines partial solutions.

As described in Section 2.5, a linear program can minimize a linear objective function $\mathbf{c}^T\mathbf{x}$ subject to constraints $A\mathbf{x} \geq \mathbf{b}$. Sections 4.2.1, 4.2.2 and 4.2.3 account for each part of our linear program. Finally, Section 4.2.4 briefly discusses our implementation.

[1] To find a linearization with small rank error, given start and end timestamps for each operation.

Algorithm 1: Multiple probing algorithm for 25 iterations

Data: H set of operations in a history
Result: *enqueues* a map from enqueue operations in H to timestamps (linearization points)
Result: *dequeues* a map from dequeue operations in H to timestamps (linearization points)

```
1.1  min ← ∞;
1.2  for i ∈ [0..25] do
1.3      (enqueues, dequeues) ← probe(H, i/25);
1.4      error ← compute_rank_error(enqueues, dequeues);  /* error is either
          mean or maximum rank error */
1.5      if error < min then
1.6          min ← error;
1.7  return (enqueues, dequeues)
```

Data: H set of operations in a history
Data: d relative distance from start to probe

```
1.8  Function probe is
1.9      enqueues ← dict;
1.10     dequeues ← dict;
1.11     for key ∈ input.keys() do
1.12         enq_time ← enq_start + (enq_end - enq_start) ∗ d;
1.13         if ∃ deq_start then
1.14             deq_time ← deq_start + (deq_end - deq_start) ∗ d;
1.15         enqueues.update(key: enq_time);
1.16         dequeues.update(key: deq_time);
1.17     return (enqueues, dequeues)
```

4.2.1 Variables

Each variable $x \in \mathbf{x}$ corresponds to an operation's linearization point (as a timestamp). The entire linearization is represented by $\mathbf{x}$, given that the item which has been enqueued or dequeued can be derived for each $x \in \mathbf{x}$.

4.2.2 Constraints

Constraints are defined by three categories of inequalities. Firstly, no variable can take a greater value than the corresponding operation's end timestamp. Secondly, no variable can take a lesser value than the corresponding operation's start timestamp. Finally, the linearization must reflect that dequeueing an item happens after enqueueing it. Hence, each variable representing the timestamp of a dequeue must take a greater value than the variable representing the timestamp of the corresponding enqueue. Using the same constraint notation as in Section 2.5, this is denoted

$$\begin{bmatrix} I \\ -I \\ A \end{bmatrix} \mathbf{x} \geq \begin{pmatrix} \mathbf{b} \\ -\mathbf{c} \\ \mathbf{d} \end{pmatrix}$$

where I is the identity matrix and $-I$ is the identity matrix where every positive component is negated. These identity matrices represent each variable's single appearance per constraint in the first and second category. A represents the two item-related variables' appearance per constraint in the third category. Specifically, it represents the subtraction of the enqueue timestamp from the dequeue timestamp, which must be positive. Hence at most two entries per row in A can be non-zero and the entry representing the enqueue timestamp is set to -1 and the entry representing dequeue the timestamp is set to 1. Additionally, the vector $\mathbf{b}$ contains all start timestamps, $\mathbf{c}$ contains all end timestamps and $\mathbf{d}$ contains only zeros (to ensure a later end timestamp than start timestamp).

4.2.3 Objective function

The objective function determines the quality of the solution produced by the linear program. Ideally, the objective function should exactly represent the rank error function. However, for a linear program this is not possible, since the rank error function is not linear (see Section 2.4) and linearity of the objective function is a requirement. Thus, the difficulty of this solution lies in defining a linear objective function which, when minimized, produces a linearization where dequeue operations have relatively small rank error. We defined one such objective function, but there may be other functions producing similar or better results. Our objective function attempts to *order* corresponding enqueue and dequeue points similarly in their respective histories. It does so by forcing corresponding operations' timestamps towards the same relative position within each operation history's time interval.

We say that the enqueue history contains start and end timestamps of all enqueue operations in the original history. The dequeue history is similarly defined. We let L_e denote the length of the enqueue history's time interval, i.e. the difference

between the first enqueue start timestamp and the last enqueue end timestamp. L_d similarly denotes the length of the dequeue history's time interval. The first enqueue and dequeue start timestamps are denoted s_e and s_d, respectively. The two equally large sets of variables $\mathbf{x}_e \subseteq \mathbf{x}$ and $\mathbf{x}_d \subset \mathbf{x}$ represent enqueue points and dequeue points in the linearization. Given that each pair x_{e_i} and x_{d_i} correspond to points of operations on the same item, the objective function is defined as:

$$min \sum_{\substack{x_{e_i} \in \mathbf{x}_e, \\ x_{d_i} \in \mathbf{x}_d}} |((x_{d_i} - s_d)/L_d) - ((x_{e_i} - s_e)/L_e)|$$

As variables are optimized according to this objective function and their constraints, corresponding enqueue and dequeue points should take timestamps with similar relative positions in their respective history's time intervals.

The reasoning behind the chosen objective function is based on the structure of a valid linearization for a history of operations on a strict FIFO queue. Consider that a linearization can be divided into two sets, one containing all enqueue points and one containing all dequeue points. Then, if corresponding enqueue points and dequeue points have the same order in their respective sets, the rank error will be zero. This is trivially derived — the item which is first enqueued is also first dequeued, the item which is second to be enqueued is also second to be dequeued, and so on. The objective function presented above does not concern exact orders, but attempts to position corresponding enqueue and dequeue points in similar orders by positioning them at similar relative positions within their respective history's time interval.

Since our objective function only considers corresponding enqueues and dequeues in relation to respective history length (not in relation to other enqueue and dequeue points), there is no guarantee that points will be ordered such that it minimizes rank error.

4.2.4 Implementation details

We implemented this solution in Python and utilized *CVXPY* [31], [32], a Python-embedded modeling language for convex optimization (which includes linear optimization). This approach was chosen since the CVXPY Python package provides an intuitive interface for developers to utilize existing solvers. The HiGHS [21] solver was used for this implementation since it performs especially well on large, sparse LPs. The aim is to process many variables at a time, and our constraint matrix is sparse for many variables (due to the identity matrices).

To process the entirety of a timestamp file and produce a linearization for it, our implementation solves partial problems which are, in a final step, combined. A constant c is defined in-code. In a pre-processing step, our implementation produces a map of items and their respective start and end timestamps per operation. Then a linear programming step decides the first c items' operations' linearization points. The linear programming step is repeated until all linearization points have been decided.

Additionally, we implement a pre-processing step to offset all timestamps by subtracting by the earliest timestamp. This shortens timestamp values and alleviates floating point-errors, which otherwise cause significant constraint violations in the decided timestamps.

4.3 Integer Programming model (IPM)

For this method, we use an integer program to exactly represent the rank error minimization problem on a history of operations on a strict FIFO queue. Each variable corresponds to the assignment of an order to an operation and constraints ensure that each operation is assigned exactly one order. A linear objective function minimizes the rank error. We utilize an existing IP solver to obtain a solution. We pursued this approach as a proof of concept, however we could not produce a functional implementation for a history of meaningful size. Thus, this method will only be briefly accounted for. Sections 4.3.1, 4.3.2 and 4.3.3 account for each part of our integer program. Finally, Section 4.3.4 briefly discusses our implementation.

4.3.1 Variables

Each variable states whether an operation has a certain order. Enqueue and dequeue operations are ordered separately. For example, in the resulting linearization, the first enqueue operation has been assigned order one and the first dequeue operation has been assigned order one. Orders can be converted into timestamps, thus assigning orders to all operations in a history is analogous to finding a linearization. Each overlapping operation in the history has multiple *plausible* orders which it may be assigned in a valid linearization. *Potential* orders are the plausible orders across all enqueue and dequeue operations, respectively. Plausible and potential orders are obtained through a conversion from start and end timestamps, where all overlapping intervals are interpreted as plausible orders (the reduction to orders is described in Section 4.4.1). We define one variable per operation and potential order. Since these variables are integer, they are decision variables stating whether an operation has a certain order or not.

4.3.2 Constraints

Constraints state whether an operation *can* be assigned a certain potential order and that they *must* be assigned some plausible order. If a potential order is not a plausible order for an operation, the corresponding variable is constrained to the value zero. The sum of the variables corresponding to all potential orders per some operation is one, stating that each operation must be assigned exactly one order. The sum of the variables corresponding to all operations per some order is also one, stating that each order must be assigned exactly once. Additional constraints which take start and end timestamps into consideration are required to ensure that enqueue operations occur before dequeue operations in the linearization. Similar constraints are required to ensure correctness in the conversion from orders to timestamps.

4.3.3 Objective function

We define two equally large sets of all assigned orders, $\mathbf{x}_e$ and $\mathbf{x}_d$, for enqueue and dequeue operations respectively. Given that the pair x_{e_i} and x_{d_i} correspond to operations on the same item, the objective function is defined as:

$$min \sum_{\substack{x_{e_i} \in \mathbf{x}_e, \\ x_{d_i} \in \mathbf{x}_d}} |x_{d_i} - x_{e_i}|$$

The reasoning behind this objective function is the same as described in Section 4.2.3. This objective function results in a linearization which is as in-order as possible, since every possible combination of operation orders is considered. Thus, for an input history of operations on a strict FIFO queue, this method provides the optimal solution (assuming good enough approximations of start and end timestamps). We do not formally argue that this objective function results in the optimal solution given a history of operations on a relaxed queue.

4.3.4 Implementation details

Our implementation of IPM implementation requires pre-processing to produce a list of plausible orders for each operation (as described in Section 4.4.1). To model the IP in Python, our implementation utilizes CVXPY [31], [32]. To solve the IP, we use the *SCIP Optimization Suite 3.2* [33] through its Python interface *PySCIPOpt* [34].

Experimentally, we observe that this implementation works for histories of very few operations (around twenty). Due to the large amount of variables in our IP model, memory issues may be encountered when running the program on larger problems. There may be more efficient IP models, perhaps ones where variables represent plausible orders per operation rather than potential orders (to vastly decrease the number of variables). Further, a partial solution method similar to that presented in Section 4.2.4 may be utilized. However, we did not properly implement constraints which consider start and end timestamps of operations. These would be necessary to ensure that orders can be converted to a valid timestamp linearization after combining partial solutions. Additionally, operations in early partial solutions may be assigned orders such that there is no available plausible order for some operation in a later partial solution. Our implementation does not consider such a scenario.

4.4 Ordering Algorithm (OA)

For this algorithm we changed the strategy from timestamps to feasible total orders. The basic idea is to find a total order for enqueue and dequeue operations, respectively, and then order them to be as similar as possible.

4.4.1 Ordering reduction

The base idea is that reducing the problem from timestamps to a total order could potentially reduce the complexity of the problem since the number of potential positions in a total order, which an operation could inhabit, should be fewer than the number of nanoseconds it lasts (the time granularity of our timestamps). We implemented a function that, given a history, computes the potential positions of all operations per type. This is computed by first dividing the operations into enqueues and dequeues. Then we, for each operation, compute the number of operations of the same type whose end timestamp is before the current operation's start timestamp, that is all operations that ended before a relevant timestamp. After that, all operations whose time intervals are overlapping the current operation's interval are computed. Then, for each operation, the earliest order possible is the number of operations that have finished before and the last possible order is the starting number plus the number of overlapping operations.

4.4.2 Algorithm description

The output from the ordering reduction is divided into several cases for prioritization. The cases are decided based on some simple calculations comparing the potential ordering positions of the enqueue and dequeue operation for each item. The cases are defined as follow:

- **One overlap:** only one value is the same in both sets of potential orders.
- **No overlap:** the sets of potential ordering values are disjunct.
- **No dequeue:** the items that are still in the queue at the end of the execution.
- **Long overlaps:** this case handles all the items with at least two overlapping potential orders.

In order to assign the enqueue and dequeue orders of the operations to be as close as possible, the next step is to assign these in some good order. The goal for each step of the algorithm is to try and assign the enqueue and dequeue operation of each item to the same or a close ordering index.

End enqueues. To start we look at a subset of operations that have the least amount of effect on the rank error, items with no dequeue that have the potential of being assigned after all enqueues that have a corresponding dequeue operation. These are put at the end of the enqueue order after the potential dequeue order indices.

One overlap. In the next step of the algorithm we look at the operations that share exactly one potential order. These are then assigned to that order in as great of an extent as possible. Any unassignable operations are dealt with later. This overlap case is handled first since there is only one potential position where they can share and ordering index, starting here means there is the least likely hood that, that specific index is already unavailable.

No overlap. Next we look at the split overlap items, try to enqueue them as close as possible. If one of the two closest indices are already unavailable, the algorithm iteratively tries one index further apart for that operation until it is free. Here again the reason that we handle this next is we know that these can not be assigned to the same index but we want the distance to be as small as possible without moving other operations at this point.

Long overlap. These items are sorted in ascending order by number of overlapping operations, to give the ones with fewer alternatives for overlapping indices a greater likelihood that one of the overlapping indices was still available. Each overlapping order is checked pairwise and assigned if available for both operations. If none of the overlapping order indices is available the item is handled by the **no overlap** procedure. This is handled after all operations that could be assigned to the same ordering index have been assigned.

Assigning the rest. After the first three steps of trying to optimize as many pairs of enqueues and dequeues as possible the strategy of the algorithm instead shifts to trying to achieve a valid output by ensuring that all items are assigned to some feasible ordering index. This is done by checking for empty ordering indices and selecting a potential unassigned item at random. If there are positions where all potential items are assigned to an index, one of those items is randomly unassigned from it's previous order index and assigned to the empty one. The previous, now empty, index is then handled in the same way recursively. The process is repeated until all operations have an assigned order index. The randomness in the last step is to avoid any potential infinite loops when reassigning items recursively.

Time complexity. The complexity of this algorithm is not entirely trivial to compute. We have several steps that run after one another most being quadratic complexity but running on an unknown portion of the total input. To calculate the worst case complexity we assume the case where the entire input would run through one of the steps which gives us a complexity of $O(n^2)$, where n is the number of items enqueued or dequeued in the input history.

4.4.3 Implementation details

Since the compute rank error function that is used for all algorithms takes only handled timestamps, we also implemented a function that took the output of the ordering algorithm and turned it into a timestamp dictionary in the format required for compute rank error. This revealed several issues with the algorithm and lead to a last step being added to check that all operations still have feasible timestamps, ones within the original timestamp interval. The implemented solution for this check worked for several smaller files that were used for testing purposes since the larger results files took an unfeasible large amount of time to run. Later it was however discovered that the implemented check only works in fixing the ordering of some files, most importantly not the ones used for the main results in this thesis, more on this in Section 5.4. The difficulty with turning the ordering result into a timestamp result

also meant that the result does not pass all tests for timestamp outputs, specifically several operations can have the same timestamp, this is however a case that can happen naturally when time stamping the approximate linearization point so it is not an issue.

4.5 Interchange (IC)

The Interchange algorithm iteratively improves a given linearization by swapping the linearization points of eligible and favorable dequeue linearization point pairs. We greedily decide the best pairs to swap based on the two operations' rank error improvement after the potential swap. Our implementation executes two passes of this method, one which swaps enqueue linearization points and one which swaps dequeue linearization points. This section presents and proves effectiveness for the algorithm which swaps dequeue linearization points. The algorithm for swapping enqueue linearization points can be described and reasoned about analogously.

In Algorithm 2, we present the core functionality of the Interchange algorithm applied to dequeue linearization points (here named *dequeue points* for conciseness). Utility functions are accounted for in Algorithm 3. The entire method is outlined as follows:

- A mapping from each item to all of its potential swap partners is initialized. As enforced on Line 3.2, criteria for an eligible pair includes that both items' dequeue points must be within both items' dequeue intervals, and neither item can have an enqueue point after the other's dequeue point. Initiating a dequeue point swap despite violation of these criteria would result in an invalid linearization. In Algorithm 2, the mapping E_x is initialized as a pre-processing step on Line 2.2.
- For each item, its *most* favorable (and thus far unswapped) partner is chosen for a dequeue point swap. Two items for which the sum of their dequeue operations' rank errors improves after switching dequeue points are considered favorable for a swap. The most favorable item is the one with the dequeue point resulting in the highest improvement after a swap, as defined on Line 2.12. Note that previously swapped items (Line 2.6) and partners (Line 2.9) are greedily excluded from selection. The swap itself is described on Line 3.11 and entails switching the two items' dequeue points in the linearization L.

Time complexity. Algorithm 2 has time complexity $O(n^3)$, where n is the number of items enqueued and dequeued in linearization L. This is easily derived from the $O(n^2)$ iteration over all eligible pairs on Lines 2.4 and 2.9 and the execution of the $O(n)$ function on Line 3.3 for each such pair.

4.5.1 Lower bound on the rank error change

The improvement in rank error results from situations where the two items' dequeue points are in a disadvantageous order but eligible for a swap. Figure 4.3 (where teal-colored lines mark linearization points) depicts such a situation. There, the

Algorithm 2: Interchange method applied to dequeue points

Data: start_x start time for the *dequeue* operation of item x
Data: end_x end time for the *dequeue* operation of item x
Data: D set of all items on which operations have been executed in the history
Data: L a set $\{(\text{enq}_x, \text{deq}_x) \mid x \in D\}$ containing enqueue and dequeue points as timestamps. The set represents a valid linearization of operations on all items in D
Result: L a set $\{(\text{enq}_x, \text{deq}_x) \mid x \in D\}$ representing a valid linearization

```
2.1  for x ∈ D do
2.2      E_x ← {y | y ∈ D ∧ swap_eligible(start_x, end_x, start_y, end_y, L)}
2.3  S ← ∅;                                              /* Track swapped items */
2.4  for x ∈ D do
2.5      if x ∈ S then
2.6          continue;
2.7      best_item ← None;
2.8      best_improvement ← 0;
2.9      for y ∈ E_x\S do
2.10         rank_error_before ← item_re(enq_x, deq_x, L) + item_re(enq_y, deq_y, L);
2.11         rank_error_after ← item_re(enq_x, deq_y, L) + item_re(enq_y, deq_x, L);
2.12         if rank_error_before − rank_error_after > best_improvement then
2.13             best_item ← y;
2.14             best_improvement ← rank_error_before − rank_error_after;
2.15     if best_item ≠ None then
2.16         L ← swap(x, best_item, L);
2.17     S ← S ∪ {x, best_item};
```

Algorithm 3: Utility functions for Interchange method

Data: $\text{start}_x, \text{start}_y$ start times for the dequeue operations of two items x, y
Data: $\text{end}_x, \text{end}_y$ end times for the dequeue operations of two items x, y
Data: L as defined in Algorithm 2
Result: True if x and y are eligible to be swapped, False otherwise
3.1 **Function** *swap_eligible* **is**
3.2 **return** $\neg((\text{start}_x > \text{deq}_y \vee \text{end}_x < \text{deq}_y \vee \text{start}_y > \text{deq}_x \vee \text{end}_y < \text{deq}_x) \vee (\text{deq}_x < \text{enq}_y \vee \text{deq}_y < \text{enq}_x))$; `/* not eligible if point outside interval or deq point before enq point */`

Data: enq_x, deq_x enqueue and dequeue points of an item x
Data: L as defined in Algorithm 2
Result: c the rank error of the dequeue operation of x
3.3 **Function** *item_re* **is**
3.4 $c \leftarrow 0$;
3.5 **for** $y \in L$ **do**
3.6 **if** $enq_y < enq_x \wedge deq_y > deq_x$ **then**
3.7 $c \leftarrow c + 1$;
3.8 **return** c

Data: x, y two items whose dequeue points should be swapped
Data: L as defined in Algorithm 2
Result: L as inputted but with swapped dequeue points for items x and y
3.9 **Function** *swap* **is**
3.10 $L \leftarrow (L \backslash \{(\text{enq}_x, \text{deq}_x), (enq_{best\_item}, deq_{best\_item})\}) \cup \{(\text{enq}_x, deq_{best\_item}), (enq_{best\_item}, \text{deq}_x)\}$;
3.11 **return** L

operations of items a and c are out-of-order but their dequeue points are eligible for a swap. The algorithm would swap the two dequeue points, since the total rank error improvement for the dequeue operations of a and c is 2. A swap results in the linearization in Figure 4.4, where the total rank error of all three dequeue operations have decreased from 3 to 0.

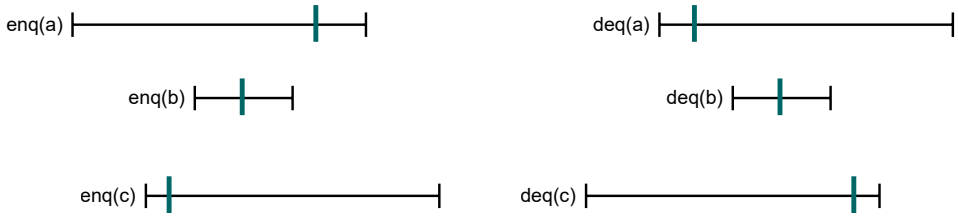

Figure 4.3: Linearization where the dequeue of item a has a rank error of 2 and the dequeue of item b has a rank error of 1.

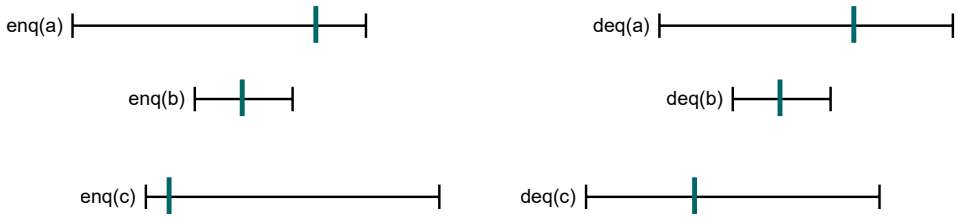

Figure 4.4: Linearization with no rank error.

The pair's total rank error was chosen as the metric for selection since it only requires two rank error calculations, avoiding higher time complexity. The reasoning below shows that this metric results in all swaps having a positive effect on the rank error (decreasing it).

Theorem 4.5.1. *Algorithm 2 produces a linearization with lower or equal total rank error over all dequeue operations, compared to the input linearization.*

We prove Theorem 4.5.1 by showing that a swap executed by the algorithm cannot increase the total rank error over all dequeue operations. The proof builds on the two following lemmas.

Lemma 4.5.2. *Any increase in rank error observed after a swap must be caused by dequeue points (which were not involved in the swap) becoming more out-of-order.*

Lemma 4.5.3. *After a swap, the rank error of any dequeue operation has only been affected by swapping the two dequeue points.*

Lemma 4.5.2 follows from Lines 2.8 and 2.12, which ensure that the summed rank error of the swapped points does not increase after a swap. Lemma 4.5.3 follows from the fact that all non-swapped points maintain their position in the linearization during a swap.

Proof of Theorem 4.5.1. Suppose that it is possible to swap two dequeue points (deq_a, deq_b) such that another dequeue point (deq_c) becomes *more* out-of-order. For such a scenario, the following premises must be true (per the definition of rank error in Section 2.4):

1. Either enq_a or enq_b (not both[2]) are before enq_c. Say $\text{enq}_a < \text{enq}_c < \text{enq}_b$.
2. Before the swap, the order of dequeue points is $\text{deq}_a < \text{deq}_c < \text{deq}_b$, to ensure that deq_c becomes more out-of-order after a swap of deq_a and deq_b.
3. After the swap, the order of dequeue points is $\text{deq}_b < \text{deq}_c < \text{deq}_a$. Thus, the rank error of deq_c has been increased by 1.

Such a scenario will never occur, by construction of Algorithm 2. Specifically, Lines 2.8 and 2.12 ensure that the summed rank error of two swapped points does not increase after a swap. Points deq_a and deq_b would not be swapped, as a swap increases their summed rank error from 0 to 2.

By Lemmas 4.5.2 and 4.5.3, there are no other scenarios where a rank error increase could be observed. Thus, we conclude that Algorithm 2 outputs a linearization which is *at least* as good as the input linearization.

□

4.5.2 Implementation details

Our implementation allows for running the core algorithm any user-specified number of times. This allows our implementation to iteratively improve the linearization, by feeding the solution from one iteration into the next iteration. Additionally, we implemented an algorithm analogous to that of Algorithm 2, which further improves the total rank error by swapping suitable enqueue points rather than dequeue points. The implementation runs both versions of the algorithm in each iteration.

Due to its complexity, the algorithm does not scale well when implemented naively. Our implementation suffered especially, as the multiple iteration strategy further worsened the time complexity. There are several potential code optimizations which may be implemented. We utilized pre-processing by storing eligible swaps and their rank error improvement. In every iteration, affected items are updated in the structure, rather than re-creating the structure every iteration. Effectively, the execution time for later iterations decreased notably (and valuably, even considering the time required for pre-processing). Additionally, we defined a stopping criteria which requires that some minimum number of swaps must be executed during an iteration to proceed with the next iteration. The number is user-defined in-code.

Providing the initial linearization. The method's result depends on the provided linearization. Naive algorithms present easily implemented starting points. For example, selecting interval end points as linearization points produces a suitable initial linearization. Other alternatives include the linearization produced by

[2]If both enq_a and enq_b are before enq_c, then swapping deq_a and deq_b does not affect the rank error of c's dequeue operation

the timestamping method. Which linearization we used as input is specified in Section 5.5.

5

Evaluation and Discussion

The aim of our algorithms is to produce a more accurate linearization than that produced by the timestamp approximation method (Section 2.2). Primary findings (Sections 5.2, 5.3, 5.4, 5.5) include comparison of performance between the timestamp approximation method and Multiple Probing, Linear Programming model, Ordering Algorithm and Interchange respectively. An overview of all methods' performances is provided in Section 5.6. Secondary findings (Sections 5.6.1, 5.6.2) are used to verify that our primary results are representative, despite the limited number of queue configurations accounted for. Additionally, in Section 5.6.3 we compare all methods to the Integer Programming model. This comparison is kept separate and considered a secondary finding, as the Integer Programming model is not applicable for histories of meaningful size.

As described in Chapter 4, each algorithm takes a history as input. Such a history is a recorded execution in which threads randomly either insert or remove items from a specific queue [30]. Which queue is used and specific queue configurations are defined for each execution. The history contains, for each operation, an item ID, which type of operation and the start and end times. We obtained histories from executions on three different queues: the relaxed 2Dd-queue (Section 2.4.1), the relaxed d-CBO queue (Section 2.4.2) and the strict FAAAQ (Section 2.3.1). All executions were performed using 32 hardware threads. The queue configurations used in this Chapter are defined in Section 5.1.

Longer histories were truncated to certain sizes (defined in Section 5.1), to ensure results could be generated within reasonable time even for the most inefficient algorithms. The truncation entails selecting dequeue operations in ascending order, as well as their corresponding enqueue operations until the desired (even) number of operations have been selected.

Performance metrics. The mean rank error across all dequeue operations in a linearization is our main performance metric, whereas maximum rank error is used to motivate and confirm algorithmic behaviors.

Queue selection. The 2Dd-queue and the d-CBO queue were selected for testing since they are among state-of-the-art for relaxed FIFO queues. Further, the 2Dd-queue has a worst-case relaxation guarantee whereas the d-CBO has an estimate bound. The FAAArrayQueue was selected as it performs on-par with state-of-the

art queues and has a favorable structure for which the theoretical linearization points can be identified and timestamped closely. Including a strict queue in the evaluation provided insight in our algorithms' performance on histories of operations with, theoretically, no relaxation.

Abbreviations. In this chapter, we abbreviate the methods as follows: Existing timestamping approximation method (TA), Multiple Probing (MP), Linear Programming model (LPM), Integer Programming model (IPM), Ordering Algorithm (OA), Interchange (IC).

5.1 Testing setup and linearization verification

For the primary results, we used the following parameters when obtaining and truncating the histories. For relaxed queues, five different configurations of sub-queues were used: 32, 64, 128, 256 or 512 sub-queues per relaxed queue[1]. The 2Dd-queue was configured to a depth of 32, and the d-CBO queue was configured to have a selection pool (the d in its name) of 2. Each history contained 50000 operations (half of them enqueues and half of them dequeues), obtained by truncating 2 millisecond executions. Table 5.1 provides a summary of these configurations.

We ran each algorithm three times per queue configurations, each execution resulting in a mean and a maximum rank error. The average mean and maximum rank error across all three executions is presented. This is done to ensure that our results are representative of each algorithm's effect, as algorithms may, due to scheduling, be biased towards some unique histories. We take this precaution only with our main findings in Sections 5.2, 5.3, 5.4, 5.5.

Queue	# sub-queues	Depth	d	# operations	# threads	Time (ms)
2Dd	32	32	-	50000	32	2
	64	32	-	50000	32	2
	128	32	-	50000	32	2
	256	32	-	50000	32	2
	512	32	-	50000	32	2
d-CBO	32	-	2	50000	32	2
	64	-	2	50000	32	2
	128	-	2	50000	32	2
	256	-	2	50000	32	2
	512	-	2	50000	32	2
FAAAQ	-	-	-	50000	32	2

Table 5.1: Configuration of queues for primary results presented in Sections 5.2, 5.3, 5.4 and 5.5.

For the secondary results, we investigated the effects of varying history length and varying execution time. In Section 5.6.1, histories of 25000, 50000 and 75000 oper-

[1]This applies for all algorithms except the ordering algorithm as specified in 5.4

ations were used. In Section 5.6.2, equally long histories obtained from 1, 2, 4 and 8 millisecond executions were used. Each history contained 10000 operations. For both sections' histories, the relaxed queues had 32 sub-queues each, the 2Dd-queue had a depth of 32 and the d-CBO had a selection pool of 2. These configurations are summarized in Table 5.2 and Table 5.3.

Queue	# sub-queues	Depth	d	# operations	# threads	Time (ms)
2Dd	32	32	-	25000	32	2
	32	32	-	50000	32	2
	32	32	-	75000	32	2
d-CBO	32	-	2	25000	32	2
	32	-	2	50000	32	2
	32	-	2	75000	32	2
FAAAQ	-	-	-	25000	32	2
	-	-	-	50000	32	2
	-	-	-	75000	32	2

Table 5.2: Configuration of queues for results presented in Section 5.6.1.

Queue	# sub-queues	Depth	d	# operations	# threads	Time (ms)
2Dd	32	32	-	10000	32	1
	32	32	-	10000	32	2
	32	32	-	10000	32	4
	32	32	-	10000	32	8
d-CBO	32	-	2	10000	32	1
	32	-	2	10000	32	2
	32	-	2	10000	32	4
	32	-	2	10000	32	8
FAAAQ	-	-	-	10000	32	1
	-	-	-	10000	32	2
	-	-	-	10000	32	4
	-	-	-	10000	32	8

Table 5.3: Configuration of queues for results presented in Section 5.6.2.

Linearization verification. The linearization created by each method is tested for correctness. We provide unit tests for testing a linearization with linearization points as timestamps or orders. For an order-based linearization, the following is tested:

- Each operation has been assigned an order.
- Each decided order is a valid choice for the operation.
- Each order is only assigned once.

For a timestamp-based linearization, the following is tested:

- Each enqueue point is within its start and end timestamps.

- Each dequeue point is within its start and end timestamps.
- Each enqueue point is before its corresponding dequeue point.
- Corresponding enqueue and dequeue points don't take the same timestamp.
- Timestamps are unique.

The linearizations produced MP, LPM and IC consistently pass all tests except for the last test. The latter two tests are not crucial. The last test fails since the nanoseconds used to record timestamps is not granular enough. Since time is continuous, corresponding enqueue and dequeue operations with the same timestamps could technically be properly ordered, if higher granularity in time metric was used. Non-corresponding operations which take the same timestamps can be arbitrarily ordered. We do this in our rank error calculation, which utilizes built-in Python sorting to order such operations.

Note that OA produces valid linearizations in some cases, and we only present results from valid linearizations.

Additionally, for the LPM solution, constraint violations are printed to enable manual checking. For linearizations produced by our algorithms, these tests showed negligible violations (typically less than 1e−8) which appear due to floating-point errors.

5.2 Multiple Probing

We measured the performance of the best naive implementation, Multiple Probing, with the purpose of evaluating its performance and potentially draw conclusions about similarities between this method and the timestamping approximation method.

Table 5.4 shows the performance of MP on histories derived from operations on the 2Dd-queue, where it is just outperformed by TA. The difference is largest for the maximum rank error on the 32 sub-queue configuration. This indicates that MP assigned some dequeue point significantly more out-of-order than the timestamp approximation. Possibly, one operation had an unusually long timespan before or after its linearization point. In such a situation, TA could approximate a more accurate point than MP, which would approximate a point long before or after its theoretical linearization point, resulting in a higher rank error for the dequeue operation.

	MP	TA	MP	TA
# sub-queues	Mean		Maximum	
32	152.58	**152.50**	1068.33	**908.33**
64	415.17	**415.12**	1150.67	**1149.67**
128	645.06	**644.84**	1209.00	**1207.67**
256	769.28	**769.11**	1177.67	**1177.00**
512	785.72	**785.34**	1221.67	**1220.67**

Table 5.4: Average mean and maximum rank error, across linearizations obtained by Multiple Probing, given histories of operations on the 2Dd-queue.

The results for histories obtained from operations on the d-CBO queue show similar performance between the two methods (see Table 5.5). TA outperforms MP slightly on one configuration (128 sub-queues) for the mean rank error and on three configurations (32, 128, 256 sub-queues) for the maximum rank error. However, the difference is at most 2 for the maximum rank error, and 0.01 for the mean rank error. On the other hand, MP outperforms TA in maximum rank error for two configurations (64, 512 sub-queues), where the largest difference is 36. The small difference in results could insinuate that the d-CBO queue is more consistent in the execution time of each operation than the 2Dd queue. Another possible explanation is that the enqueue and dequeue operations in the d-CBO queue have a closer relative linearization point than in the 2Dd queue, that is they are both approximately the same relative distance between the start and end of each operation whereas the difference might be higher for the 2Dd queue resulting in a higher rank error. Given histories obtained from operations on the FAAAQ, MP is also slightly out-performed by TA (see Table 5.6).

	MP	TA	MP	TA
# sub-queues	Mean		Maximum	
32	23.47	23.47	257.33	**256.33**
64	46.37	46.37	**457.67**	458.00
128	92.08	**92.07**	838.33	**836.67**
256	182.02	182.02	1136.33	**1134.33**
512	311.14	311.14	**1109.67**	1145.67

Table 5.5: Average mean and maximum rank error measured in linearizations obtained by Multiple Probing, given histories of operations on the d-CBO queue.

MP	TA	MP	TA
Mean		Maximum	
0.09	**0.06**	96.67	**96.33**

Table 5.6: Average mean and maximum rank error measured in linearizations obtained by Multiple Probing, given histories of operations on the FAAAQ.

We conclude that our naive methods for deciding linearization points using start and end timestamps do not produce linearizations with lower rank error across all dequeue operations than those produced by TA. However, from these intermediate results we infer that most operations likely take similarly long to execute and TA decides linearization points at similar relative points within all operation time intervals, which is similar to the point decided by MP. Perhaps the results of MP could have been slightly improved by a probing a higher number of increments between the start and end point but the trade of of runtime to improvement is not likely to be worth it.

5.3 Linear Programming model

For the evaluation of LPM, we re-iterate that LPM iteratively processes partial solutions for distinct sections of the full history. Each partial history contains 600 operations for the results obtained in this section, as decided on after brief experimentation.

Given histories derived from operations on the 2Dd-queue, LPM produces a slightly lower rank error for most configurations, except for 512 sub-queues (see Table 5.7). The difference in rank error between LPM and TA is (ascending, in number of sub-queues) 1.1, 1.71, 1.15, 0.66, -1.07. This indicates that LPM loses efficiency for a higher number of sub-queues in the 2Dd-queue. For a lower number of sub-queues, the partial solution structure of LPM can process operations within a large portion of a 2D window at a time, which would include out-of-order operations which were executed similarly in time (and could thus be decided more accurately by the objective function). For larger number of sub-queues, a partial solution in LPM can only contain operations which fill some of the width in a 2D window. The potential for deciding more accurate points could thus be limited.

	LPM	TA	LPM	TA
# sub-queues	Mean		Maximum	
32	**151.40**	152.50	**907.33**	908.33
64	**413.41**	415.12	1151.00	**1149.67**
128	**643.69**	644.84	1209.67	**1207.67**
256	**768.45**	769.11	1181.67	**1177.00**
512	786.41	**785.34**	1225.67	**1220.67**

Table 5.7: Average mean and maximum rank error measured in linearizations obtained by the Linear Programming model (with partial history size 600), given histories of operations on the 2Dd-queue.

For histories of operations on the d-CBO queue, the LPM consistently outperforms TA in both mean and maximum rank error. The improvement in mean rank error is, per configuration (ascending, in number of sub-queues), 3.19, 2.96, 2.7, 2.51, 3.77. This is a notable improvement, as the 32 sub-queue configuration result in 14% lower rank error using LPM compared to TA. The d-CBO queue distributes

operations evenly across sub-queues (based on operation counts). A history with evenly distributed operations is likely a beneficial input for LPM, as each partial solution will contain operations which were executed closely in time and can be assigned more accurate linearization points.

	LPM	TA	LPM	TA
# sub-queues	Mean		Maximum	
32	**20.28**	23.47	**248.33**	256.33
64	**43.41**	46.37	**448.67**	458.00
128	**89.37**	92.07	**825.33**	836.67
256	**179.51**	182.02	**1122.33**	1134.33
512	**307.37**	311.14	**1105.00**	1145.67

Table 5.8: Average mean and maximum rank error measured in linearizations obtained by the Linear Programming model (with partial history size 600), given histories of operations on the d-CBO queue.

Regarding the histories derived from operations on the FAAAQ, Table 5.9 shows that the LPM cannot maintain as low of a mean rank error as the TA, but it produces a linearization with much lower maximum rank error. One or a few operations are likely significantly out-of-order in the TA linearization, but most operations are not. However, in the LPM linearization, more operations are out-of-order but no operation is as gravely out-of-order. It is easy to see that the objective function of LPM could minimize maximum rank error given favorable input, as it can optimize linearization points individually for each operation in a way that TA cannot.

LPM	TA	LPM	TA
Mean		Maximum	
1.25	**0.06**	**55.00**	96.33

Table 5.9: Average mean and maximum rank error measured in linearizations obtained by the Linear Programming model(with partial history size 600), given histories of operations on the FAAAQ.

We conclude that LPM mostly outperforms TA when measuring mean rank error over all dequeue operations in linearizations of 2Dd and d-CBO queues. However, for linearizations of the strict FAAAQ, TA maintains a lower mean rank error, while LPM can achieve a significantly lower maximum rank error.

5.4 Ordering Algorithm

Unlike the results of the other algorithms, the results from this algorithm was created by running the algorithm on histories of size 10000 operations, and with window depth 32 for the 2Dd queue, on 32 threads. The reason for this was the late realization that the algorithm is unable to generate a result for most of the specific files

that are used for the other algorithms. In order to be able to calculate average improvements over three files the option with the most passing files was used instead. The results of the files that did run can be found in the 5.6 section.

In Table 5.10 we can see that the algorithm is able to achieve some improvement in both mean and max rank error, over the timestamp approximation, for the relaxed data structures. On the other hand the results for the faaaq are significantly worse than the timestamp approximation results.

One probable reason for the poor results in the faaaq is the last assignment step of the algorithm utilizing randomness which means that we are not trying to minimize the max rank error.

Interestingly the ordering algorithm seems to achieve a better result on different aspects for the two relaxed data structures. For the d-CBO queue the improvement in mean rank error is 4.4% while the improvement in maximum rank error is only 1.4%. On the other hand in the 2Dd queue the mean rank error improvement was only 0.14% while the improvement in max rank error was 1.9%. However since the sample size is so small it is non conclusive weather this can be considered a pattern.

	OA	TA	OA	TA
queue type	Mean		Maximum	
d-CBO	**21.91**	22.90	**192.33**	195
2Dd	**190.70**	190.96	**920.33**	938.33
faaaq	4.09	**0.03**	54.69	**7.75**

Table 5.10: Average mean and maximum rank error measured in linearizations obtained by Ordering Algorithm, given histories of operations on the all queues with 32 sub queues and file size of 10000 operations. Observe this is not based on the same data as the results in Sections 5.2, 5.3, 5.5.

5.5 Interchange

IC utilized linearizations obtained from TA as its starting point. Brief testing showed that using that linearization gave better results than using the linearization of any of our naive algorithms. After applying IC, the resulting linearization is at least as good as the inputted, but we show experimentally that it is typically better.

For a linearization derived from operations on the 2Dd-queue, the IC improves the mean rank error by approximately 4, regardless of sub-queue configuration. Table 5.11 depicts the improvement, which also shows that IC has a slightly lower maximum rank error than TA across all sub-queue configurations as well.

	IC	TA	IC	TA
# sub-queues	Mean		Maximum	
32	**148.61**	152.50	**902.33**	908.33
64	**411.22**	415.12	**1144.33**	1149.67
128	**641.05**	644.84	**1202.33**	1207.67
256	**765.10**	769.11	**1169.67**	1177.00
512	**781.40**	785.34	**1213.67**	1220.67

Table 5.11: Average mean and maximum rank error measured in linearizations obtained by Interchange (with TA linearization as starting point), given histories of operations on the 2Dd-queue.

The improvement is similar for a linearization obtained from operations on the d-CBO queue (Table 5.12). The mean rank error is improved by approximately 4 for all sub-queue configurations. This is approximately an 18% improvement for the 32 sub-queue configuration, but less impactful for those with more sub-queues (and thus higher rank error overall).

The maximum rank error shows larger improvements for the d-CBO than for the 2Dd-queue, which may be attributed to the stochasticity of the d-CBO. Perhaps randomizing a pool for selection can result in a disadvantageous operation order of operations occurring close in time, such that they may be swapped for an improvement. The 2Dd-queue's systematic approach of searching for sub-queues may perhaps result in operations with high individual rank errors to be executed less closely in time, thus limiting its possibility for improvement by swapping points.

	IC	TA	IC	TA
# sub-queues	Mean		Maximum	
32	**19.35**	23.47	**248.33**	256.33
64	**42.09**	46.37	**447.67**	458.00
128	**87.79**	92.07	**826.00**	836.67
256	**177.74**	182.02	**1129.00**	1134.33
512	**307.00**	311.14	**1101.00**	1145.67

Table 5.12: Average mean and maximum rank error measured in linearizations obtained by Interchange (with TA linearization as starting point), given histories of operations on the d-CBO queue.

IC does not notably improve the linearization obtained from the FAAAQ, indicating that the method finds local optima but cannot be used as a general method of finding a global optima (which may be naturally derived by the greedy and iterative structure of the algorithm).

IC	TA	IC	TA
Mean		Maximum	
0.06	0.06	**95.00**	96.33

Table 5.13: Average mean and maximum rank error measured in linearizations obtained by Interchange (with TA linearization as starting point), given histories of operations on the FAAAQ.

Given a linearization obtained using TA, IC improves the mean rank error across its dequeue operations by approximately 4, across all configurations of the 2Dd and d-CBO queues. For the 32 sub-queue configuration of d-CBO, this results in an 18% improvement in mean rank error.

5.6 Method comparison

In this section, we compare the performances of each of our methods, with TA as the benchmark value. Additionally, in Sections 5.6.1 and 5.6.2, we test variations of two key aspects which may be effecting the results presented here. Finally, in Section 5.6.3, we produce few results on small linearizations, to compare the IPM solution to the other methods.

The results presented below are measured as increase or decrease in rank error (compared to TA), where a decrease (negative number) is desired. Note that each linearization contains 25000 dequeue operations, hence a mean decrease of 1 is equivalent to a total rank error decrease of 25000. We compare the results of TA, MP, LPM, OA and IC on histories obtained from operations on the 2Dd-queue (Table 5.14), the d-CBO queue (Table 5.15) and the FAAAQ (Table 5.16). Note that this is an overview of the results presented in Sections 5.2, 5.3 and 5.5.

IC outperforms all other methods in mean rank error, across all sub-queue configurations. It maintains an improvement of around 4 regardless of queue and configuration, as can be observed in the sixth column of Tables 5.14, 5.15 and 5.16. Further, IC outperforms MP, OA and LPM in maximum rank error on histories derived from operations on the 2Dd-queue. However, LPM and IC perform similarly well considering maximum rank error on histories derived from the d-CBO queue, each outperforming the other on a few configurations.

	TA	MP	LPM	OA	IC	TA	MP	LPM	OA	IC
# Sub-queues	Mean					Maximum				
32	152.50	+0.08	-1.1	-2.26[1]	**-3.89**	908.33	+160	-1	+30.67[1]	**-6**
64	415.12	+0.05	-2.09	-	**-3.9**	1149.67	+1	+1.33	-	**-5.3**4
128	644.84	+0.22	-1.15	-	**-3.79**	1207.67	+1.33	+2	-	**-5.34**
256	769.11	+0.17	-0.66	-	**-4.01**	1177.00	+0.67	+4.67	-	**-7.33**
512	785.34	+0.38	+1.07	-	**-3.94**	1220.67	+1	+5	-	**-7**

Table 5.14: Comparison of average mean and maximum rank error obtained from different methods, given histories of operations on the 2Dd-queue, relative to TA.

	TA	MP	LPM	OA	IC	TA	MP	LPM	OA	IC
# Sub-queues	Mean					Maximum				
32	23.47	±0	-3.19	-1.01[1]	**-4.12**	256.33	+1	**-8**	2[1]	**-8**
64	46.37	±0	-2.96	-	**-4.28**	458.00	-0.33	-9.33	-	**-10.33**
128	92.07	+0.01	-2.7	-	**-4.28**	836.67	+1.66	**-11.34**	-	-10.67
256	182.02	±0	-2.51	-2.54[1]	**-4.28**	1134.33	+2	**-12**	-22.33[1]	-5.33
512	311.14	±0	-2.76	-	**-4.14**	1145.67	-36	-40.67	-	**-44.67**

Table 5.15: Comparison of average mean and maximum rank error obtained from different methods, given histories of operations on the d-CBO queue, relative to TA.

Table 5.16 shows that no other method can outperform TA in mean rank error across dequeue operations in a linearization of the FAAAQ. The rank error occurs only due to approximating the linearization points, causing some approximated points to be out-of-order. Theoretically, a linearization of the FAAAQ has no rank error. TA is able to make a better approximation than MP and LPM with regard to mean rank error. However, the LPM method finds a linearization with notably lower maximum rank error, which is understandable considering its method of minimizing relative distances between corresponding enqueues and dequeues, which may affect the maximum rank error notably when there are few operations with high rank error.

TA	MP	LPM	OA	IC	TA	MP	LPM	OA	IC
Mean					Maximum				
0.06	+0.03	+1.19	-	**±0**	96.33	+0.34	**-41.33**	-	-1.33

Table 5.16: Comparison of average mean and maximum rank error obtained from different methods, given histories of operations on the FAAAQ, relative to TA.

5.6.1 Effects of varying history length

The histories utilized to obtain the primary results presented above each contained 50000 operations. These were truncated by selecting the 25000 first enqueue operations and their corresponding dequeue operations. In this section, we aim to investigate if this truncation may affect the results notably. For example, if a larger history may result in higher relative rank error improvement, we may conclude that our previous results were understated.

We proceed by presenting the relative difference in rank error as a percentage of the TA method's result, for histories of varying lengths. A negative percentage indicates a decrease in rank error. A decrease by 5% indicates that the method results in a rank error which is 95% of the rank error resulting from TA. The desired result in this section is a similar relative difference in rank error for all history sizes, as it would indicate that our main findings should be representative for longer histories (as

[1]This value corresponds to only one or two files not the normal three which means that the numerical comparison is not completely accurate.

are often derived from executions). The following results have only been obtained from one history per configuration. Results have been obtained for histories of 5000, 10000, 25000 and 75000 operations, as well as a "complete" history obtained from a 2 ms execution. These complete histories contain 113000, 178000 and 98000 operations for the 2Dd-queue, d-CBO queue and FAAAQ respectively. Results for IC has not been obtained for the complete histories, as we were restricted by its high complexity and computation time.

Given histories of operations on the 2Dd-queue, we do not note any significant, deviating results (see Table 5.17). Small deviations are difficult to measure, as the TA results in a high mean and maximum rank error for such histories.

	TA	MP	LPM	OA	IC	TA	MP	LPM	OA	IC
# Operations	Mean					Maximum				
5000	229.73	±0%	-1%	±0%	-1%	949	±0%	±0%	±0%	±0%
10000	196.17	±0%	-1%	±0%	-2%	949	±0%	±0%	±0%	±0%
25000	165.41	±0%	±0%	±0%	-2%	949	±0%	±0%	±0%	±0%
50000	154.50	±0%	-1%	±0%	-2%	949	±0%	±0%	±0%	±0%
75000	147.42	±0%	-1%	-	-3%	949	±0%	±0%	-	±0%
113000	145.20	±0%	-1%	-	-	949	±0%	±0%	-	-

Table 5.17: Given differently sized histories of operations on the 2Dd-queue: mean and maximum rank error obtained from different methods, relative to the TA method.

Table 5.18 shows the relative increase or decrease given different history sizes on the d-CBO queue. For the mean rank error, the MP, LPM and IC methods all perform similarly regardless of history size. For the maximum rank error, however, both MP and LPM show much worse results (significant increase in maximum rank error) for the full history. This may be due to the fact that a full history allows for an operation to be much more out-of-order. MP chooses a relative point to optimize mean rank error, which does not specifically handle outliers which are significantly out-of-order. The LPM solution may be limited by its partial solution implementation in achieving a low maximum rank error for very large files. IC does not have a significant increase in any of its presented results, however, we do not present its performance for a full history and may miss a potential deviance there.

	TA	MP	LPM	OA	IC	TA	MP	LPM	OA	IC
# Operations	Mean					Maximum				
5000	22.39	±0%	-6%	-4%	-10%	197	+1%	-4%	-4%	-3%
10000	22.62	±0%	-9%	-	-4%	207	±0%	-4%	-	-6%
25000	23.23	±0%	-3%	-	-6%	207	±0%	-4%	-	-6%
50000	23.52	±0%	-3%	-	-7%	207	±0%	-4%	-	-6%
75000	23.60	±0%	-3%	-	-7%	207	±0%	-4%	-	-6%
178000	23.61	±0%	-6%	-	-	207	+141%	+69%	-	-

Table 5.18: Given differently sized histories of operations on the d-CBO queue: mean and maximum rank error obtained from different methods, relative to the TA method.

Results for the FAAAQ (see Table 5.19) show much more deviation between history lengths, likely due to the low rank error provided by the TA method being used as comparison. For the two smallest histories, MP and LPM deviate significantly for both mean and maximum rank error. IC deviates in mean rank error for these, and somewhat in maximum rank error for the second smallest history. For larger histories, MP and IC are quite stable in their performance for mean and maximum rank error. LPM only performs similarly in mean rank error for three of the histories, and in maximum rank error for two of them. LPM deviates significantly in mean rank error for the largest history, where TA is able to maintain a low rank error which LPM cannot perform similarly to. OA has similarly poor performance here as in for the earlier FAAAQ examples.

	TA	MP	LPM	OA	IC	TA	MP	LPM	OA	IC
# Ops	Mean					Maximum				
5000	<0.01	+1011%	+39344%	+74622%	-99%	1	+100%	+2900%	-6200%	±0%
10000	<0.01	+1011%	+32400%	+102956%	-94%	2	+100%	+1400%	+3400%	-50%
25000	0.09	+33%	+989%	+5300%	±0%	15	-7%	+100%	+393%	±0%
50000	0.12	+33%	+625%	-	-8%	240	±0%	-83%	-	±0%
75000	0.12	+33%	+633%	-	-8%	240	±0%	-83%	-	±0%
98000	0.12	+33%	+3833%	-	-	240	±0%	+6%	-	-

Table 5.19: Given differently sized histories of operations on the FAAAQ: mean and maximum rank error obtained from different methods, relative to the TA method.

Generally, we see that the methods perform similarly (relative TA) regardless of history size. Notable outliers include MP and LPs performance for the entire d-CBO history, which worsened significantly for the maximum rank error recorded. Additionally, the mean and maximum rank error resulting from LPM showed great variation in the FAAAQ histories. We do not observe any trends which indicate that the methods perform better or worse on larger histories than on smaller. We reiterate that each algorithm was only ran once per configuration, hence we see a need for further analysis of performance robustness.

5.6.2 Effects of varying execution time

All primary results are obtained from histories based on executions of random operations on queues during 2 ms. The sizes of the histories obtained from such executions do not increase linearly with an increased execution time (history sizes increase slower). Thus, the operations may be distributed differently in a history obtained from a longer execution than one obtained from a shorter execution. In this section, we aim to investigate if the execution time affects a method's relative increase or decrease in rank error compared to TA, with the same measurements as in the previous section. Each algorithm was run once per configuration. Each history contained 10000 operations. For the 2Dd-queue and d-CBO queue, 32 sub-queues were used.

All methods perform quite similarly regardless of execution time for both the history obtained from the 2Dd-queue (see Table 5.20) and the d-CBO queue (see Table 5.21). We note a slight deviance for the 2Dd-queue history based on a 1 ms execution, where MP, LPM, OA and IC all achieve a decrease in maximum rank error which is not achieved for the longer execution histories. This may be due to bias in our methods towards the specific history used. In the case of maximum rank error, it is sufficient for one operation to be approximated badly by TA (e.g. due to scheduling) but approximated well by our methods for such a result.

	TA	MP	LPM	OA	IC	TA	MP	LPM	OA	IC
Time (ms)	Mean					Maximum				
1	185.17	0%	-1%	0%	-1%	949	-6%	-7%	-6%	-7%
2	196.17	0%	-1%	0%	-2%	949	0%	0%	0%	0%
4	183.38	0%	-1%	-	-2%	833	+1%	0%	-	0%
8	191.55	0%	-1%	0%	-2%	917	0%	-1%	0%	0%

Table 5.20: Comparison of mean and maximum rank error obtained from different methods, for different execution times, given histories of operations on the 2Dd-queue.

	TA	MP	LPM	OA	IC	TA	MP	LPM	OA	IC
Time (ms)	Mean					Maximum				
1	22.61	0%	-12%	-	-16%	183	+1%	-10%	-	-3%
2	22.62	0%	-9%	-5%	-14%	207	0%	-4%	-4%	-6%
4	22.93	0%	-12%	-4%	-15%	200	-1%	-7%	0%	-5%
8	23.14	0%	-11%	-4%	-14%	178	0%	-5%	-2%	-3%

Table 5.21: Comparison of mean and maximum rank error obtained from different methods, for different execution times, given histories of operations on the d-CBO queue.

We do not observe any clear trends in results derived from the FAAAQ (see Table 5.22). Given histories obtained from shorter executions, the MP and LPM deviate in

mean rank error. They achieve a low mean rank error, however, as the TA achieves a much lower rank error, the relative mean rank error is high.

	TA	MP	LPM	OA	IC	TA	MP	LPM	OA	IC
Time (ms)	Mean					Maximum				
1	<0.01	+669%	+47977%	+164900%	-85%	2	+50%	+2650%	+3550%	-50%
2	<0.01	+1011%	+32400%	+104067%	-94%	2	+100%	+1400%	+3400%	-50%
4	0.07	+38%	+2200%	+5534%	-17%	16	+6%	+194%	+338%	-6%
8	0.05	+1334%	+3748%	+9096%	-13%	17	+200%	+364%	-48%	-9%

Table 5.22: Comparison of mean and maximum rank error obtained from different methods, for different execution times, given histories of operations on the FAAAQ.

The methods perform similarly (relative TA) regardless of execution time for the relaxed queues. The 1 ms execution of operations on the 2Dd-queue is an outlier, where the maximum rank error recorded for all our methods is lower than expected, likely due to scheduling in that specific history. We do not observe any trends which indicate that the methods perform consistently better or worse on histories based on longer or shorter executions. We reiterate that each method was only ran once per configuration, hence we see a need for further analysis of performance robustness.

5.6.3 Comparison to IPM

In this section, we make a brief comparison of results of the IPM and the other methods, using single histories for the 2Dd-queue, d-CBO queue and FAAAQ. We use extremely short histories of 20 operations each and present the mean and maximum rank error obtained. The IPM has a quadratic memory consumption in number of operations per history, hence computing larger histories requires unproportionate resources.

As the files are very short, most methods will produce a linearization with fairly low rank error, and comparison may be difficult. However, a brief comparison is still interesting, as the IPM produces an optimal linearization for a history of operations on a strict queue, and likely produces a close-to-optimal linearization for a history of operations on a relaxed queue. All histories are obtained from a 2 ms execution, and the 2Dd-queue and d-CBO queue both utilize 32 sub-queues. We present the results compared to the rank error achieved by the IPM, and a low number is desirable.

We observe from these results that none of the methods TA, MP, LPM or IC consistently achieve the minimum possible mean and maximum rank error for all queues, even for such a small problem as used here. However, for this small problem, the LPM and IC perform most closely to IPM.

	IPM	TA	MP	LPM	OA	IC	IPM	TA	MP	LPM	OA	IC
Queue	Mean						maximum					
2Dd	2.2	+0.2	+0.1	±0	+0.1	±0	5	+1	+1	±0	+1	±0
d-CBO	1.6	+0.7	+0.5	+0.3	+0.2	+0.3	4	+1	±0	±0	±0	±0
FAAAQ	0	±0	±0	±0	±0	±0	0	±0	±0	±0	±0	±0

Table 5.23: Comparison of total and maximum rank error obtained from IPM and all other methods, for 20 operation histories of operations on the 2Dd-queue, d-CBO queue and FAAAQ.

6

Conclusion

In this thesis, we showed that there exists a linearization with lower rank error across all dequeue operations than that provided by the current timestamping method (TA). Specifically, we developed methods for finding optimized linearizations with regard to the mean rank error across dequeue operations, given histories of operations on the relaxed 2Dd-queue and d-CBO queue. Additional results were obtained for histories of operations on the strict FAAAQ. We presented three principle methods, two of which (LP, OA) produce a linearization from a history and one (IC) which improves an existing linearization.

Our primary results show that LP and IC produce linearizations with lower rank error summed across all dequeue operations than those produced by TA. No presented method could, significantly, outperform TA in mean rank error for dequeue operations in linearizations of FAAAQ. However, given histories of operations on the relaxed 2Dd-queue and d-CBO, LP out-performed TA in mean rank error for most configurations. IC out-performed LP and TA in mean rank error for all configurations. IC produced a linearization with up to 18% lower mean rank error (across all dequeue operations) than TA, when given a history of operations on a d-CBO queue with 32 sub-queues.

From our secondary results we infer that the primary results are reliable and representative of the methods' performances when observing mean rank error in relaxed queues. Neither the length of a history nor the length of the execution time used to obtain a truncated history significantly affects the relative performance of our methods compared to TA. We could not assert these results for the FAAAQ. We make these conclusions but note a need for further analysis of performance robustness. Further, we show that none of our three primary methods, nor TA, are certain to provide an optimal solution. Hence, there may be larger discrepancy between mean rank error across dequeue operations in the linearization produced by TA and an optimal solution than we have been able to show.

We conclude that relaxed FIFO queues may cause less rank error than is commonly measured, motivating further research on this topic.

6.1 Future work

Our findings indicate that more accurate linearizations can be obtained than those of a currently common method. Although our algorithms can be re-created or downloaded, we see the need for a more extensive framework for researchers to test relaxed queues. Such a framework should preferably include a variety of efficient algorithms adapted for different types of queues. Using a more efficient way to calculate rank error, such as [35], could be valuable for increasing efficiency in our algorithms and other similar implementations. Additionally, a more extensive survey similar to our thesis, but including all top-performing relaxed FIFO queues, would provide useful data for research on relaxed queues.

Regarding our methods, the main area of improvement may be found in Interchange. The Interchange algorithm may be further tested to investigate its potential. As the optimization finds a local optimum, it may be possible to produce even more accurate linearizations by providing other initial linearizations. For example, using a stochastic algorithm to provide the initial linearization may help the Interchange algorithm find better local optima. Additionally, implementing a partial solution structure for Interchange may enable processing larger histories faster.

We end this section with a final reflection on the thesis. Our approach to this project was practical rather than theoretical. A different approach to this problem would be to extend existing methods for linearizing strict data structures, such as Lowe's automaton-based method [24], to enable linearizing relaxed data structures with the goal to minimize rank error. Such an approach would require deep knowledge in the area but may result in more predictable and efficient algorithms.

Bibliography

[1] N. Shavit, "Data structures in the multicore age," *Commun. ACM*, vol. 54, no. 3, pp. 76–84, Mar. 2011, ISSN: 0001-0782. DOI: 10.1145/1897852.1897873. [Online]. Available: https://doi.org/10.1145/1897852.1897873.

[2] 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*, ser. PODC '96, Philadelphia, Pennsylvania, USA: Association for Computing Machinery, 1996, pp. 267–275, ISBN: 0897918002. DOI: 10.1145/248052.248106. [Online]. Available: https://doi.org/10.1145/248052.248106.

[3] P. Ramalhete, *Faaarrayqueue - mpmc lock-free queue (part 4 of 4)*, https://concurrencyfreaks.blogspot.com/2016/11/faaarrayqueue-mpmc-lock-free-queue-part.html, 2016.

[4] 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*, ser. POPL '13, Rome, Italy: Association for Computing Machinery, 2013, pp. 317–328, ISBN: 9781450318327. DOI: 10.1145/2429069.2429109. [Online]. Available: https://doi.org/10.1145/2429069.2429109.

[5] 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.

[6] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, Jul. 1990, ISSN: 0164-0925. DOI: 10.1145/78969.78972. [Online]. Available: https://doi.org/10.1145/78969.78972.

[7] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming, Revised Reprint*, 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012, ISBN: 9780123973375.

[8] A. Haas, M. Lippautz, T. A. Henzinger, *et al.*, "Distributed queues in shared memory: Multicore performance and scalability through quantitative relaxation," in *Proceedings of the ACM International Conference on Computing Frontiers*, ser. CF '13, Ischia, Italy: Association for Computing Machinery, 2013, ISBN: 9781450320535. DOI: 10.1145/2482767.2482789. [Online]. Available: https://doi.org/10.1145/2482767.2482789.

[9] 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, ser. PPoPP '25, Las Vegas, NV, USA: Association for Computing Machinery, 2025, pp. 382–395, ISBN: 9798400714436. DOI: 10.1145/3710848.3710892. [Online]. Available: https://doi.org/10.1145/3710848.3710892.

[10] M. Williams, P. Sanders, and R. Dementiev, "Engineering MultiQueues: Fast Relaxed Concurrent Priority Queues," in *29th Annual European Symposium on Algorithms (ESA 2021)*, P. Mutzel, R. Pagh, and G. Herman, Eds., ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 204, Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, 81:1–81:17, ISBN: 978-3-95977-204-4. DOI: 10.4230/LIPIcs.ESA.2021.81. [Online]. Available: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ESA.2021.81.

[11] C. M. Kirsch, H. Payer, H. Röck, and A. Sokolova, "Performance, scalability, and semantics of concurrent fifo queues," in *Algorithms and Architectures for Parallel Processing*, Y. Xiang, I. Stojmenovic, B. O. Apduhan, G. Wang, K. Nakano, and A. Zomaya, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 273–287, ISBN: 978-3-642-33078-0.

[12] A. Rukundo, A. Atalar, and P. Tsigas, "Monotonically relaxing concurrent data-structure semantics for increasing performance: An efficient 2d design framework," in *33rd International Symposium on Distributed Computing*, 2019.

[13] Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Transactions on Computers*, vol. C-28, no. 9, pp. 690–691, 1979. DOI: 10.1109/TC.1979.1675439.

[14] J. Derrick, B. Dongol, G. Schellhorn, B. Tofan, O. Travkin, and H. Wehrheim, "Quiescent consistency: Defining and verifying relaxed linearizability," in *FM 2014: Formal Methods*, C. Jones, P. Pihlajasaari, and J. Sun, Eds., Cham: Springer International Publishing, 2014, pp. 200–214, ISBN: 978-3-319-06410-9.

[15] M. D'Antonio, K. von Geijer, T. S. Mai, P. Tsigas, and H. Vandierendonck, "Relax and don't stop: Graph-aware asynchronous sssp," in *Proceedings of the 1st FastCode Programming Challenge*, ser. FCPC '25, The Westin Las Vegas Hotel & Spa, Las Vegas, NV, USA: Association for Computing Machinery, 2025, pp. 43–47, ISBN: 9798400714467. DOI: 10.1145/3711708.3723446. [Online]. Available: https://doi.org/10.1145/3711708.3723446.

[16] 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.

[17] J. Matouek and B. Gärtner, *Understanding and Using Linear Programming (Universitext)*. Berlin, Heidelberg: Springer-Verlag, 2006, ISBN: 3540306978.

[18] A. Koberstein, "The dual simplex method, techniques for a fast and stable implementation," Ph.D. dissertation, Paderborn University, Germany, 2005.

[19] C.-Y. Huang, C.-Y. Lai, and K.-T. Cheng, "Chapter 4 - fundamentals of algorithms," in *Electronic Design Automation*, L.-T. Wang, Y.-W. Chang, and K.-T. Cheng, Eds., Boston: Morgan Kaufmann, 2009, pp. 173–234, ISBN: 978-0-

12-374364-0. DOI: https://doi.org/10.1016/B978-0-12-374364-0.50011-4. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780123743640500114.

[20] J. E. Mitchell, *Branch and cut**, May 12, 2010. [Online]. Available: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=6436117a15a75a544b693b4f1cf472a00e799734 (visited on 08/20/2025).

[21] Q. Huangfu and J. J. Hall, "Parallelizing the dual revised simplex method," *Mathematical Programming Computation*, vol. 10, no. 1, pp. 119–142, 2018.

[22] T. Achterberg, "Scip-a framework to integrate constraint and mixed integer programming," 2004.

[23] E. F. Inc. "*SCIP* [Online]." [Accessed: 21 May 2025]. ().

[24] G. Lowe, "Testing for linearizability," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 4, e3928, 2017.

[25] M. Emmi and C. Enea, "Sound, complete, and tractable linearizability monitoring for concurrent collections," *Proc. ACM Program. Lang.*, vol. 2, no. POPL, Dec. 2017. DOI: 10.1145/3158113. [Online]. Available: https://doi.org/10.1145/3158113.

[26] D. Alistarh, T. Brown, J. Kopinsky, J. Z. Li, and G. Nadiradze, "Distributionally linearizable data structures," in *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, 2018, pp. 133–142.

[27] L. Zhang, A. Chattopadhyay, and C. Wang, "Round-up: Runtime verification of quasi linearizability for concurrent data structures," *IEEE Transactions On Software Engineering*, vol. 41, no. 12, pp. 1202–1216, 2015.

[28] C. Wang, Y. Lv, and P. Wu, "Decidability of linearizabilities for relaxed data structures," *Science China Information Sciences*, vol. 61, no. 1, p. 012 103, 2018.

[29] I. Dahl and H. Schaff, *Minimizing-relaxation-errors*, https://github.com/minimizing-relaxation-errors, 2025.

[30] K. von Geijer and A. Rukundo, *Semantic-relaxation-dcbo*, https://github.com/dcs-chalmers/semantic-relaxation-dcbo, 2024.

[31] S. Diamond and S. Boyd, "CVXPY: A Python-embedded modeling language for convex optimization," *Journal of Machine Learning Research*, vol. 17, no. 83, pp. 1–5, 2016.

[32] A. Agrawal, R. Verschueren, S. Diamond, and S. Boyd, "A rewriting system for convex optimization problems," *Journal of Control and Decision*, vol. 5, no. 1, pp. 42–60, 2018.

[33] G. Gamrath, T. Fischer, T. Gally, *et al.*, "The scip optimization suite 3.2," 2016.

[34] S. Maher, M. Miltenberger, J. P. Pedroso, D. Rehfeldt, R. Schwarz, and F. Serrano, "PySCIPOpt: Mathematical programming in python with the SCIP optimization suite," in *Mathematical Software – ICMS 2016*, Springer International Publishing, 2016, pp. 301–307. DOI: 10.1007/978-3-319-42432-3_37.

[35] E. Kleen and V. Olin, "Efficiently calculating rank errors for relaxed fifo queues," M.S. thesis, Chalmers University of Technology, 2025.